

Árvai Zoltán • Fár Attila Gergő • Farkas Bálint  
Fülöp Dávid • Komjáthy Szabolcs • Turóczy Attila • Velvárt András

# WINDOWS PHONE

## FEJLESZTÉS LÉPÉSRŐL LÉPÉSRE



**Microsoft**



**Windows Phone**

Put people first.



# **Windows Phone fejlesztés lépésről lépésre**

***Készült a Microsoft Magyarország megbízásából***

**Árvai Zoltán – Fár Attila Gergő – Farkas Bálint – Fülöp Dávid –  
Komjáthy Szabolcs – Turóczy Attila – Velvárt András**

# **Windows Phone fejlesztés lépésről lépésre**



**Jedlik Oktatási Stúdió  
Budapest, 2012**





A szerzők a könyv írása során törekedtek arra, hogy a leírt tartalom a lehető legpontosabb és naprakész legyen. Ennek ellenére előfordulhatnak hibák, vagy bizonyos információk elavulttá válhattak.

A könyvben leírt programkódokat mindenki saját felelősségére alkalmazhatja. Javasoljuk, hogy ezeket ne éles környezetben próbálják ki. A felhasználásból eredő esetleges károkért sem a szerzők, sem a kiadó nem vonható felelősségre. A forráskódok elérhetők a <https://devportal.hu/wp7/konyv> oldalon keresztül.

Az oldalakon előforduló márka- valamint kereskedelmi védjegyek bejegyzőjük tulajdonában állnak.

***A könyv vagy annak bármely része, valamint a benne szereplő példák a szerzőkkel kötött megállapodás nélkül nem használhatók fel üzleti célú oktatási tevékenység során! A könyv tudásanyaga államilag finanszírozott közép- és felsőoktatásban, illetve szakmai közösségek oktatásában bármely célra felhasználható.***

© 2011 Microsoft. Minden jog fenntartva.

**A könyv papír alapon megvásárolható a Jedlik Oktatási Stúdió honlapján keresztül:**  
[http://joskiado.hu/termek/windows\\_phone\\_fejlesztes\\_lepesrol\\_lepesre.html](http://joskiado.hu/termek/windows_phone_fejlesztes_lepesrol_lepesre.html)

Szerkesztette: Novák István

Szakmai lektor: Novák István

Anyanyelvi lektor: Dr. Bonhardtné Hoffmann Ildikó

Borító: Varga Tamás

Kiadó: Jedlik Oktatási Stúdió Kft.

1215 Budapest, Ív u. 8-12.

Internet: <http://www.jos.hu>

E-mail: [jos@jos.hu](mailto:jos@jos.hu)

Felelős kiadó: a Jedlik Oktatási Stúdió Kft. ügyvezetője

Nyomta: LAGrade Kft.

Felelős vezető: Szutter Lénárd

ISBN: 978-615-5012-13-6

Raktári szám: JO-0338

# Tartalomjegyzék

<b>Előszó .....</b>	<b>13</b>
<b>1. Bevezetés a „Windows Phone” platform-ba.....</b>	<b>15</b>
<b>A Windows Phone Platform .....</b>	<b>15</b>
Windows Phone hardver követelmények.....	16
A Windows Phone platform képességei .....	16
<b>Alkalmazás fejlesztés Silverlight alapokon.....</b>	<b>19</b>
Ismerkedés a XAML-lel.....	19
XAML alapú felhasználói felületek.....	21
<b>Hello Windows Phone.....</b>	<b>22</b>
<b>Összefoglalás.....</b>	<b>26</b>
<b>2. Felhasználói élmény tervezése Windows Phone-on .....</b>	<b>27</b>
<b>A Windows Phone jelenség .....</b>	<b>27</b>
<b>Ikonografikus és infografikus felületek.....</b>	<b>28</b>
<b>A METRO Dizájn Nyelv .....</b>	<b>29</b>
Ismerős megjelenés.....	29
Tiszta és pehelykönnyű.....	29
Digitális eredetiség.....	30
Gyönyörű tipográfia.....	30
Állandóan mozgásban.....	30
A tartalom az elsődleges.....	30
<b>Design vezérelt fejlesztés.....</b>	<b>31</b>
Tartalom és információs architektúra .....	32
Használhatóság és ergonómia.....	32
Esztétika és vizuális élmény.....	32
<b>Csináld magad! .....</b>	<b>33</b>
A dizájn szórakoztató!.....	33
Innováció tervezés közben.....	33
A probléma megértése.....	33
Magasabb termékminőség .....	33
Tervezés Sketchflow-val .....	34
<b>Élmény vagy alkalmazás .....</b>	<b>35</b>
Ismerjük meg a felhasználóinkat! .....	35
Releváns tartalom .....	36
Személyes alkalmazások.....	36
Hasznos, jól használható, kíváncsú .....	36
<b>Gyakorlati Dizájn.....</b>	<b>36</b>

Mozgás és animáció .....	36
Windows Phone 7 specifikus komponensek tervezése .....	37
Vezérlők használata .....	39
Érintőképernyős alkalmazások tervezése .....	41
<b>Összefoglalás .....</b>	<b>42</b>
<b>3. Alkalmazásfejlesztés Windows Phone-on .....</b>	<b>43</b>
<b>Alapvető WP7 vezérlők .....</b>	<b>44</b>
Parancsvezérlők .....	49
Szöveg megjelenítési vezérlők .....	55
Szövegbeviteli vezérlők .....	58
Listavezérlők .....	62
<b>Adatkötés .....</b>	<b>65</b>
Erőforrások .....	65
Több vezérlő kötése egyazon forráshoz .....	68
Adatkötés UI-elemek között .....	70
<b>Az adatok megjelenésének testreszabása adatsablonokkal .....</b>	<b>72</b>
<b>A vezérlők kinézetének testreszabása vezérlősablonokkal .....</b>	<b>74</b>
<b>Az alkalmazás állapotainak létrehozása VisualState-ek segítségével .....</b>	<b>78</b>
Animációk hozzáadása .....	81
<b>Témák használata .....</b>	<b>83</b>
<b>Összefoglalás .....</b>	<b>84</b>
<b>4. Haladó alkalmazásfejlesztés Windows Phone-on .....</b>	<b>85</b>
<b>Térképek kezelése .....</b>	<b>85</b>
Feliratkozás a szolgáltatásra .....	85
A Bing Maps vezérlő használata .....	86
Pushpinek .....	89
Rétegek alkalmazása .....	90
Egyéb szolgáltatások .....	93
<b>Navigáció .....</b>	<b>97</b>
Oldalak .....	97
Adatátvitel az oldalak között .....	99
Navigációs metódusok .....	99
BackStack .....	100
Az alkalmazás kiegészítése navigációs vezérlőkkel .....	101
<b>Pivot és Panorama .....</b>	<b>103</b>
Különbségek .....	104
<b>Silverlight Toolkit for Windows Phone7 .....</b>	<b>105</b>
Telepítés .....	105
Fontosabb vezérlők .....	106
Példaprogram .....	109
<b>Összefoglalás .....</b>	<b>111</b>
<b>5. Az alkalmazás életciklusa .....</b>	<b>113</b>

<b>Multitasking</b> .....	<b>113</b>
A feladatütemezés elméletben .....	114
A multitasking megvalósítása WP 7.5 platformon.....	115
<b>Állapotok kezelése</b> .....	<b>117</b>
Életciklus-események kezelése .....	117
Az alkalmazás állapotának mentése és helyreállítása.....	119
<b>Folyamatok a háttérben</b> .....	<b>125</b>
Elméleti áttekintés.....	125
Zenelejátszás integrálása saját alkalmazásokba.....	126
Figyelmeztetések és riasztások kezelése.....	132
Adatok letöltése a háttérben.....	133
<b>Összefoglalás</b> .....	<b>135</b>
<b>6. Alapvető telefonos funkciók használata</b> .....	<b>137</b>
<b>A példaalkalmazás létrehozása</b> .....	<b>138</b>
<b>Launcherek</b> .....	<b>142</b>
PhoneCallTask.....	142
SmsComposeTask.....	143
EmailComposeTask.....	144
WebBrowserTask.....	144
MediaPlayerLauncher .....	145
SearchTask.....	146
<b>Chooserek</b> .....	<b>147</b>
SavePhoneNumberTask .....	147
PhoneNumberChooserTask.....	148
CameraCaptureTask.....	149
PhotoChooserTask .....	149
<b>Adatok megosztása alkalmazások között</b> .....	<b>150</b>
Contacts.....	151
Appointments .....	152
<b>Kamera használata taszk nélkül</b> .....	<b>153</b>
<b>Összefoglalás</b> .....	<b>161</b>
<b>7. További telefonos funkciók használata</b> .....	<b>163</b>
<b>Az érintőképernyő kezelése</b> .....	<b>163</b>
Alapvető érintési események .....	163
Multitouch manipulációk.....	166
A nyers érintési adatok kezelése.....	169
<b>Helymeghatározás</b> .....	<b>173</b>
<b>Helyzetmeghatározás</b> .....	<b>176</b>
A gyorsulásmérő használata .....	176
A giroszkóp használata .....	179
Az iránytű használata.....	180
Az érzékelők együttes használata a Motion API segítségével .....	183

<b>Összefoglalás .....</b>	<b>185</b>
<b>8. Adatkezelés .....</b>	<b>187</b>
<b>Isolated Storage .....</b>	<b>187</b>
<b>Application Settings .....</b>	<b>188</b>
<b>IsolatedStorageFileStream .....</b>	<b>194</b>
Isolated Storage gyakorlat .....	196
Önálló feladatok .....	200
<b>IsolatedStorage – Tool .....</b>	<b>200</b>
<b>Lokális adatbázisok használata Windows Phone 7-en .....</b>	<b>202</b>
Linq To SQL .....	202
DataContext .....	203
Mapping .....	203
Adatbázis létrehozása .....	204
Adatbázis létrehozása II. ....	206
Adatok felvitele – INSERT .....	210
Adatok lekérdezése – SELECT .....	210
Adatok módosítása – UPDATE .....	211
Adatok törlése – DELETE .....	211
Az adatbázis biztonsága .....	212
Adatbázis kezelés gyakorlat .....	212
<b>Összefoglalás .....</b>	<b>215</b>
<b>9. Kommunikáció szerverrel.....</b>	<b>217</b>
<b>A kommunikáció módjai .....</b>	<b>217</b>
<b>Webszolgáltatások használata .....</b>	<b>218</b>
A webszolgáltatások működése.....	218
Webszolgáltatás egy mintaalkalmazásban .....	219
<b>Webes tartalom letöltése .....</b>	<b>224</b>
<b>Adatelérés az OData protokollon keresztül .....</b>	<b>226</b>
<b>Adattárolás Windows Azure-ban.....</b>	<b>230</b>
Fájlok (blob-ok).....	231
Várakozási sorok (queue-k).....	234
Táblák (table-k) .....	236
Árazás .....	239
<b>Felhasználó-hitelesítés szerveroldalról .....</b>	<b>240</b>
<b>Összefoglalás .....</b>	<b>243</b>
<b>10. Lapkák és értesítések .....</b>	<b>245</b>
<b>Néhány példa a lapkák és értesítések használatára .....</b>	<b>246</b>
<b>A lapkák tulajdonságai.....</b>	<b>246</b>
Méret .....	246
Kiszögezés és elrendezés .....	247
Statikus és dinamikus lapkák.....	247
A lapkák felépítése .....	247

Másodlagos lapkák .....	248
Deep Linking .....	249
<b>Lapkák létrehozása és frissítése alkalmazásunkból .....</b>	<b>249</b>
A ShellTile API .....	249
Lapka frissítése .....	249
Másodlagos lapka létrehozása .....	253
Másodlagos lapka törlése .....	254
<b>Lapkák frissítése Background Agent-ek segítségével .....</b>	<b>254</b>
<b>Lapkák frissítése ShellTileSchedule segítségével .....</b>	<b>258</b>
Időzítés beállítása az elsődleges lapkára .....	258
Időzítés beállítása egy másodlagos lapkára .....	260
Időzítés törlése .....	260
<b>A felugró értesítések .....</b>	<b>260</b>
<b>A Push Notification szolgáltatás .....</b>	<b>261</b>
A Push Notification szolgáltatás működése .....	262
Lapkák frissítése Push Notification-ök segítségével .....	263
Értesítések megjelenítése Push Notification-ök segítségével .....	267
Raw üzenetek fogadása a Push Notification szolgáltatáson keresztül .....	268
Push Notification-önök és a Windows Azure .....	269
<b>Összefoglalás .....</b>	<b>270</b>
<b>11. Játékok a Mango világában .....</b>	<b>271</b>
<b>Windows Phone – a motorháztető alatt .....</b>	<b>271</b>
<b>Grafikus programozási és XNA alapok .....</b>	<b>272</b>
A kép kialakítása .....	272
A játékciklus .....	277
Egy XNA program felépítése .....	277
Sprite-ok .....	281
Bemenetek kezelése .....	283
Ütközések detektálása .....	285
Animációk .....	286
<b>XNA és Silverlight integráció .....</b>	<b>291</b>
Silverlight és XNA együtt a gyakorlatban .....	291
Összefoglalás .....	299
<b>12. Marketplace .....</b>	<b>301</b>
<b>APPHUB .....</b>	<b>301</b>
<b>Készüléken történő hibakeresés és regisztráció .....</b>	<b>301</b>
Alkalmazások telepítése .....	303
ChevronWP7 .....	304
<b>Publikálás .....</b>	<b>305</b>
Screenshot készítés .....	310
Windows Phone Marketplace Test Kit .....	311
Ikonok .....	313

<b>Marketplace Task .....</b>	<b>314</b>
<b>Trial Mode .....</b>	<b>316</b>
<b>Összefoglalás .....</b>	<b>317</b>
<b>13. Teljesítmény .....</b>	<b>319</b>
<b>A tipikus teljesítmény-problémákról .....</b>	<b>319</b>
<b>A CPU és a GPU feladata .....</b>	<b>320</b>
<b>Csillagok, csillagok.....</b>	<b>320</b>
<b>Lassú betöltődés kezelése .....</b>	<b>323</b>
Az elrendezés (layout) költségeinek csökkentése.....	323
Splash Screen .....	324
Inicializációs teendők elhalasztása .....	324
Háttérképek alkalmazása.....	324
Előre elvégzett munka .....	325
Kis összefoglalás.....	326
<b>Akadozó animációk okai, megoldása.....</b>	<b>326</b>
Frame Rate Counters .....	328
Redraw Regions .....	329
Használjuk a GPU-t! .....	330
Cache vizualizáció.....	331
Beépített animációk és a GPU.....	332
UI, Compositor és Input szálak.....	332
<b>Hosszú válaszidő kezelése .....</b>	<b>333</b>
<b>Általános tippek.....</b>	<b>334</b>
<b>Listák.....</b>	<b>334</b>
<b>Szubjektív teljesítmény .....</b>	<b>335</b>
Tilt effektus .....	336
A folyamat jelzése .....	337
Köztes animációk .....	337
Amikor a lassabb gyorsabb(-nak tűnik).....	337
<b>Memória-optimalizálás .....</b>	<b>338</b>
A memória foglaltság kijelzése .....	339
Optimalizálás.....	339
<b>Windows Phone Performance Analysis.....</b>	<b>339</b>
<b>Összefoglalás .....</b>	<b>343</b>





# Előszó

Amikor első hordozható telefonomat 18 évvel ezelőtt megkaptam, az még egy kisebb diplomatafarska méretével egyezett meg. Egyetlen dolgot lehetett vele csinálni: telefonálni. Még csak telefonkönyve sem volt, nekem kellett előkeresni a számokat a noteszből és bepötyögni azokat egy-egy telefonhívás során.

Szinte hihetetlen mértékű változáson estek azóta át a mobil készülékek! Ma leginkább az „okostelefon” megjelölést használjuk azoknak a tényérben elérő – szinte kizárólag érintőképernyős – mini csodáknak a megnevezésére, amelyek számítógépek, és a telefonálás lehetősége csupán egyike tucatnyi képességeiknek. Olyan eszközök, amelyeken internetet is elérő alkalmazásokat tudunk futtatni.

Az autók szerelmesei között vannak olyanok, akik szeretik gépjárművüket maguk bütykölni. A repülőgépmodellek megszállottjai között vannak olyanok, akik maguk is gépeket építenek. Ugyanígy az okostelefonok felhasználói között is többen akadnak, akik saját alkalmazásokat szeretnének készíteni, és ezek piaci sikeréből akár meg is élni.

Ez a könyv az alkalmazásfejlesztőknek készült, több korábbi kezdeményezéshez hasonlóan a magyarországi Windows Phone szakmai közösség aktív munkájával jött létre. Olyan szakemberek írták, akik korábban már megmutatták, hogy értenek a platform használatához és programozásához. Ebben a könyvben azokat az ismereteiket is átadják, amelyek segítenek a Windows Phone fejlesztésbe frissen bekapcsolódó érdeklődőknek megismerni a platform legfontosabb alapelemeit, és egyúttal betekintést is adnak annak működésébe és használatába.

**Árvai Zoltán** (1. és 2. fejezetek), **Farkas Bálint** (9. és 10. fejezetek), **Fár Attila Gergő** (5. és 11. fejezetek), **Fülöp Dávid** (3. és 7. fejezetek), **Komjáthy Szabolcs** (4. és 6. fejezetek), **Turóczy Attila** (8. és 12. fejezetek), valamint **Velvárt András** (13. fejezet) mindannyian rengeteg energiát fordítottak arra, hogy ismereteiket megosszák a fejlesztőközösséggel. Olyan magyar nyelvű könyvet hoztak létre, amelyben személyes tapasztalataikon, a platformhoz kapcsolódó élményeiken keresztül mutatják be a Windows Phone alkalmazásfejlesztés alapvető eszközeit és fogásait.

A könyv és a hozzátartozó programozási példák a weben is elérhetők a **Devportal.hu** oldalon.

Novák István

Budapest, 2011. december



# 1. Bevezetés a „Windows Phone” platform-ba

Az utóbbi évek egyik legjelentősebb technológiai robbanásának lehettünk tanúi az okostelefonok, illetve az okoskészülékek piacán. Évente okostelefonok milliói találhatnak gazdára! Ezek az eszközök napról napra egyre inkább hétköznapijaink részévé válnak. Sokszor társként, életünk, digitális identitásunk egyfajta kiterjesztéseként gondolunk rájuk. A világot helyezik kezünkbe, bármi, bármikor elérhetővé válik. A telefonok tudják, hol vagyunk, kik vagyunk, kik a barátaink, mit szeretünk csinálni és kivel, mikor hol kellene lennünk, és maximálisan igyekeznek minket kiszolgálni. Nem csoda hát, hogy a Twitter, a Facebook, a Flickr és még számtalan online és folyamatos jelenlétet igénylő szolgáltatás világában ezek az eszközök robbanásszerű fejlődésen mentek át és hatalmas kereslet mutatkozik irántuk, amely évről évre csak nő.

A Microsoft egyike volt az elsőnek az okostelefonok piacán. A Windows Mobile termékcsalád hosszú utat járt be – még ha ez az út igencsak rögös is volt. Ez a mobil operációs rendszer azonban elsősorban üzleti használatra készült, azzal a gondolattal, hogy az asztali Windows-t vihetjük a zsebünkben. Az új igényeknek, a fogyasztói piacnak azonban ezek az elvek és ez a termék nem tudott megfelelni, a konkurens termékekkel nem volt képes felvenni a versenyt. A Microsoft - ezt a tényt felismerve - szemmel láthatólag nem erőltette tovább ezt a vonalat, és alapjaiban új operációs rendszerrel, a Windows Phone-nal jelent meg a piacon. Ezt a terméket a megváltozott, új igényekhez igazítva tervezték, gyakorlatilag a nulláról indulva.

Az okostelefonok új piaca nemcsak a felhasználók számára jelentenek izgalmas, új lehetőségeket, hanem a fejlesztőcégek számára is. Az elmúlt években számos mesébe illő sikertörténetet hallhattunk olyan cégekről, akik a semmiből, 1-2 év alatt meghatározó szereplőkké nőttek ki magukat. Így nem csoda, hogy az okostelefonok megjelenésével egyfajta fejlesztői „aranyláz” tört ki. Sajnos az idő előrehaladtával egyre nehezebb sikertörténeteket írni, de a Windows Phone újabb ilyen lehetőséget jelent! A Microsoft, a Windows Phone és saját alkalmazásboltja (Marketplace) megjelenésével új piacot nyit a vállalkozó szellemű fejlesztők számára, és újabb lehetőségeket kínál, hogy újabb sikertörténetek születhessenek.

## A Windows Phone Platform

A Windows Phone megjelenésében merőben újszerű, az eddigi mobil operációs rendszerektől eltérő dizájnnal és szellemiséggel rendelkezik. A Windows Phone különlegessége, hogy a Microsoft készüléket nem, csupán mobil operációs rendszert fejleszt, ezzel lehetőséget biztosítva a készülék gyártói számára saját termékpaletták kialakítására.

A gyakorlott fejlesztők ezeket a sorokat olvasva biztosan felhördülnek és elgondolkodnak, hogy ez vajon mekkora fragmentációt jelenthet a készülékek piacán, az alkalmazást hány különböző eszközre és hardver konfigurációra kell elkészíteni. Az Apple a fragmentáció kiküszöbölésére rendkívül egyszerű stratégiát alkalmaz: a készülékeket ő maga gyártja. A Google ezzel a kérdéssel nem nagyon foglalkozik, illetve a könyv írásának pillanatáig nem sokat foglalkozott. A Google platformján (Android) a fejlesztők számára komoly kihívást jelent az alkalmazások felkészítése a fragmentációra, sokszor kényszerülnek olyan döntéseket hozni, hogy bizonyos kategória alatt nem támogatnak telefonokat. A Microsoft valahol a kettő közötti utat választotta. Habár saját eszközt nem dobott a piacra, de nagyon pontos specifikációt írt elő a telefon hardveres képességeit illetően. Az a telefon lehet Windows Phone, amely az előírt hardver követelményeket maradéktalanul kielégíti.

### Windows Phone hardver követelmények

A Windows Phone készülékeknek számos hardveres előírásnak kell megfelelniük:

- **Fizikai gombok:** A Windows Phone készülékeken kötelezően jelentkező hardveres gombok, amelyek szorosan kapcsolódnak az operációs rendszer és a navigációs mechanizmus működéséhez.
  - *Vissza gomb (Back):* Az alkalmazáson belüli navigációt, illetve futó taszkok közötti váltást szolgálja.
  - *Kezdő oldal gomb (Home):* A Start képenyőre navigál bárhonnan, bármely alkalmazásból.
  - *Kereső gomb:* Minden esetben a beépített Bing keresőt hozza fel. 7.1-ben ez a viselkedés nem módosítható.
- **Kijelző:**
  - A kijelzőnek legalább *800 x 480-as felbontással* kell rendelkeznie.
  - A kijelzőnek támogatnia kell a *kapacitív technológiát* és minimum 4 pontos érintést meg kell tudnia különböztetni.
- **Hálózati jellemzők:**
  - A készüléknek hálózati kommunikáció tekintetében *mobil adathálózatok kezelését*, valamint *Wi-Fi-t* kell támogatnia.
- **Memória:**
  - Operatív memória tekintetében legalább *256 MB memóriával* kell rendelkeznie a készüléknek. (A gyártók döntő többsége 512MB-ot tesz a készülékeibe.)
  - Adatok tárolásra *minimum 8GB-os flash tárolóegységgel* kell rendelkeznie a készüléknek.
- **Navigáció:**
  - Műholdas navigáció tekintetében Assisted-GPS támogatás szükséges.
- **Mozgásérzékelés:** A telefon mozgásának, döntésének és egyéb pozícióinak követésére *gyorsulásmérőt* kell beépíteni.

A Windows Phone által támogatott egyéb opcionális hardverek listája:

- Iránytű
- Giroszkóp (jóval pontosabb pozíció/állapot meghatározásra képes, mint a gyorsulásmérő)
- Hátsó kamera
- Előlapi kamera

### A Windows Phone platform képességei

A Windows Phone platform alapvetően négy részből tevődik össze:

1. Futtatókörnyezet
2. Integráció a felhővel
3. Marketplace portál szolgáltatások
4. Fejlesztőeszközök

#### Futtatókörnyezet

A Windows Phone alkalmazások izoláltan, ún. sandboxban („homokozó”) futnak, elsősorban biztonsági okok miatt. Az alkalmazások ennek köszönhetően nem férhetnek hozzá kontrollálatlanul tetszőleges erőforráshoz, ezt a futtatókörnyezet szigorúan ellenőrzi.

Saját termékek fejlesztése során két platform közül választhatunk: Silverlight, illetve XNA.

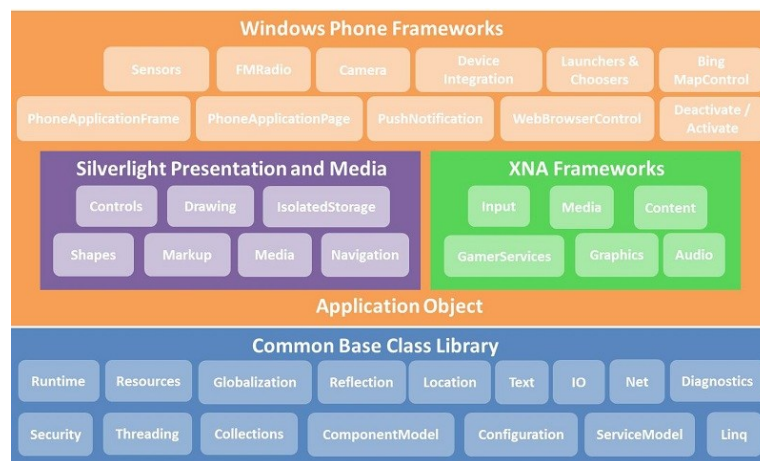
*Silverlight* felhasználásával látványos, médiaközpontú alkalmazásokat fejleszthetünk, melyek komponensei jól integrálódnak a Windows Phone Metro stílusú látványvilágával. A *Silverlight*nek köszönhetően csupán a kreativitásunk szabhat határt a felhasználói élménynek, felhasználói felületeknek.

*XNA* segítségével menedzselt kódú kétdimenziós *sprite* alapú, illetve komoly háromdimenziós modellekre épülő játékokat is írhatunk. *XNA* felhasználásával fejleszthetünk Xbox-ra, Windows-ra és Windows Phone-ra is játékokat, csupán kisebb módosításokat kell végrehajtanunk. Az *XNA* API számos játékfejlesztésben felmerülő gyakori problémára (textúra és modell betöltés, fény és árnyékhatások, transzformációk, stb.) kínál kompakt megoldást, így a fejlesztők valóban a játék logikájára koncentrálhatnak. *XNA*-val a fejlesztés gyors és produktív.

A 7.1-es Mango operációs rendszer különlegessége, hogy a két technológia között az átjárhatóság biztosított. Ágyszhatunk *XNA* tartalmat *Silverlight*os felületekbe és viszont.

Windows Phone-nal ismerkedő fejlesztők számára a fejlesztési élmény olyan, mintha a .NET keretrendszeren dolgoznának. Ennek oka az ún. Base Class Library (BCL), amely a .NET platformon a jól ismert és leggyakrabban használt alapkönyvtárak jelentős részét tartalmazza, mint például szálkezelés, gyűjtemények vagy éppen a Linq. Ennek köszönhetően a már korábban elsajátított értékes tudás hordozható a két platform között.

Míg a BCL, a *Silverlight*, illetve az *XNA* ismerős koncepciókat hozott át az asztali Windows-os PC-k világából, addig számos új elem is megjelent, hiszen egy új platformról, egy mobil eszközről beszélünk. Ennek megfelelően menedzselt API-k biztosítják a telefon különböző szenzoraihoz, operációs rendszerszintű képességeihez (kapcsolatok, email küldés, életciklus menedzsment) a hozzáférést. Az 1-1 ábrán a futtatókörnyezet felépítését láthatjuk.



1-1 ábra: A futtatókörnyezet felépítése

## Integráció a felhővel

A Windows Phone alapvetően egy kliens oldali eszköz, és mint ilyen, limitált erőforrással, számítási kapacitással és tudáshalmazzal rendelkezik. A felhőben azonban folyamatosan futó szolgáltatások érhetők el, hogy azok a telefon tudását és képességeit kiterjeszthessék.

- **Értesítések:** A mai mobil eszközök kapacitása, az akkumulátor üzemideje és egyéb erőforrásai nem engedik meg, hogy párhuzamosan alkalmazások tucatjai kommunikáljanak különböző szolgáltatásokkal és egyéb komponensekkel. A Windows Phone-ok támogatják az ún. értesítéseket (*push notifications*). Ezek a felhőben futó szolgáltatások üzeneteket küldenek a telefonok felé, ha valamilyen új információ áll rendelkezésre az alkalmazás és a felhasználó számára. Ezek az értesítések megjelenhetnek az alkalmazás saját lapkáján, vagy éppen külön értesítésként is.
- **Térképszolgáltatások:** A felhőben pozicionálási és térképszolgáltatások is elérhetők. Ezek Wi-Fi, cellainformációk, illetve GPS adatok segítségével biztosítják a pontos helymeghatározást.

- **Közösségi hálózatok:** A Windows Phone nagyon fontos részét jelentik a közösségi hálózatok és az integrált élmény biztosítása az ún. *hub*okon keresztül. Soha ne felejtjük el, hogy a telefon egy kliens, amely szolgáltatások ezreihez képes csatlakozni, és segítségükkel rendkívül gazdag és értékes információhalmazt biztosítani.
- **Xbox Live szolgáltatások:** A Windows Phone képes a felhőben futó Xbox Live szolgáltatásokhoz is kapcsolódni, így nyomon követhetjük barátainkat, összehasonlíthatjuk elért eredményeiket a miénkkel, üzeneteket küldhetünk nekik, vagy akár otthoni Xboxunkat is vezérelhetjük telefonunkról.

### ***A Marketplace portálszolgáltatásai***

A Windows Phone Marketplace egy központi portál, ahova a fejlesztők elkészült alkalmazásait tölthetik fel, és amelyen keresztül értékesíthetik azokat. A felhasználók számára ez hatalmas piacter, ahol a különböző tartalmak és alkalmazások közül válogathatnak.

#### **Regisztráció fejlesztők számára**

A telefonra történő fejlesztést megelőzi a fejlesztői programba történő jelentkezés, amely egy Live ID segítségével az ún. App Hub-on, a <http://create.msnd.com> címen található portálon végezhető el. A fejlesztőeszközök is erről a portálról tölthetők le közvetlenül. A regisztrációt és a hitelesítést követően a fejlesztői készülékek zárolását feloldhatjuk, és ennek eredményeképpen saját alkalmazásokat tudunk rá telepíteni a Visual Studio 2010 segítségével. Az éves fejlesztői díj 99 dollár. Cserébe a publikációs lehetőségen túl számos szolgáltatást kapunk a portálon, ahol a fejlesztői vezérlőpulton értékes információkhoz, statisztikákhoz juthatunk alkalmazásainkat illetően. A Marketplace-en keresztül történik a számlázás és a frissítések kezelése is.

#### **Alkalmazások közzététele**

Az alkalmazások, feltöltésüket követően, egy hitelesítési folyamaton mennek keresztül, amely során a Microsoft egy szigorú tesztorozat futtatásának segítségével meggyőződik arról, hogy az alkalmazásunk stabil, nem szolgál rossz célt, továbbá kielégíti az alkalmazások által követendő szabályokat. Ezek a szabályok az *Application Certification Requirements for Windows Phone* című dokumentumban kerültek rögzítésre. A dokumentum a következő címen érhető el: [http://msdn.microsoft.com/en-us/library/hh184843\(v=VS.92\).aspx](http://msdn.microsoft.com/en-us/library/hh184843(v=VS.92).aspx).

Miután az alkalmazásunk sikeresen átment a teszteken, egy tanúsítványt kap, ennek segítségével feltölthetjük a Windows Phone alkalmazás boltba, ahol az ár és a régiók kiválasztása után az alkalmazás széles közönség számára válik elérhetővé.

### ***Fejlesztőeszközök***

A Windows Phone-ra fejleszteni készülők számára fontos kérdés az indulás költségvonzata. Szerencsére elmondhatjuk, hogy a fejlesztési infrastruktúra költsége gyakorlatilag nulla. Természetesen egy Windows-t futtató PC-re szükség lesz, továbbá egy Windows Phone készülék sem árt, bár ez utóbbi nem kötelező, csupán erősen ajánlott. A fejlesztőeszközök a Windows Phone fejlesztők számára ingyenesen elérhetők.

Az induláshoz csupán a Windows Phone SDK 7.1-es változatot kell letölteni, az alábbi címről:

<http://www.microsoft.com/download/en/details.aspx?displaylang=en&id=27570>

Az SDK a következő fontosabb komponenseket tartalmazza:

- **Visual Studio 2010 Express for Windows Phone:** Kizárólag Windows Phone projektek kezelésére alkalmas, ingyenes Visual Studio 2010 változat, amellyel üzleti célra is fejleszthetünk alkalmazásokat.
- **Windows Phone Emulator:** Fejlesztési időben használható Windows-on futó emulátor, amely a telefon legtöbb képességét képes emulálni.
- **Microsoft Expression Blend for Windows Phone:** A Windows Phone felhasználói felületek fejlesztésével és építésével foglalkozó dizájnerek és fejlesztők első számú eszköze. A Microsoft Expression Blend 4.0 termék kizárólag Windows Phone projektek kezelésére alkalmas, de teljes értékű változata.

Amennyiben korábban vásárolt teljes értékű Visual Studio 2010 változattal rendelkezünk (Professional, Premium, Ultimate) a fejlesztőeszközök beépülnek a telepített változatba.

## Alkalmazás fejlesztés Silverlight alapokon

Nem is olyan régen a Windows platformra készülő alkalmazások jelentős részét a .NET keretrendszer egyik kulcstechnológiájának, a Windows Forms-nak a segítségével fejlesztették ki. A technológia hatékony, kiforrott és rendkívül produktív volt. Sajánlatos módon további sorsát megpecsételte saját képességeinek erős korlátozottsága. Az alkalmazás- és a vezérlőmodell, amely a Windows Forms-ot jellemezte rendkívül rugalmatlan és merev volt. Ennek következtében nem tudott megfelelni az új igényeknek, amelyeknek középpontjában a modern, izgalmas és jól használható felhasználói felületek helyezkedtek el.

Annak érdekében, hogy érezzük ezt a kötöttséget, képzeljük el, hogy olyan alkalmazást fejlesztünk, ahol különböző folyamatokat és annak státuszait kell megjeleníteni. A folyamatokat egy táblázat sorai reprezentálják, a státuszt pedig minden sorban jelölje egy százalékos érték, melyet reprezenáljunk egy **ProgressBar**-al. A megvalósítás során hamar belefutunk abba a rendkívül kellemetlen ténybe, hogy a Windows Forms **ListView** vezérlője nem támogatja a **ProgressBart**, mint egy cella lehetséges elemét. Az ilyen típusú kötöttségek nagyon szűk korlátok közé szorították a termék készítőinek kreativitását, játékterét.

A Windows Forms maradiságának okán a technológia leváltására a Microsoft elindította az Avalon projektet, melyet nem sokkal később Windows Presentation Foundation (WPF) néven ismert meg a világ a .NET 3.0 keretrendszer megjelenésekor. A WPF számos új koncepció mellett egy nagyon fontos újdonságot hordozott magában – az új vezérlőmodellt. A gondolat nagyon egyszerű volt, a vezérlők alapvetően tartalom jellegű vezérlők (**ContentControl**) és bármit tartalmazhatnak. Ennek köszönhetően egy **Button** vezérlőbe egy másik **Button**ot, majd abba egy **DataGrid**et helyezhetünk el, aminek természetesen túl sok értelme ugyan nincs, de nagyon jól példázza a modell rugalmasságát.

### Ismerkedés a XAML-lel

Az új modell mellé egy másik fontos koncepció is párosult, ez pedig a felhasználói felület deklaratív leírása. A felhasználói felületek leírásának egyik és egyben talán legtermészetesebb módja a leíró nyelvek (Markup Language) használata. Gondoljunk csak a HTML-re, melynek segítségével egészen különleges webes felületek építhetők! A Microsoft az új felhasználói felület leírására az ún. XAML nyelvet választotta. A XAML (eXtensible Application Markup Language) egy kiterjeszthető alkalmazásleíró nyelv, így nem felhasználói felület specifikus. Az XML egyfajta kiterjesztése egy különleges XAML Parser-rel, melynek feladata a XAML értelmezése és a leírt objektumgráf felépítése. Ez lett a WPF legjelentősebb újdonsága. A WPF megjelenését követően a Microsoft hamarosan úgy ítélte, hogy a XAML alapú felületek jelentik a jövőt, és a webre is célszerű lenne elkészíteni egy WPF-hez hasonló technológiát. Ezt a technológiát később Silverlightként ismerte meg a világ. A sors különleges fintora, hogy az évek elteltével a Silverlight egyre inkább próbált megszabadulni a böngészőtől és önálló életre kelni. A Windows Phone fő alkalmazásfejlesztési platformja a 7.0 változatban és a 7.1-ben is a Silverlight. A XAML alapú rendszerek komoly sikertörténetet tudnak magukénak. Meghódították az asztali PC-k világát, a webes és intranetes környezetekben is komoly szerepet kaptak. Legújabbban pedig különböző mobil, illetve hordozható eszközökön fejleszthetünk alkalmazásokat XAML alapokon. A technológia és a koncepció igazi bebetonozását a Windows 8 Metro stílusú alkalmazások XAML alapú fejlesztésének lehetősége jelenti.

### XAML szintaktika – értékadás

A XAML alapvetően az XML szintaktikai előírásait követi, illetve terjeszti ki, mint alkalmazás leíró nyelv. Ha XAML-ben leírunk egy elemet (pl **<Button/>**), akkor abból futásidőben egy **Button** példány fog elkészülni. A **Button** egyéb tulajdonságait, mint például a szélességét vagy magasságát, esetleg a tartalmát, attribútum, illetve tulajdonság szintaxissal lehet meghatározni. Míg az előbbi egyszerű értékek beállítása esetén elegendőnek bizonyulhat, komplexebb értékadásoknál az utóbbi mechanizmust alkalmazzuk.

## 1. Bevezetés a „Windows Phone” platform-ba

Tartalom meghatározása attribútum szintaxissal:

```
<Button Content="Hello World"/>
```

A fenti példában a gomb tartalma egy egyszerű string, ezért elegendő attribútum szintaxist alkalmaznunk. Tartalom meghatározása tulajdonság szintaxissal:

```
<Button>
  <Button.Content>
    <StackPanel>
      <Ellipse Fill="Red" Width="10" Height="10"/>
      <TextBlock Text="Hello World"/>
    </StackPanel>
  </Button.Content>
</Button>
```

A fenti példában a **Button** belső tartalmát egy némileg összetettebb vezérlőfa határozza meg. A **Button** belsejébe egy panel kerül, amely egymás alá rendezi el az elemeket. A belső panelbe pedig egy piros 10 x 10-es kör, valamint a „Hello World” felirat kerül. Ezt a belső tartalmat attribútum szintaxissal nem lehet meghatározni, ezért a Content tulajdonságot „ki kell nyitni”, azaz tulajdonság szintaxissal kell beállítani a tartalmat. Figyeljük meg, hogy a kör (**Ellipse**) és a felirat (**TextBlock**) saját tulajdonságait attribútum szintaxissal határoztuk meg.

### XAML szintaktika – Prefixek használata

A C#-ban vagy éppen a Visual Basic-ben megszokhattuk, hogy amikor leírjuk az objektumok nevét, akkor a megfelelő névtérre kell hivatkoznunk. Ha a Windows Phone-ra egy saját Button vezérlőt fejlesztünk, akkor azt szeretnénk helyezni a felhasználói felületen használni. De honnan tudhatja a XAML Parser, hogy mi most melyik gombra gondolunk? A beépített **Button** érdekel minket, vagy a saját **Button** szeretnénk használni? Erre a problémára a prefixek használata jelenti a megoldást. A megfelelő XAML elemeken definiálnunk kell a névtereket egy hivatkozással, az ún. **Prefix**szel együtt. Ezt követően a prefix felhasználásával könnyedén hivatkozhatjuk meg a saját komponenseinket.

Prefix definiálása saját komponens használatához:

```
xmlns:myControls="clr-namespace:MyControlsLibrary;assembly=MyControlsLibrary"
```

A fenti példában a **MyControlsLibrary.dll** állomány **MyControlsLibrary** névtérét rendeljük a **myControls** prefixhez.

Saját komponens meghivatkozása prefixen keresztül

```
<myControls:MyButton>
</myControls:MyButton>
```

A fenti példában a **myControls** prefix által reprezentált névtérben található **MyButton** komponenst hivatkozunk meg.

### XAML szintaktika – Leírónyelvi kiterjesztések

Gyakran előfordul, hogy a XAML Parser számára valami különleges dolgot kell jelezni, amelyet vagy nem tudunk az XML szintaktikai eszközeivel leírni, vagy éppenséggel túl körülményes lenne. Az ilyen esetekben (például adatkötés, hivatkozás statikus erőforrásokra) ún. Markup Extension-öket, azaz leírónyelvi kiterjesztéseket használhatunk. Ezek sajátos szintaktikája a következő:

**PropertyNév="{kiterjesztésneve [hivatkozás paraméterek]}"**



Néhány példa markup extension-ökre:

```
<Button Style="{StaticResource buttonStyle}"/>
<TextBox Text="{Binding FirstName, Mode=TwoWay}"/>
```

Ne aggódjunk amiatt, ha a fenti sorok egyelőre semmit nem jelentenek számunkra! A későbbi fejezetek során ezek jelentésére fény derül.

## XAML alapú felhasználói felületek

Egy egyszerű Windows Phone felhasználói felületet leíró XAML részlet anatómiája:

```
<phone:PhoneApplicationPage
  x:Class="PhoneApp2.SecondPage"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:phone="clr-namespace:Microsoft.Phone.Controls;assembly=Microsoft.Phone"
  xmlns:shell="clr-namespace:Microsoft.Phone.Shell;assembly=Microsoft.Phone"
  xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
  xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
  FontFamily="{StaticResource PhoneFontFamilyNormal}"
  FontSize="{StaticResource PhoneFontSizeNormal}"
  Foreground="{StaticResource PhoneForegroundBrush}"
  SupportedOrientations="Portrait" Orientation="Portrait"
  mc:Ignorable="d" d:DesignHeight="768" d:DesignWidth="480"
  shell:SystemTray.IsVisible="True">

  <Grid x:Name="LayoutRoot" Background="Transparent">

    </Grid>

</phone:PhoneApplicationPage>
```

A **PhoneApplicationPage** objektum egy Windows Phone képernyőt reprezentál. Az **x:Class** tulajdonság segítségével azt határozzuk meg, hogy melyik saját osztály származik a **PhoneApplicationPage** osztályból, azaz melyik az az osztály, amelyik a kapcsolódó mögöttes kódban megtalálható, kiegészíthető. A következő sorokban számos prefix definíciót látunk, ebből egy kiemelkedik, ahol nem szerepel prefix, csupán az **xmlns** definíció. Ez lesz az alapértelmezett névtér, azaz ha prefix nélkül írunk le xaml-ben egy elemet, akkor azt ebben a névtérben fogja keresni a XAML Parser.

A prefix definíciók után néhány **PhoneApplicationPage** tulajdonság kerül beállításra, mint a betűkészlet típusa, mérete, illetve az előtér színe. Ezek rendszerszintű beállításokra (ún. erőforrásokra) hivatkoznak.

A következő két tulajdonság azt határozza meg, hogy az alkalmazás milyen orientációkat támogat (**SupportedOrientations**) és mi az alapértelmezett orientáció (**Orientation**). A „d” prefixszel ellátott beállítások egytől egyig a tervezőfelület számára meghatározott értékek, futásidőben nem játszanak szerepet. Az **mc:Ignorable** paraméter azt határozza meg, hogy a „d” prefixet nem kell figyelembe venni futásidőben, ezt kizárólag a tervezőeszköz tartja számon.

A fenti **SystemTray.IsVisible** tulajdonság határozza meg azt, hogy a felső rendszerinformációs sáv (akkumulátor töltöttség, óra, stb.) látszik-e az alkalmazásban, vagy elrejtjük.

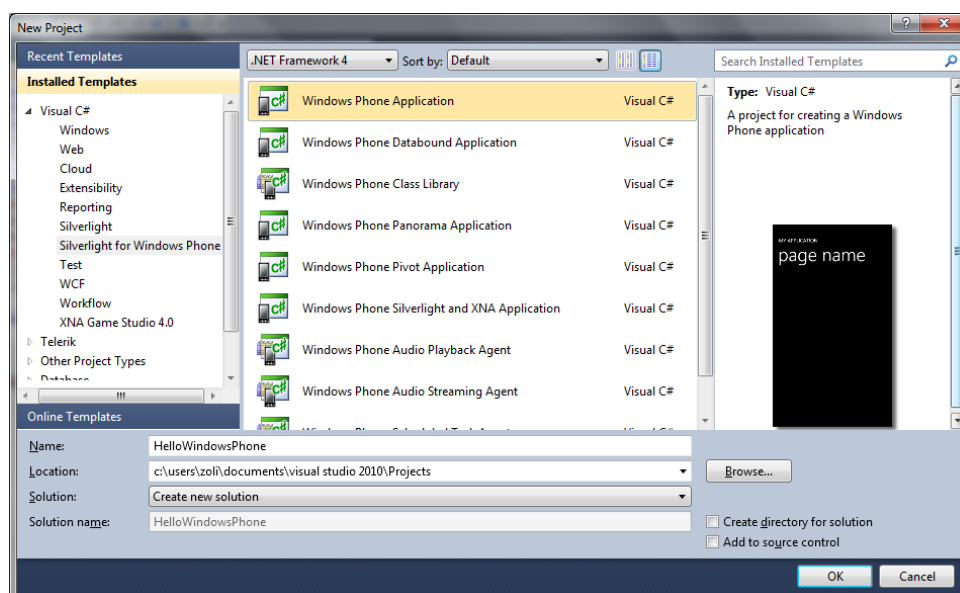
Végül a **PhoneApplicationPage** elem kerül definiálásra, már csak a benne levő tartalom – azaz az oldal tartalma – a kérdéses. Jelen esetben egyetlen **Grid** panel található benne, amelyen transzparens hátteret határoztunk meg, továbbá a panel neve „LayoutRoot” lett. De mi az az **x** prefix? Az **x:Class**-nál is feltűnt már korábban. Az **x** prefix a XAML nyelv saját névtérét reprezentálja. Azaz, ha valami XAML-specifikus dologra van szükségünk, akkor azt a XAML névtérben találhatjuk meg, amelyhez az **x** prefix került hozzárendelésre.

# Hello Windows Phone

Egyetlen valamirevaló fejlesztő sem vághat bele egy új technológiával való ismerkedésbe anélkül, hogy el ne készítse rá a dedikált Hello World alkalmazását.

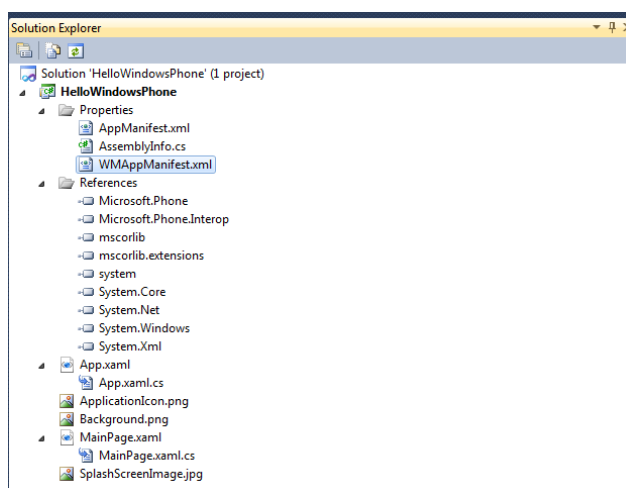
Készítsük hát el első Windows Phone alkalmazásunkat!

1. Hozzunk létre egy új Windows Phone alkalmazást! Kattintsunk a File menüre, válasszuk ki a New Project parancsot! Az elérhető alkalmazás típusok listájából válasszuk a Windows Phone Application sablont! A projektet nevezzük el HelloWorldsPhone-nak, a többi beállítás alapértelmezett értékét hagyjuk meg (1-2 ábra)! Kattintsunk az OK gombra! A dialógus ablakban láthatjuk, hogy számos különböző projektsablon áll rendelkezésünkre. Ezekkel a későbbi fejezetek során részletesebben is megismerkedünk majd. Egyelőre gondoljunk ezekre úgy, mint különböző alkalmazás típusok, illetve különböző felépítéssel rendelkező alkalmazások speciális változatai.



1-2 ábra: Új projekt létrehozása

2. A következő lépésben a Visual Studio megkérdezi tőlünk, hogy melyik operációs rendszer változatra szeretnénk fejleszteni. Győződjünk meg róla, hogy a Windows Phone OS 7.1 van kiválasztva, majd kattintsunk az OK gombra!
3. Vegyük szemügyre a Visual Studio által létrehozott elemeket! A Solutions Explorer ablak tartalma az 1-3 ábrán látható.



1-3 ábra: A projekt felépítése

A HelloWorldWindowsPhone projekt elemeinek tartalmát az alábbi táblázat mutatja be:

Elem neve	Elem jelentése
Properties	Az alkalmazás tulajdonságait és beállításait tartalmazó állományok
References	Hivatkozott egyéb könyvtárak
App.xaml (App.xaml.cs)	Az alkalmazást reprezentáló objektum és a mögöttes kódja
ApplicationIcon.png	A lapkához tartozó kép
Background.png	Az alkalmazás háttérképe
MainPage.xaml (MainPage.xaml.cs)	Az alkalmazás főoldala és a kapcsolódó mögöttes kód
SplashScreenImage.png	Induláskor megjelenő, töltési idő alatt látható kép

A Properties alatt található **WMAppManifest.xml** fájl megér külön egy kis magyarázatot:

```
<Deployment xmlns="http://schemas.microsoft.com/windowsphone/2009/deployment" AppPlatformVersion="7.1">
  <App xmlns="" ProductID="{9bb8c371-293f-4726-a4ba-5b2e2efb152d}"
    Title="HelloWindowsPhone" RuntimeType="Silverlight" Version="1.0.0.0"
    Genre="apps.normal" Author="HelloWindowsPhone author"
    Description="Sample description" Publisher="HelloWindowsPhone">
    <IconPath IsRelative="true" IsResource="false">ApplicationIcon.png</IconPath>
    <Capabilities>
      <Capability Name="ID_CAP_GAMERSERVICES"/>
      <Capability Name="ID_CAP_IDENTITY_DEVICE"/>
      <Capability Name="ID_CAP_IDENTITY_USER"/>
      <Capability Name="ID_CAP_LOCATION"/>
      <Capability Name="ID_CAP_MEDIALIB"/>
      <Capability Name="ID_CAP_MICROPHONE"/>
      <Capability Name="ID_CAP_NETWORKING"/>
      <Capability Name="ID_CAP_PHONEDIALER"/>
      <Capability Name="ID_CAP_PUSH_NOTIFICATION"/>
      <Capability Name="ID_CAP_SENSORS"/>
      <Capability Name="ID_CAP_WEBBROWSERCOMPONENT"/>
      <Capability Name="ID_CAP_ISV_CAMERA"/>
      <Capability Name="ID_CAP_CONTACTS"/>
      <Capability Name="ID_CAP_APPOINTMENTS"/>
    </Capabilities>
    <Tasks>
      <DefaultTask Name="_default" NavigationPage="MainPage.xaml"/>
    </Tasks>
    <Tokens>
      <PrimaryToken TokenID="HelloWindowsPhoneToken" TaskName="_default">
        <TemplateType5>
          <BackgroundImageURI IsRelative="true"
            IsResource="false">Background.png</BackgroundImageURI>
          <Count>0</Count>
          <Title>HelloWindowsPhone</Title>
        </TemplateType5>
      </PrimaryToken>
    </Tokens>
  </App>
</Deployment>
```

A fenti XML kódrészletben jól látható, hogy az alkalmazás alapbeállításai itt kerülnek meghatározásra: a háttérkép, az ikon, az alkalmazás felirata, stb. A legérdekesebb szekció azonban a **Capabilities** elem és gyerekei. Ezeknek a gyerekelemeknek a felvételével határozhatjuk meg, hogy az alkalmazásunk a telefon

mely képességeihez fér hozzá, mit képes kihasználni. Biztonsági szempontból érdemes ügyelni arra, hogy itt csak azok az elemek maradjanak meg, amelyeket ténylegesen is használunk.

4. Kattintsunk duplán a **MainPage.xaml** állományra, majd vegyük szemügyre annak tartalmát:

```
<phone:PhoneApplicationPage
  x:Class="HelloWindowsPhone.MainPage"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:phone="clr-namespace:Microsoft.Phone.Controls;assembly=Microsoft.Phone"
  xmlns:shell="clr-namespace:Microsoft.Phone.Shell;assembly=Microsoft.Phone"
  xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
  xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
  mc:Ignorable="d" d:DesignWidth="480" d:DesignHeight="768"
  FontFamily="{StaticResource PhoneFontFamilyNormal}"
  FontSize="{StaticResource PhoneFontSizeNormal}"
  Foreground="{StaticResource PhoneForegroundBrush}"
  SupportedOrientations="Portrait" Orientation="Portrait"
  shell:SystemTray.IsVisible="True">

  <!--LayoutRoot is the root grid where all page content is placed-->
  <Grid x:Name="LayoutRoot" Background="Transparent">
    <Grid.RowDefinitions>
      <RowDefinition Height="Auto"/>
      <RowDefinition Height="*" />
    </Grid.RowDefinitions>

    <!--TitlePanel contains the name of the application and page title-->
    <StackPanel x:Name="TitlePanel" Grid.Row="0" Margin="12,17,0,28">
      <TextBlock x:Name="ApplicationTitle" Text="MY APPLICATION"
        Style="{StaticResource PhoneTextNormalStyle}"/>
      <TextBlock x:Name="PageTitle" Text="page name"
        Margin="9,-7,0,0" Style="{StaticResource PhoneTextTitle1Style}"/>
    </StackPanel>

    <!--ContentPanel - place additional content here-->
    <Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0"></Grid>
  </Grid>

  <!--Sample code showing usage of ApplicationBar-->
  <!--<phone:PhoneApplicationPage.ApplicationBar>
    <shell:ApplicationBar IsVisible="True" IsMenuEnabled="True">
      <shell:ApplicationBarIconButton IconUri="/Images/appbar_button1.png"
        Text="Button 1"/>
      <shell:ApplicationBarIconButton IconUri="/Images/appbar_button2.png"
        Text="Button 2"/>
      <shell:ApplicationBar.MenuItems>
        <shell:ApplicationBarMenuItem Text="MenuItem 1"/>
        <shell:ApplicationBarMenuItem Text="MenuItem 2"/>
      </shell:ApplicationBar.MenuItems>
    </shell:ApplicationBar>
  </phone:PhoneApplicationPage.ApplicationBar-->

</phone:PhoneApplicationPage>
```

A **PhoneApplicationPage** beállítások már korábban tisztázásra kerültek, ezért koncentráljunk most a **LayoutRoot** elnevezésű **Grid** panel tartalmára! Ha sok benne az ismeretlen elem, ne aggódjunk, a későbbi fejezetekben azokat részletesen bemutatják! Az érdekes rész az **ApplicationTitle** és **PageTitle** névvel rendelkező **TextBlock** vezérlők **Text** tulajdonságai. A **Text** tulajdonság határozza meg a felirat tartalmát. Módosítsuk az értékeket a következő módon:

```
<StackPanel x:Name="TitlePanel" Grid.Row="0" Margin="12,17,0,28">
  <TextBlock x:Name="ApplicationTitle" Text="MY FIRST APPLICATION"
```

```

        Style="{StaticResource PhoneTextNormalStyle}"/>
        <TextBlock x:Name="PageTitle" Text="hello page"
            Margin="9,-7,0,0" Style="{StaticResource PhoneTextTitle1Style}"/>
    </StackPanel>

```

5. Keressük meg a **ContentPanel** elnevezésű **Grid** panelt! Ennek a belseje fogja reprezentálni az oldal tartalmát. Helyezzük el benne az alábbi kódrészletet:

```

<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <Button Content="Greet Me" Click="Button_Click"
        HorizontalAlignment="Center" VerticalAlignment="Center"/>
</Grid>

```

Ezzel egy középre igazított gombot helyeztünk el, melyen a „Greet Me” felirat található. Ezen felül a **Click** eseményre is feliratkoztunk egy **Button\_Click** eseménykezelő metódussal.

6. Kattintsunk az egér jobb gombjával a **Button\_Click** szövegre, és válasszuk a Navigate To Event Handler parancsot! Ennek hatására a Visual Studio megnyitja a **MainPage.xaml.cs** kódállományt, és a megfelelő eseménykezelő metódusra navigál. Egészítsük ki a kódot, az alábbi módon:

```

private void Button_Click(object sender, RoutedEventArgs e)
{
    MessageBox.Show("Windows Phone says hello!");
}

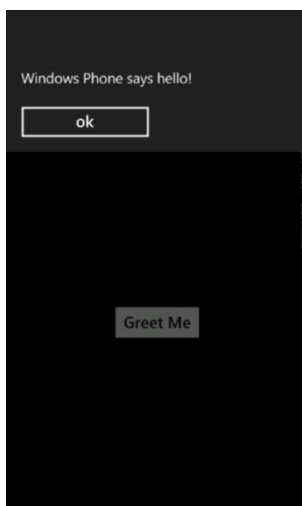
```

7. Futtassuk az alkalmazást! A Visual Studio felső sávjában, közvetlenül a menü alatt láthatjuk a Windows Phone Emulator beállítást. Ez azt jelenti, hogy a Visual Studio alapértelmezett módon az emulátoron futtatja majd az alkalmazást. Ha az emulátor még nem fut, akkor indít egy újat, telepíti rá, majd futtatja rajta az alkalmazást.

**Figyelem:** Az emulátort csak egyszer kell elindítani, ha végeztünk a futtatással, nem érdemes leállítani azt, hanem a Visual Studióban érdemes a futást megszakítani. Az emulátor elsőre lassabban indul el, de egyetlen futó példányon bármennyi tesztfuttatást indíthatunk! A lenyíló menüben a Windows Phone Device-t is kiválaszthatjuk, ha szeretnénk közvetlenül egy valódi eszközön futtatni az alkalmazást.

Az éles eszközön való tesztelés esetén a következő feltételeknek rendelkezésre kell állniuk:

- Zune Desktop alkalmazás fut a háttérben
  - A telefon össze van kötve USB adatkábelrel a számítógéppel
  - A telefonon a lockscreen fel van oldva
8. A futtatás megkezdéséhez kattintsunk a Debug menüre, majd válasszuk ki a Start Debugging menüpontot!
  9. Teszteljük az alkalmazást! Kattintsunk a Greet Me gombra! Az eredmény az 1-4 ábrán látható.



**1-4 ábra: A „Hello Windows Phone” alkalmazás futás közben**

## Összefoglalás

Ebben a fejezetben megismerkedhettünk a Windows Phone alkalmazásplatformmal. Betekintést nyerhettünk a platform legfontosabb építőelemeibe, a Silverlight és a XAML nyelv alapjaiba. Bemutatásra került a fejlesztőeszköz, valamint megírtuk életünk első Windows Phone alkalmazását, amelynek során részletesen megvizsgáltuk, hogyan is épül fel egy Windows Phone projekt, melyek a legfontosabb elemei, beállításai. A következő fejezetekben ezt a tudást mélyítjük el, hogy végül professzionális alkalmazásokat tervezhessünk és fejleszthessünk.

## 2. Felhasználói élmény tervezése Windows Phone-on

A történelem során vállalkozások ezrei keresték a siker zálogát jelentő receptet. Ilyet senki nem hozott nyilvánosságra, de számos kulcsfontosságú hozzávalót sikerült azonosítani a sikerhez vezető úton. Az évszázadok során a piacok talán legfontosabb mozgatórugójának az innováció bizonyult. Az újdonság alkotásának képessége felbecsülhetetlen értékű kincsnek számít az informatika világában is. Az elmúlt években mamutokat láthattunk visszasülyedni a közepszerűségbe, ismeretlen, vagy éppen csőd közeli cégeket mamutokká válni.

Ennek az izgalmas és érdekes hullámvasútnak az egyik legfőbb oka az innováció (vagy éppen annak hiánya volt), amely az elmúlt években egy új területre gyűrűzött be. A megszokott funkcionális innováció mellett újabb irányt vett, a felhasználók, az emberek felé. Ennek az innovációs hullámvasútnak a legmarkánsabb területe éppen a mobilpiac. Legtöbbször talán emlékszünk még a fél tégl méretű, telefonálásra éppen csak alkalmas készülékek megjelenésére, majd az ezt követő funkcionális innovációra, melynek során a telefonok egyre okosabbak lettek. Egyre komplexebb szoftverek és operációs rendszerek jelentek meg, melyeknek köszönhetően ezek a szinte már számítógépnek nevezhető készülékek egyre inkább eltávolodtak a felhasználók gondolkodásától, valós igényeitől.

Néhányan azonban sikeresen felismerték, hogy az okostelefonok szerepét és jelenségét alapjaiban újra kell gondolni és tervezni! Ez a piac más, mint a hagyományos értelemben vett IT szektor, ahol számos esetben a felhasználók rákényszerülnek a lehetséges és elérhető megoldásokra. Ezen a piacon, a fogyasztói termékek piacán a felhasználó diktál! Az ő igényei az elsődlegesek, és az a piaci szereplő lehet a nyerő, az írhatja a következő sikertörténetet, aki felismeri ezt az új helyzetet. A Windows Phone tökéletes példája ennek a történetnek. A Microsoft korábbi okostelefon rendszerét, a Windows Mobile-t hátrahagyva, alapjaiban gondolta újra, hogyan lépnek interakcióba a felhasználók telefonjaikkal, hogyan és mire használják azt. Ennek az „újrarendezésnek” eredményeképpen született meg a Windows Phone 7.

### A Windows Phone jelenség

Ebben a történetben nem a Microsoft a felfedező, sokkal inkább az Apple és a Google, akik komoly sikerre vitték saját iOS, illetve Android alapú mobil rendszereiket. A Microsoftnak ilyen értelemben egy markáns piacra kellett belépnie, és egy ilyen belépés sohasem egyszerű, különösen nem ilyen versenytársak mellett. A sikerhez azonban nem elég egy jó másolat elkészítése. A Microsoft jól tudta, hogy ezúttal az innovátor szerepét kell magára vállalnia, ezért a legfontosabb területen, a felhasználói élmény tekintetében a versenytársakétól merőben eltérő, más rendszert alkotott. Olyan készülékkel akart piacra lépni, amely jól elkülönül a többi konkurens terméktől, és egy új szellemiséget tükröz. A 2-1 ábrán látható a Windows Phone, egyéb okostelefonok társaságában. A Microsoft terméke valóban jól megkülönböztethető, egyedi megjelenéssel rendelkezik.





2-1 ábra: Windows Phone 7 a versenytársak mellett

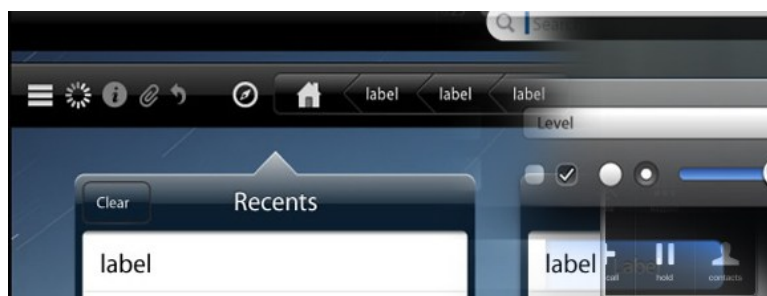
### Ikonografikus és infografikus felületek

Az egyediség kulcsa egy új koncepció, az ún. *infografikus felület*. Míg a versenytársak termékei rendre a megszokott és egyébként nagyon jól bevált ikonografikus megjelenést részesítik előnyben, addig a Microsoft új utat választott. Azokon a területeken, ahol a felhasználók kiszolgálása, a tartalom, az információ átadás, a jól használhatóság különös jelentőséggel bír, egyre nagyobb teret kapnak az infografikus felületek. A weben, a reklám és a média világában ez az irányzat már jó ideje elsőbbséget élvez. Az infografika az információközlés igazi művészete.

Ez persze nem jelenti azt, hogy azok a versenytársak, akik az ikonografikus felületet részesítik előnyben, rossz úton járnának. Ikonografikus környezetben és koncepcióval is rendkívül jól használható, izgalmas és szép felületeket készíthetünk. A Microsoft által választott út kimondja, hogy nem az alkalmazás és a vezérlők kerete vagy a „csicsa” az, ami meghatározza egy alkalmazás jellegét, hanem a benne levő tartalom. (A csicsa jelen esetben semmiképp sem pejoratív értelemben értendő.) A 2-2 ábrán egy infografikus, a 2-3 ábrán pedig egy ikonografikus felület látható. Ez az új gondolkodásmód egy igen izgalmas és markáns design nyelv létrejöttét eredményezte.



2-2 ábra: Infografikus megjelenés



2-3 ábra: Ikonografikus megjelenés



## A METRO Dizájn Nyelv

Az infografikus megközelítés szellemében a Microsoft kialakította saját ún. *dizájn nyelvét*, amely a Windows Phone 7-es készülékeken jelent meg először. Azóta ez a megjelenés gyakorlatilag a Microsoft összes termékében visszaköszönt, rohamtempóban vette át a stafétát a régebbi felhasználói felületektől, olyan termékekben, mint például az Xbox360 Dashboard vagy éppen az új Windows 8 UI.

A dizájn nyelv csupán az alapvető szellemiséget határozza meg, néhány alapvetést fektet le. Nézzük meg ezeket közelebbről!

### Ismerős megjelenés

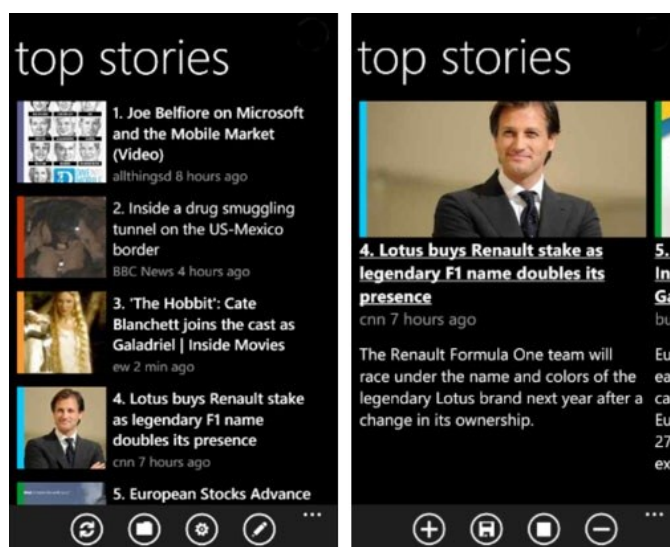
A felhasználói felület nem más, mint a kommunikációs csatorna a felhasználókkal, ezen keresztül adható tudtukra, hogyan is működik az alkalmazás. A felhasználók nehezebben azonosulnak mindennel, ami új és idegen számukra. Az életünk és a környezetünk tele van olyan elemekkel, melyeket jól megszoktunk, ismerősek és könnyen érthetőek: jelzések és apró infografikák a reptereken, a metró aluljárókban, a pályaudvarokon, a bankokban, az utcán. A Metro is ebből az eszköztárból merít ikonok, jelzések, képek, feliratok alkalmazásakor, amikor az elsődleges szempont a tájékozódás, az egyértelmű vizuális kommunikáció és az érthetőség. A 2-4 ábrán láthatjuk a Metro dizájn forrásait.



2-4 ábra: Metro dizájn források

### Tiszta és pehelykönnyű

A felhasználói felületnek gyorsnak és válaszképesnek kell lennie, de ugyanakkor pehelykönnyed működést és érzést kell biztosítani. A bonyolult felületek, a nehézkes megjelenés csupán a komplexitáshoz és a zsúfoltság érzéséhez járulhatnak hozzá. A *Metro* felület könnyed, letisztult és nyílt, bátran alkalmazza a szabad és üres területeket, valamint célorientált, ahogyan ezt a 2-5 ábra is tükrözi.



2-5 ábra: Letisztult, szabad területben gazdag megjelenés

### Digitális eredetiség

Nincs szükség arra, hogy másnak mutassuk az eszközt, mint ami valójában. Legyünk őszinték, ez egy digitális termék! Ma már nincs szükség arra, hogy egy gombot 3D-snek tüntessünk fel ahhoz, hogy jelezzük, a gomb kattintható! Nincs szükség fölösleges grafikai elemekre, legyünk egyszerűek, letisztultak és modernnek, koncentráljunk a lényegre!



2-6 ábra: Digitális megjelenés

### Gyönyörű tipográfia

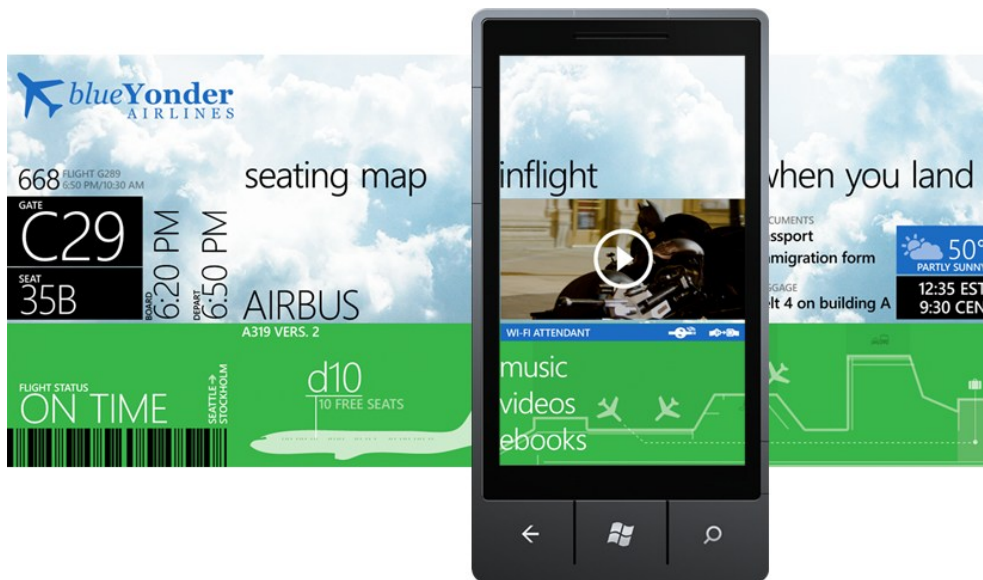
Fölszemes grafikai elemek nélkül is készíthetünk rendkívül ízléses és gyönyörű felhasználói felületeket. Koncentráljunk a szép betűtípusokra, figyeljünk oda a méretekre, az egyensúlyra, a hangsúlyozásra, a jó vizuális hierarchiára! Legyen a tartalom maga gyönyörű! Megfelelő betűtípusok és egyensúly alkalmazása mellett egyetlen fehér papírlap is lehet rendkívül esztétikus és vonzó.

### Állandóan mozgásban

A mozgás, az animációk jelentik a felhasználók és az eszközök közötti kétirányú interakció gerincét. Az érintőkijelzőnek köszönhetően természetes mozdulatokkal vezérelhetjük az alkalmazásainkat. A felhasználók számára azonnal jeleznünk kell, hogy műveleteik, tevékenységük milyen hatással bír a felületre nézve. A vizuális visszajelzés elsődleges eleme az animáció. Gyors, könnyed animációkkal pillanatok alatt megtaníthatunk bárkit, hogyan használja az alkalmazásunkat. Az animációk nem öncélúak, hanem tanító, informatív jellegűek. A jól kidolgozott animációk meghatározzák az alkalmazás ritmusát és dinamikáját is.

### A tartalom az elsődleges

A felhasználói élményt tervező szakértők körében jó ideje elfogadott és hangsúlyozott elv a „Content is King”, azaz a „tartalom mindennekfelett” elv. Különösen fontos szerepet játszik ez a gondolat érintőkijelzős alkalmazások tervezésekor. A felhasználók ezeket az alkalmazásokat érintéssel („tapogatással”) fedezik fel, szavakkal, képekkel, videókkal próbálnak interakcióba lépni. Ilyenkor a felületen a keret, a túl sok fölösleges, öncélú vizuális elem gátolja a felhasználó szabad mozgásterét, a felfedezést, megértését az alkalmazás működésére vonatkozóan. Az egyik legfontosabb irányelv a Metro dizájnban a tartalom elsődlegessége a kerettel szemben. Koncentráljunk a tartalomra, hagyjuk el a fölösleges keretet, és építsünk minimál-dizájnnal rendelkező, letisztult, tágas felhasználói felületeket!



2-7 ábra: Tartalom a középpontban

A Metro dizájn nyelv, melynek elveit szem előtt tartva professzionális, innovatív, de ugyanakkor konzisztens, új generációs felületeket alakíthatunk ki, mára a Microsoft saját termékeinek elsődleges dizájnává nőtte ki magát. A Windows Phone 7 az első olyan termék, amely alkalmazza ezeket az elveket minden szoftverében, kezdve a játék - és szórakozáscentrikus Xbox Live Hub-bal, a szociális hálókat összefogó People Hub-on át, egészen a professzionális alkalmazásokig, mint például a Office termékcsalád (2-8 ábra).



2-8 ábra: Metro-s megjelenés alkalmazás típustól függetlenül

## Design vezérelt fejlesztés

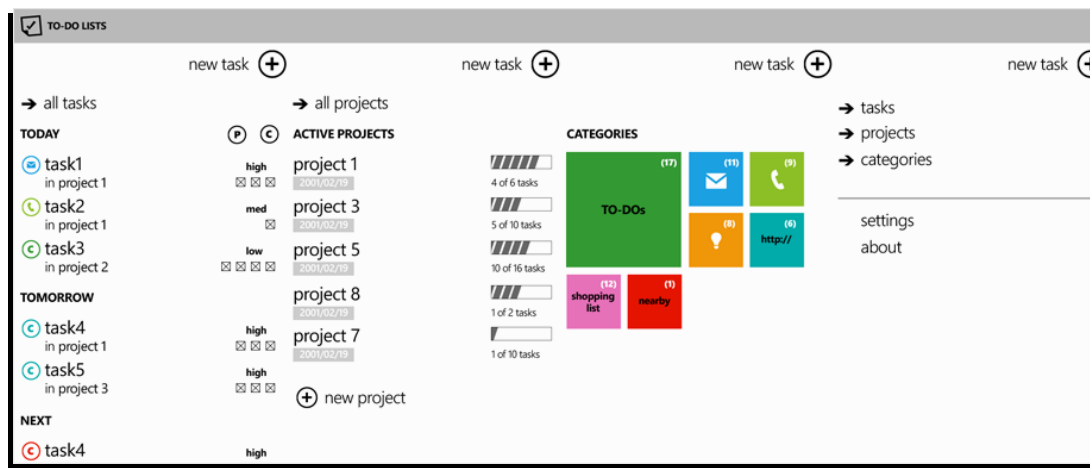
A Windows Phone 7 alkalmazásfejlesztés eszenciája némileg eltér az eddig megszokottól. Ebben az esetben ugyanis a fogyasztói piacra fejlesztünk, ahol felhasználók ezrei, remélhetőleg tízezrei fogják használatba venni és „nyúzni” alkalmazásainkat. Ennyi különböző ember igényeinek kielégítése igen komoly és nehéz feladat. Nem véletlen, hogy komoly szakmák létrejöttét eredményezték ezek a körülmények. El kell fogadnunk, hogy az esetek többségében nem bízhatjuk a fejlesztőre a felhasználói élmény kialakítását, hanem azt a megfelelő szakembereknek kell kezükbe venniük. Ők a *User Experience Design*erek.

A *User Experience Design* (UXD), vagyis a felhasználói élmény tervezése mint tevékenység számos területet foglal magába. Komolyabb rendszerek építésekor ezekre a területekre külön specializálódott szakembereket alkalmaznak. Annak az esélye azonban, hogy egy teljes UXD csapatot alkalmazhatunk mobil alkalmazásaink fejlesztésénél, elég csekély. Elkeserednünk azonban nem érdemes, kellő

odafigyeléssel mi magunk is kellemes élményt alakíthatunk ki. Ehhez azonban tisztában kell lennünk a kulcsfontosságú területekkel!

### Tartalom és információs architektúra

A Metro egész szellemisége a tartalom köré épül, a tartalom az, amelynek ki kell emelkednie, annak kell ragyognia. Az alkalmazás középpontjában a tartalom van, ezért létfontosságú, hogy gondosan elemezzük, hogy mit és hogyan mutatunk meg a felhasználónak! Ne a vizualitásból induljunk ki, hanem az értékből, az információból! Döntsük el, melyek a legfontosabb elemek, melyeket láttatni kell a felhasználóval, mi a legértékesebb információ! Miután ez a tudás már a birtokunkban van, nekiállhatunk megtervezni, hogy miként reprezentálhatjuk ezt látványosan. A weben tömegével megtalálható infografikai példák komoly segítséget nyújthatnak. Szintén sok ötletet meríthetünk egyéb sikeres alkalmazásokból. Használjuk bátran a vizuális eszközöket, grafikonokat, infografikákat! Végig tartsuk szem előtt, hogy a tartalom megfelelően tálalva, prioritás és érték szerint kerüljön megjelenítésre (2-9 ábra)!



2-9 ábra: Tartalom meghatározása Metro elvek szerint

### Használhatóság és ergonómia

A felhasználói élmény tervezésének másik fontos területe az ergonómiai kérdések tisztázása, a használhatóság biztosítása. Ez igen tág témakör, amelyben a vizuális hierarchiától kezdve a navigáción át, egészen az apró részletekig, sok mindent gondosan meg kell terveznünk. A legfontosabb dolog, hogy a felhasználó fejével gondolkodjunk, és ennek megfelelően alakítsuk ki a felhasználói felületet! Sajnálatos módon az objektivitásukat még a gyakorlottabb szakemberek is igencsak nehezen tudják megőrizni. Ha elég sokat foglalkozunk a saját termékünkkel, előbb-utóbb nem vesszük észre benne a finomságokat, rosszabb esetben a nagyobb bajokat sem. Ezért a legfontosabb a kezdő ergonómus számára, hogy tesztelje le, amit csinál. Rendkívül sokat segíthetnek az ún. *folyosó tesztek*: kapjunk el valakit a folyosón és kérjük meg, hogy hajtson végre valamit az alkalmazásunkkal. Teszteljünk minél többet, figyeljük a felhasználóink „ügyetlenkedését”, kérjük ki mások véleményét, és folyamatosan finomítsuk a felhasználói felületünket!

### Esztétika és vizuális élmény

A dizájnerek fontossága soha sem volt kétséges. A fogyasztói termékek piacán azonban elengedhetetlen a szép, esztétikus alkalmazás, a kiemelkedő vizuális élmény. Ne essünk tévedésbe! Az a gyakorlat, hogy a Metro dizájn elveit betartjuk, és egy fekete-fehér alkalmazást készítünk, nem elegendő. A közepszerűségből csak úgy emelkedhetünk ki, ha maradandó élményt nyújtunk! Sok mindenre sajnálhatjuk az erőforrásainkat, a vizuális dizájnról nem.





2-10 ábra: Dizájn alkalmazása a tartalom fölött

## Csináld magad!

A dizájn-vezérelt tervezés legfontosabb eleme maga a tervezési folyamat. Számtalan szoftver szenved a dizájn és a UI tervek hiánya következtében előálló káosztól. A szöveges specifikáció, a használati esetek alapján történő fejlesztés nem visz közelebb a jó felhasználói élményhez, helyenként még a szoftver és a felület instabilitásához is hozzájárulhat.

### A dizájn szórakoztató!

Tervezzünk magunk! Csupán három dologra van szükségünk a munkához! Papírra, ceruzára és arra, hogy ne munkának fogjuk fel, amit csinálunk. A dizájn szórakoztató, az ötletelésről szól. Az egyik legizgalmasabb pillanat és folyamat, amikor az alkalmazás körvonalazódni látszik. Rajzolás, illetve az interakciós folyamat végiggondolása közben azonnal megmutatkoznak a rossz koncepciók, a hibás gondolatok. Az ötletelés, a brainstorming hozzájárul ahhoz, hogy egyre jobb és minőségibb termék szülessen.

### Innováció tervezés közben

Amikor az alkalmazás megjelenik a szemünk előtt, amikor a funkciók, az ötletek záporoznak, születnek azok a gondolatok, amelytől a termék igazán innovatív lehet. Ezek azok az órák, napok, amelyek meghatározzák, hogy az elkövetkező hónapok kemény munkájának eredménye gyümölcsöző lesz vagy éppen csúfos bukás. Írjuk le az ötleteket, semmit se zárjunk ki! A csoportos ötletelés fantasztikus eredményekre vezethet. Gondolataink rendszerezéséhez nagy segítséget nyújthatnak a különböző mind-mapping szoftverek.

### A probléma megértése

Miközben a tervezés zajlik, a felhasználói felületek épülnek, és az innovatív gondolatok születnek, egyre jobban megértjük a problémakört. Ez nagyon fontos „mellékhatása” a tervezési folyamatnak. Sokkal jobban átlátjuk a feladatot, azt hogy a felhasználóknak - mind a funkciók, mind a felhasználói felület tekintetében – hol helyezkednek el a prioritások és a súlypontok. Időközben nem árt újra és újra végigszaladni a terveken és újraértékelni, mi is az, amit építünk, és hogyan szolgálhatja a legjobban a felhasználó igényeit.

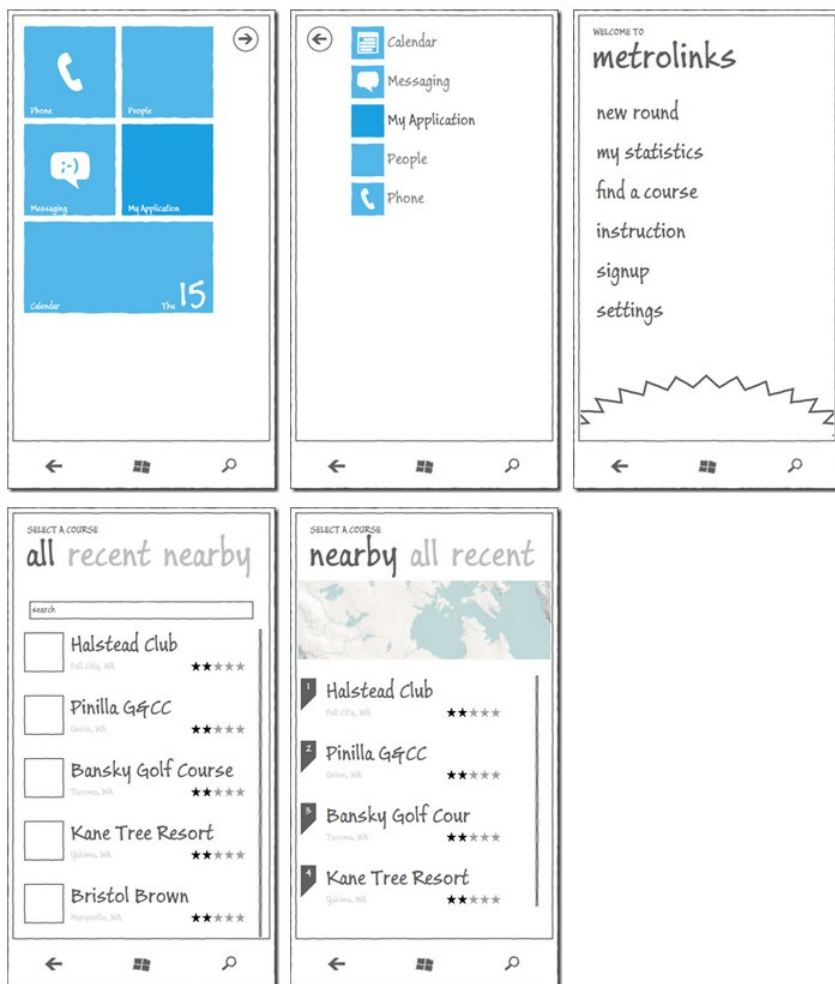
### Magasabb termékminőség

A pontos UI tervezés komoly előnye, hogy a fejlesztés során sokkal egyértelműbb, hogyan alakulnak majd a felületek, az újratervezés minimális lesz, és ennek eredményeképpen jóval kevesebbet fogunk hibázni a fejlesztés alatt. Azok a termékek, melyeknek a felhasználói felületét előre megtervezték, sokkal jobb minőségűek, mint az ad-hoc „evolúciós” felhasználói felülettel rendelkező társaik.

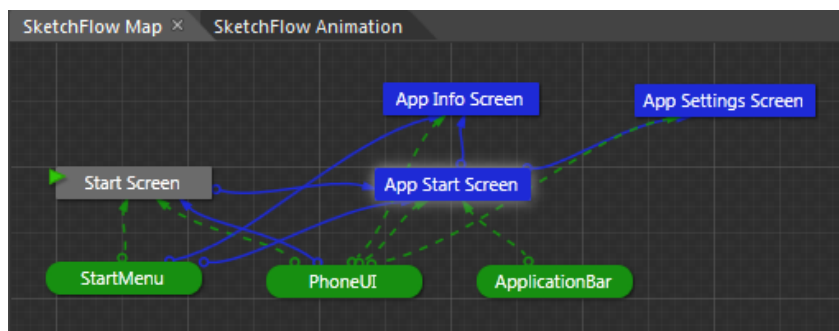
Amennyiben a felhasználói felület tervezését formális eszközökkel támogatni kívánjuk, használhatunk erre specializálódott eszközöket is, mint például az Expression Blend Sketchflow.

### Tervezés Sketchflow-val

A Microsoft Expression Blend for Windows Phone termék rendkívül hatékony eszköz. Ez a fejlesztőeszköz Windows Phone fejlesztéshez ingyenesen elérhető. A felhasználói felület végleges kialakításában, a hatékony és gyors munkában elengedhetetlen szerepe van. Az Expression Blend egyéb változataiban lehetőségünk van ún. Sketchflow projektek létrehozására is. Ezek a projektek kizárólag sketchek, vázlatok, dinamikus prototípusok készítésére alkalmasak. Windows Phone fejlesztéshez elérhető a Codeplex-en, Sketchflow-s Projekt sablon a Blend-hez (<http://wp7sketchflow.codeplex.com/>), melynek segítségével Windows Phone alkalmazások prototípusát készíthetjük el. (2-11 és 2-12 ábrák)



2-11 ábra: Sketchflow-val tervezett dinamikus prototípus



2-12 ábra: Sketchflow navigációs térkép

Sketchflow-t használva akár saját magunk számára, akár ügyfeleink számára egyértelműen kommunikálhatjuk, miként fog működni az alkalmazásunk. A Sketch projektek tervezése elsősorban a felhasználói élmény tervezésről szól, a funkciók, a logikus és helyes működés az elsődleges. Azaz ebben a fázisban ergonómiai szempontokat veszünk figyelembe.

Papíralapú tervezésnél érdemes lehet külső sablonokat felhasználni. Például: a Windows Phone 7 Sketch Padet (<http://www.uistencils.com/products/windows-phone-sketch-pad>) vagy a Windows Phone 7 Sticky Padet (<http://www.uistencils.com/products/windows-phone-sticky-pad>).

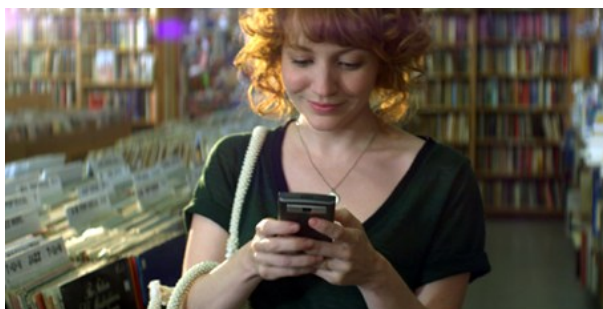
## Élmény vagy alkalmazás

Mobilalkalmazás boltok nem olyan régóta léteznek, de már megjelentek olyan tanulmányok, melyekben azt elemzik, hogy az alkalmazások életciklusa hogyan alakul. Általános mintaként azonosították, hogy az alkalmazások mindössze 10%-a éri meg a használatbavételt követő 100-ik napot. Az alkalmazások nagy részét a letöltést követő pár napban a felhasználók elhagyják. Ennek részben a temérdek „egy funkciós” alkalmazás az oka, amelyeknek nem céljuk a felhasználói bázis kialakítása, csupán az egyszeri letöltés. A legsikeresebb alkalmazások – amelyek folyamatosan a top 50-ben szerepelnek – sok energiát fektetnek felhasználói bázisuk megtartásába és kielégítésébe.

Hogyan érdemes hát nekiállnunk az alkalmazás megtervezésének? Windows Phone alapú rendszerek esetén egy különleges koncepcióval gazdagabbak lehetünk. A konkurens okostelefon gyártók az alkalmazásokra szó szerint alkalmazásként, „app”-ként gondolnak. Windows Phone alatt azonban nem csupán egyszerű alkalmazásokról van szó, sokkal inkább integrált élményről. Ahhoz, hogy kiemelkedő élményt nyújtsunk, néhány dolgot szem előtt kell tartanunk!

### *Ismerjük meg a felhasználóinkat!*

Az alkalmazás fejlesztésekor a célközönség megválasztása és azonosítása kulcsfontosságú. A Metro Design kialakításakor a Microsoft is meghatározta saját célközönségét két ún. *perszóna* definiálásával. Ők Anna és Miles (2-13 ábra).



**2-13 ábra: A Windows Phone 7 átlagos felhasználói**

Anna sikeres PR szakértő és elfoglalt fiatal anyuka, Miles pedig építész saját vállalkozásában. A Windows Phone 7 tervezésekor az UX csapat ezt a két fiktív személyt használta a felhasználói igények meghatározásában, pontos definiálásában. Saját alkalmazásunk tervezésekor is kiemelkedő szerepe van a célközönség pontos meghatározásának. Könnyebb és precízebb a tervezés minden fázisa, amennyiben

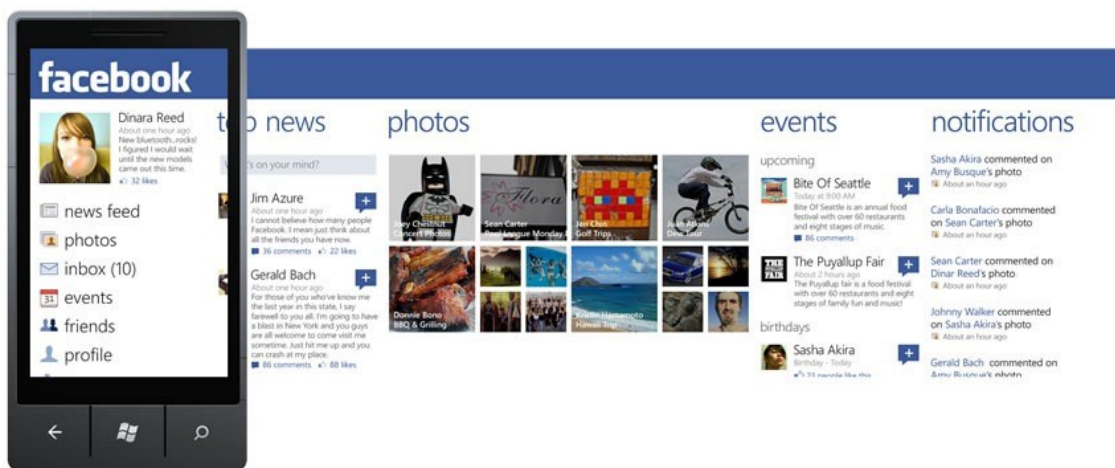
megszemélyesítjük a jellemző felhasználóinkat. A korosztály, a nem, a közös jellemzők meghatározása jelentősen leegyszerűsíti a tervezés folyamatát.

### Releváns tartalom

Az alkalmazások élettartamát és életciklusát vizsgáló tanulmányokban a legtovább élő alkalmazások jellemzően a sport- és hírközpontú alkalmazások. Friss és értékes tartalom, újdonság szolgáltatása nélkül szinte lehetetlen a felhasználóinkat visszavonzani. A felhasználói kör számára értékes és érdekes tartalmat próbáljuk szolgáltatni a lehető leggyakoribb rendszerességgel!

### Személyes alkalmazások

Az ember számára a legérdekesebb saját maga. Azok az alkalmazások, amelyek róla szólnak, az ő igényeiről, amelyek vele foglalkoznak, amelyek úgy tesznek, mintha szinte csak neki íródtak volna, sokkal sikeresebbek. Nagyon jó példa erre az Amazon alkalmazása, illetve weboldala: releváns, személyes, személyre szabott, naprakész tartalom és ajánlatok. Minél inkább a felhasználó képére tudjuk formálni az alkalmazást, annál nagyobb sikerre számíthatunk!



2-14 ábra: Személyes alkalmazás (Facebook)

### Hasznos, jól használható, kíváncsú

A tökéletes alkalmazás hasznos, a felhasználó számára értékes funkcionalitással bír, könnyen használható és egyszerűen tanulható. Jól integrálódik az operációs rendszer megszokott jellemzőivel (pl. navigáció), és kíváncsú, jó érzés használni.

Ezek a tényezők együttesen fektetik le azokat a jellemzőket és alapvetéseket, melyek segítségével jó eséllyel népszerű és közkedvelt alkalmazást készíthetünk felhasználóink körében.

## Gyakorlati Dizájn

### Mozgás és animáció

#### A Mozgás jelentősége

A mozgás és az animáció többről szól, mint objektumok folyamatos átmozgításáról. Windows Phone esetén a mozgás és az animáció a legfontosabb visszajelzése annak, hogy a felhasználói interakciónak virtuális értelemben vett fizikai hatása van. A mozgás és az animáció minősége és jellege határozza meg elsősorban, hogy a felhasználó mennyire érzi folyamatosan válaszára késznek az alkalmazást. A gyors, sima és könnyed animációk ezt az érzetet keltik, míg a lassú, nehézkes, netán akadozó animációk nagyban rontják a felhasználói élményt.

A Windows Phone alkalmazások felhasználói felületén alkalmazott mozgásokat két részre érdemes bontani:



- **Átmenet** (*transition*): Vizuális visszajelzés arról, hogy egyik nézetről másik nézetre váltunk. Az átmeneteket a felhasználó idézi elő.
- **Animáció**: Vizuális visszajelzés, melynek forrása nem feltétlenül felhasználói tevékenység (például értesítések).

A mozgások tervezésénél fontos figyelembe venni, hogy azok ne öncélúak legyenek, mindig valamilyen fontos célt szolgáljanak, amely lehetőleg segítse a felhasználó megértését az alkalmazás működését illetően.

- Érintés nélkül, fontos információk közlése (pl. értesítések, élő lapkák)
- Felhasználói tevékenységhez kapcsolódó vizuális visszajelzések (pl. gombnyomás, navigáció)
- Tanító jellegű, megértést segítő animációk (pl. egy kép a helyéről kerül kinagyításra érintés hatására, majd bezáráskor oda kerül/zsugorodik vissza)

### ***A mozgás mint élmény***

A mozgás a felhasználói élmény kialakításában kulcsfontosságú szerepet játszik.

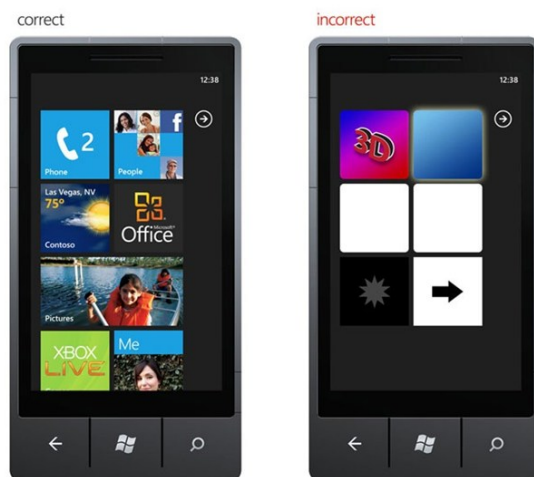
- A mozgás kellemes a szemnek. Nemcsak tanító hatása van, de szívesen nézzük, játszunk vele. Az érintőképernyős felület a legtöbb ember számára szórakoztató és érdekes.
- A mozgás segíthet a teljesítmény-problémák palástolásában. Az adatok betöltése, az információ megjelenítése okozhat némi késleltetést. Átmenetek bevezetésével elrejtethetjük ezt az átmeneti időszakot, míg anélkül – azonnali nézetváltás esetén – esetleg üres képernyő fogadhatja a felhasználót.
- A mozgás karaktert kölcsönöz az alkalmazásnak. A gyors és sima animációk azt az érzést keltik a felhasználóban, hogy „az adott pillanat és az adott hely az, ami számít: koncentráljunk a jelenre, koncentráljunk az élményre”.
- A mozgás konzisztens működéshez vezethet. Ha az átmeneteinket ügyesen és egységesen alakítjuk, akkor az alkalmazást a felhasználó konzisztensnek fogja találni, és nagyobb biztonsággal fog a képernyők között navigálni, magabiztosabban fogja az alkalmazást használni.
- A mozgás eleganciát kölcsönöz az alkalmazásnak. Nem titok, a jól megtervezett szép animációk esztétikai élményt biztosítanak az alkalmazás felhasználói számára.

### ***Windows Phone 7 specifikus komponensek tervezése***

A szépség és a konzisztens kinézet a mobil alkalmazások integráns része. A teljes összképet számos komponens együtt határozza meg. A Start lapka, a Splash képernyő, ikonok, vezérlők, melyek az operációs rendszerben található vezérlőkkel azonos működést, élményt nyújtanak. Ezek együttesen hívják fel a figyelmet az adott feladatokra, prioritásokra, releváns információkra, melyeket szép és megkapó módon célszerű találnunk. Ennek megfelelően saját élő lapkákra, animált ikonokra, szép és kellemes Splash képernyőkre van szükség.

### ***Start Menü és a lapkák***

Windows Phone 7 esetén a hagyományos Asztal és az ikonok koncepcióját felváltotta a Start képernyő és a rajta elhelyezett ún. lapkák (*tile*). Ezek gyakorlatilag az ikonokat helyettesítik. A Start képernyő átrendeázhető, a felhasználó dönti el, hogy mely alkalmazásokat jeleníti meg a Start képernyőn. A lapka kinézete jelentősen befolyásolja, hogy a telefon használatbavételekor a felhasználót milyen élmény és látvány fogadja. Így minél dekoratívabb, informatívabb egy alkalmazás lapkája, annál nagyobb a valószínűsége, hogy a Start képernyőre kerülhet. A lapkák lehetnek élő lapkák is, melyek az időben változnak, animálódnak, és alapinformációkat közölnek – például öt új olvasatlan levél van a postaládában, 30 fok lesz Budapesten, stb. Nemcsak vizuális előnyt jelent az élő lapka, de bizony jóval használhatóbbá is teszi az alkalmazást! A lapka az alkalmazás belépési pontja, az első vizuális elem, a megjelenés egyik legfontosabb eleme. Ennek megfelelően fontos, hogy egyedi, hasznos és különleges legyen (2-15 ábra)!



**2-15 ábra: Helyes és helytelen lapka dizájnok**

### A dizájn fontos

A lapkák esetén a dizájn a legfontosabb! Az elégtelen minőségű, gyengén megtervezett és megrajzolt lapkával rendelkező alkalmazás azt is kockáztatja, hogy a Marketplace-ből kisebb eséllyel kerül letöltésre. Fontos, hogy megragadja a szemet! A lapka képeknek 173x173 pixel méretűeknek kell lenniük, JPEG vagy PNG formátumban. Az ennél nagyobb vagy kisebb képek átméretezésre kerülnek, felfelé vagy lefelé, ami torzításhoz, vágáshoz vezethet. Ezenfelül az alkalmazás listába érdemes külön 63x63 pixeles képet biztosítani, ellenkező esetben az alapértelmezett lapka kerül átméretezésre. Különböző képek használata esetén azonban maradjunk minden esetben konzisztensek!

A lapkák tervezése során kerüljük el az alábbiakat:

- 3D-s lapkák használatát
- A gradiensek használatát, árnyékok alkalmazását
- Lekerekített lapkákat
- Fekete vagy fehér háttereket, melyek a megfelelő Windows Phone témák esetén „transzparens” háttérként funkcionálnak (2-16 ábra)
- Transzparens háttereket, színes belső képekkel

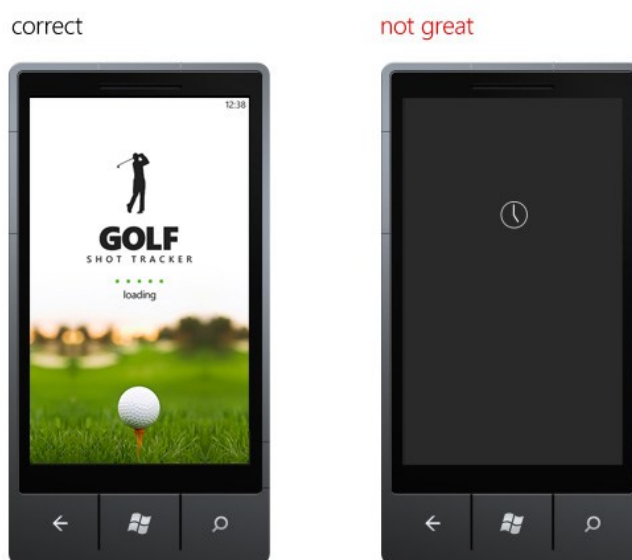


**2-16 ábra: Helytelen háttérszínek használata lapkáknál**

## Splash képernyő

Az alkalmazások indításakor számos inicializációs feladat elvégzésére lehet szükség. Ilyenkor a felhasználói felület még nem áll készen arra, hogy a felhasználók használatba vegyék az alkalmazást. A betöltés során alkalmazunk Splash képernyőket! A Splash képernyő ennek megfelelően a második szint, közvetlenül a lapkák után, amellyel a felhasználó találkozik. A betöltő képernyőn a felhasználó könnyedén eltölthet néhány (max. 10) másodpercet, így az itt töltött idő és az élmény, ami itt fogadja, jelentősen meghatározza a felhasználó hozzáállását az alkalmazás többi részéhez.

Fontos tehát a jó betöltő képernyő megtervezése. A betöltési idő lehet egy rövid pillanat is, így nem érdemes komolyabb információt megjeleníteni ezen a képernyőn. Hosszabb szövegek, instrukciók elhelyezése teljesen fölösleges. Erre a képernyőre célszerű úgy gondolni, mint az alkalmazás reklámjára, hangulatfokozóra, mielőtt szembetalálná magát a felhasználó a tényleges alkalmazással (2-17 ábra).



2-17 ábra: Splash képernyők alkalmazása

## Vezérlők használata

Windows Phone 7-ben a hagyományos vezérlők mellett számos specializált, az operációs rendszerre jellemző vezérlő is helyet kapott. Ezek a vezérlők a fejlesztők számára is elérhetők az SDK-ban vagy a Toolkiten keresztül. Minden vezérlőhöz saját tervezési és használati útmutató tartozik, melyekkel érdemes mélyebben is megismerkedni.

A Windows Phone-ra jellemző vezérlők közül kettő kiemelkedik. Ez a két vezérlő a Panorama és a Pivot.

### A Panorama vezérlő

A Panorama vezérlő egy széles jobbra-balra csúsztható, görgethető teljes képernyős konténer, és alapvetően navigációs modell is egyben. Leginkább úgy gondolhatunk rá, mint egy magazin főoldalára, ahol a legfontosabb, legérdekesebb és legizgalmasabb dolgok szerepelnek (2-18 ábra).



**2-18 ábra: A Panorama vezérlő**

A Panorama vezérlő messze túlnyúlik a látható területen, a görgetés során pedig egészen egyedi animációval és tulajdonságokkal rendelkezik. Az animáció során a tartalom és a feliratok különböző sebességgel, időben eltolódva hajtódnak végre, ezáltal rendkívül látványos, dinamikus parallaxszerű effektust eredményezve.

A vezérlő használatakor néhány dolgot célszerű szem előtt tartani:

- A háttérrel célszerű egyetlen színnel vagy egy óvatosan választott háttérképpel kitölteni.
- Háttérképként jellemzően JPEG ajánlott 480x800-as, illetve 1024x800-as felbontásban. Ez biztosítja a jó teljesítményt és a gyors betöltést, valamint az átméretezés elkerülését.
- Háttér meghatározásakor gondoskodjunk a megfelelő kontrasztról a fontok és a háttér között! Ne menjen a design az olvashatóság kárára!
- Ne használjunk olyan egyéb vezérlőket a Panorama egyes nézeteiben, amelyek magukban is görgethetők horizontális irányban!
- Teljesítmény megfontolások miatt szorítkozzunk maximum négy különböző nézetre egyetlen Panorama vezérlőben!
- A Panorama címe célszerűen sima szöveg legyen!

### **A Pivot vezérlő**

A Pivot vezérlő a Windows Phone saját Tab vezérlője. A vezérlő önálló, elkülönített nézeteket helyez el egymás mellett, és biztosítja a navigációt közöttük. A Panorama vezérlővel ellentétben nem egyetlen nagy téren helyezkednek el az egyes szekciók. A navigáció a menüelemek fejlécét érintve, illetve horizontális mozgatással lehetséges.

A vezérlő használatakor néhány dolgot célszerű szem előtt tartani:

- Pivot vezérlőket ne ágyazzunk egymásba!
- Ne kombináljuk a Pivot és a Panorama vezérlőket!
- Ne használjunk túl sok Pivot elemet, a felhasználók könnyen elveszhetnek bennük!
- A Pivot alapértelmezett navigációs mechanizmusát ne írjuk felül!
- Bizonyos vezérlőket nem célszerű használni a Pivot belsejében (például Slider, Toggle, Map vezérlők).

## Érintőképernyős alkalmazások tervezése

### A hardver

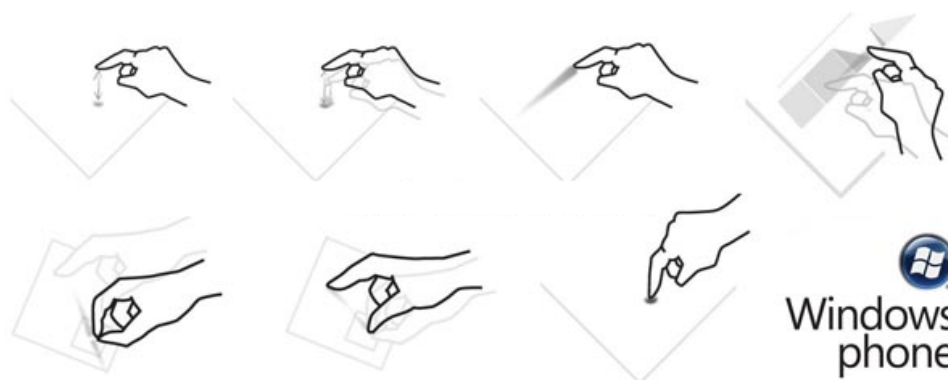
A kijelzők technológiája rengeteget fejlődött az elmúlt években. Az egyik legfontosabb áttörés a kapacitív érintőképernyők megjelenése volt. A technológia jellemzőinek köszönhetően rendkívül sima és folyamatos élményt lehet biztosítani. Pontos és csupán könnyed érintést igényel, így a ma kapható modern okostelefonok jelentős része ilyen típusú kijelzőket alkalmaz. A Windows Phone 7 telefonok követelményei között szerepel, hogy az eszközöket kapacitív kijelzővel kell ellátni.

### Az új követelmények

Érintőképernyős alkalmazások létrehozásának lehetősége teljesen új távolokat nyit a szoftveripar szereplői előtt. Az ilyen típusú felületek új, eddig még nem ismert vagy nem használt interakciós modellek születését teszik lehetővé. Ugyanakkor a kijelzők mérete – okostelefonokról lévén szó – erősen korlátozott, így az új típusú modellek kidolgozása során figyelembe kell venni az emberi tényező mellett a méretkorlátokat is.

### Interakciós modellek

Habár nem beszélhetünk szabványosított modellekről, teljesen új mozdulatok, érintés-kombinációk bevezetése kockázatos lehet. Az ilyen nagy lépésekhez a fantázián túl szükség lehet szakember (pszichológus, ergonómus) segítségére is. Ellenben számos elfogadott multitouch mozdulat közül választhatunk (2-19 ábra).



**2-19 ábra: Használható gesztusok Windows Phone-on**

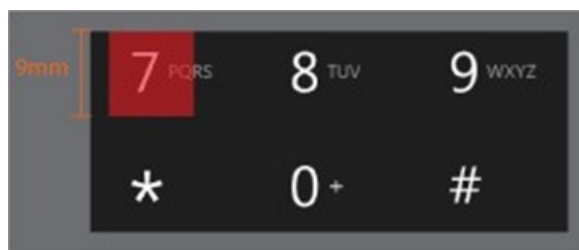
A Windows Phone 7 jelen pillanatban 12 különböző mozdulatot képes megkülönböztetni API szinten, mint például a Pinch, Slide, Tap, TapAndHold, és így tovább.

### Méretkorlátok az érintőkijelzőn

A felhasználó számára a célobjektumokkal történő érintkezést követően, közvetlenül, vizuális visszajelzést kell biztosítanunk. Ennek megfelelően bizonyos korlátokat célszerű szem előtt tartanunk.

### Minimum méretek érintéshez

Az érintésre működő felületek kialakítása finom egyensúlyozás a méretek, az objektumok közötti szabad területek és a vizuális elemek között. Az egyik, a felhasználó számára legfrusztrálóbb negatív élmény, ha nem tudja az adott gombot, linket, képet vagy bármilyen objektumot elsőre eltalálni. Hosszas használhatósági tesztelés során megállapították, hogy az ideális célobjektum területe minimum 81mm<sup>2</sup>-es (9mm x 9mm).



2-20 ábra: Érintéskor a célobjektum ajánlott területe

Ennél kisebb objektumok is elképzelhetők, tipikusan listák esetében, itt a minimum korlát a 7mm-es magasság. Ilyenkor azonban célszerű a célobjektum szélességét magasabb értékben megállapítani!



2-21 ábra: Ajánlott sormagasság

### Minimum méretek vizuális elemek tekintetében

A vizuális elemeket illetően minden olyan elemnek, amely érintésre reagál, minimum  $4.2\text{mm}^2$ -esnek kell lennie. Ennél kisebb területet a felhasználók nem fognak interaktívnek tekinteni. Ez az érték csupán a kis vizuális elemekre értendő. Az ideális méret, amelyen már a vizuális visszajelzés (lenyomás animáció) is érzékelhető  $10\text{-}15\text{mm}^2$  között mozog.

### Vizuális elrendezés

A sikertelen találatnál csak egy frusztrálóbb dolog van: amikor a felhasználó nem azt az objektumot érinti meg, amelyet szándékozott, azaz „félrenyom”. A fenti szempontok figyelembevételével ennek a kockázata jelentős mértékben csökkenthető. A helyzeten tovább javít az ún. *holt terület* helyes meghatározása. Holt területnek nevezzük azt a szabad területet, amely két célobjektum között helyezkedik el, és semmilyen interakcióval nem rendelkezik. Bár a holt területet célszerű arányaiban megállapítani, a 2mm-es minimum jó alsó határértéknek tekinthető. Ezenfelül az érinthető, interaktív területet is célszerű nagyobbobbnak választani a vizuális elem területénél.

A helyes interakciós modellek megválasztása és támogatása kulcsfontosságú szereppel bír a felhasználói élmény, a használhatóság vonatkozásában!

## Összefoglalás

Ebben a fejezetben megismertedtünk a Metro Design legfontosabb elveivel, valamint a dizájnvezérelt tervezés módszerével. Ezeket az elveket betartva olyan stabil termékek születhetnek, melyek mind számunkra, mind a felhasználóink számára élvezetes, örömteli perceket szerezhetnek. A gyakorlati tippek fontos útmutatóként szolgálnak az ergonómiai követelmények kielégítésében.

# 3. Alkalmazásfejlesztés Windows Phone-on

Az eredetileg RIA fejlesztésre kialakított Silverlight keretrendszer talán legfontosabb alap gondolata a gazdag tartalmú, összetett felhasználói felületek (a továbbiakban UI – User Interface) kialakításának támogatása. Ahogy a Silverlight „asztali” verziójában is, a Windows Phone 7-re átszabott változatban is két nagyobb részben jelenik meg ennek megvalósítása: a vezérlőkönyvtárban, illetve a keretrendszer UI-hoz kapcsolódó szolgáltatásaiban.

Előbbiben értelemszerűen a vezérlőket találjuk. Ezek azok az alapvető építőelemek, amelyek segítségével a legegyszerűbbtől a legbonyolultabb üzleti alkalmazásokig bármilyen felhasználói felületet átlátható, kézre álló módon felépíthetünk. Ennek a fejezetnek az első részében ezekkel az építőelemekkel ismerkedünk meg.

Az itt ismertetett vezérlők használatán felül a vezérlők osztályhierarchiájának kialakítása lehetővé teszi, hogy kiterjesszük azokat – új tulajdonságokat, viselkedést adjunk nekik –, illetve saját vezérlőket hozunk létre, akár több már meglévő elem együttes használatával, akár őstípusokból való leszármaztatás és a vezérlő teljes funkcionalitásának megvalósítása segítségével.

A Silverlight UI-építést és -működést támogató részének másik része azoknak a szolgáltatásoknak a halmaza, amelyek testre szabhatóvá, izgalmassá, könnyen használhatóvá teszik a vezérlőket, és lehetőséget adnak arra, hogy a korábbi UI-keretrendszerek által biztosított lehetőségeknél jóval több mindent tehessünk meg – sokkal kisebb erőfeszítéssel. A vezérlők kinézetét egy nagyszerű vezérlősablonrendszerrel, illetve stílusok segítségével változtathatjuk meg, ezenfelül pedig könnyedén animálhatjuk azokat. Különböző vizuális állapotokat hozhatunk létre, amelyekkel akár az egyes vezérlők, akár a teljes felület bizonyos helyzetekben felvett megjelenítését írhatjuk le központi, egységes módon. Az adatkötési rendszerrel és az adatsablonok használatával pedig rengeteg kódot megspórolva építhetünk dinamikus, adatfüggő felületeket.

Ebben a fejezetben ezeket a témaköröket nézzük meg közelebbről, és megismerkedünk még néhány fontos koncepcióval – például az erőforrások és a témák használatával –, amelyek a felületek építéséhez elengedhetetlenek. Szerencsére a Microsoft nagy hangsúlyt fektet a különböző technológiai közötti átjárhatóságra és a hordozható tudásra, ezért, ahogy látni is fogjuk, sok minden megegyezik az „asztali” Silverlighttal.

Mivel már rengeteg könyv megjelent Silverlight 4 témában, és az ezekben leírtak komolyabb változtatások nélkül alkalmazhatók Windows Phone 7-fejlesztésben is, ez a fejezet csak egy gyors, áttekintő összefoglalást kíván adni. Aki a WP7-es Silverlight nagytestvérének használata során már találkozott az itt leírt típusokkal és koncepciókkal, az is találhat néhány új dolgot (például az InputScope-ok), de korábban megszerzett tudása azonnal és közel száz százalékosan átvihető WP7-re.

A Silverlight alapú fejlesztésre – mint azt a korábbi fejezetek már említették – két eszközt is biztosít számunkra a Microsoft. A Visual Studio 2010 elsősorban a fejlesztőknek készült; tökéletes választás a fejlesztés teljes életciklusának felügyeletéhez a kezdeti kódírástól a tesztelésig és a telepítésig. Azonban a felületek kialakítása egyes esetekben eléggé fájdalmas tud lenni a VS használatával. Többek között a VS ezen gyenge pontját ellensúlyozandó hozta létre a Microsoft az Expression termékcsaládot.

Az Expression termékek elsősorban a dizájnereknek készültek – ahogy a Silverlight elődjét, a Windows Presentation Foundationt is azzal a céllal hozták létre, hogy segítsék a dizájnerek és a fejlesztők együttműködését. A fejezet során végig az Expression Blend 4 segítségével hozzuk létre és szabjuk testre a felületeket.



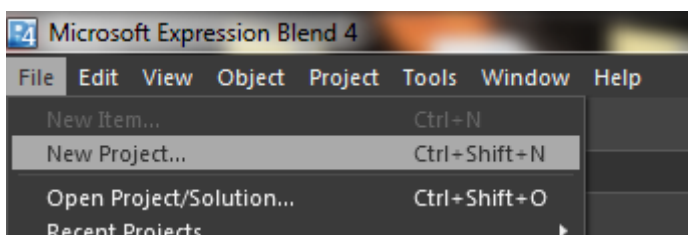
Megjegyzés: mind a Visual Studio 2010, mind az Expression Blend 4 rendelkezik ingyenes, kifejezetten WP7-fejlesztésre kihegyezett verzióval, amelyek részét képezik a WP7 fejlesztői csomagnak.

## Alapvető WP7 vezérlők

Ebben a részben először az alapvető parancskiadási vezérlőkkel foglalkozunk, majd áttérünk a szövegmegjelenítésre és a szövegbevitelre. A **ListBox** osztályon keresztül áttekintjük a listavezérlők általános használatát. Az egyszerűség kedvéért egyetlen alkalmazáson belül helyezzük el ezeket a vezérlőket. Hozzunk is létre egy alkalmazást a Blend 4 segítségével!

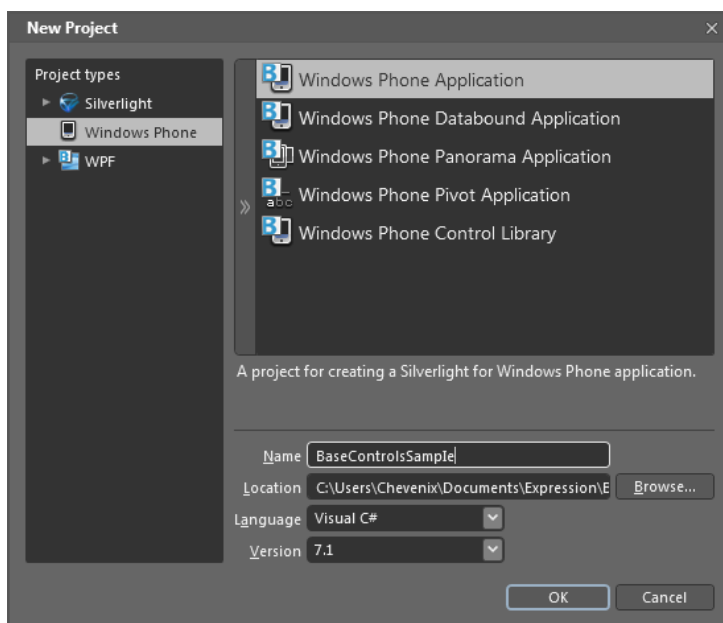
Az Expression Blendet a Start Menü az All Programs menüpontban a Microsoft Expression mappában találjuk. Elindítása után egy nyitóképernyő fogad, amelyen létrehozhatunk új projektet vagy megnyithatunk egy már létezőt, illetve segítséget kérhetünk a program használatával kapcsolatban (Help fül). A Samples fülön mintaalkalmazásokat tekinthetünk meg, a Close gombbal (Alt+C) bezárhatjuk az alkalmazást.

A legtöbb programnál szokásos főmenü itt is elérhető. A File menü New Project menüpontjára kattintva hozhatunk létre új projektet (3-1 ábra).



3-1 ábra: Új projekt létrehozása Expression Blenddel

A megjelenő ablakban választhatjuk ki, hogy milyen projektet szeretnénk létrehozni, illetve megadhatunk egyéb beállításokat. A bal oldali listából a Windows Phone pontot, a jobb oldalon pedig a Windows Phone Application projektsablont kell választanunk. Alul nevet adhatunk az alkalmazásnak (**BaseControlsSample**), kiválaszthatjuk a projekt fizikai helyét a merevlemezen, nyelvet választhatunk (főtt marha nincs, érjük be a C#-pal), illetve beállíthatjuk a Windows Phone verziót (7.1), amint azt a 3-2 ábra mutatja.



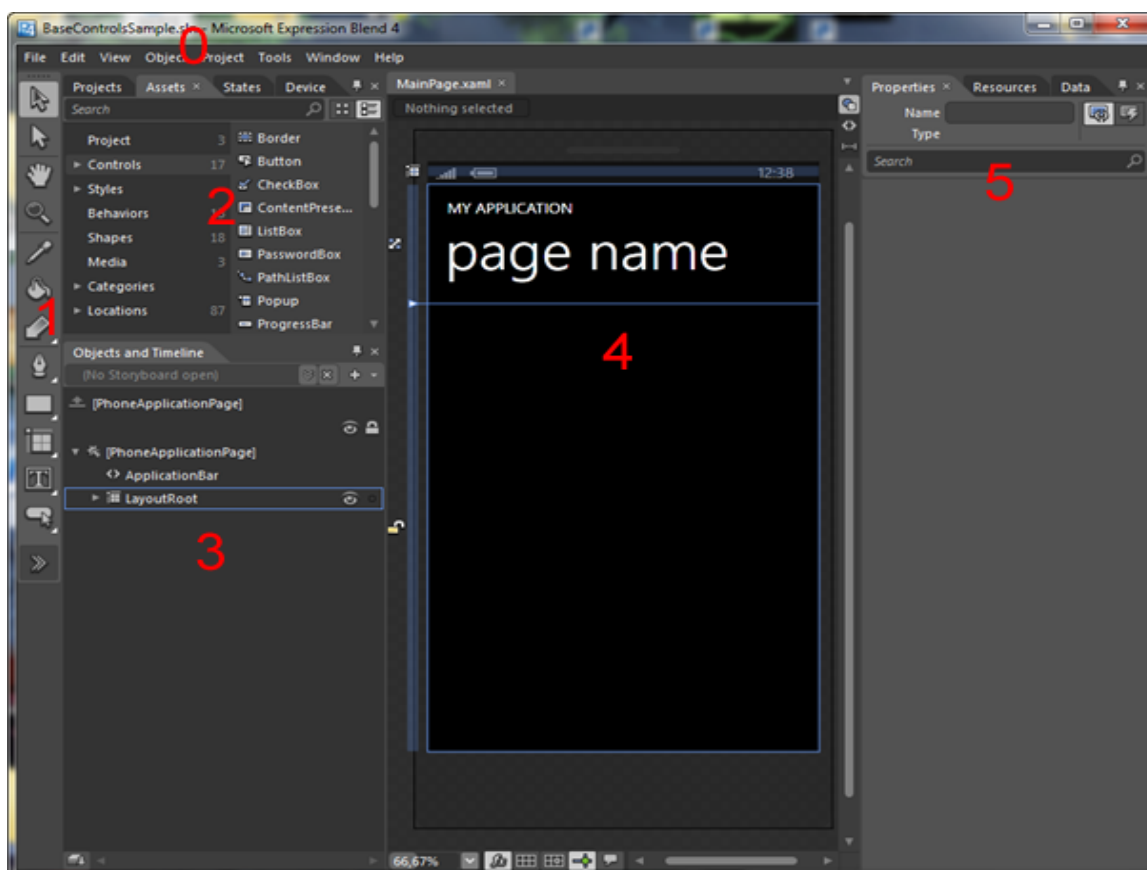
3-2 ábra: Sablon kiválasztása



Tartsuk észben, hogy a Blend alapbeállításként nem ugyanabban a könyvtárban hozza létre a projekteket, mint a Visual Studio, hanem a **C:\Users\<felhasználónév>\<dokumentumok>\Expression\Blend 4\Projects** mappában!

A választott 7.1-es OS verzió ne tévesszen meg senkit! A Mangóként ismert, 7.5-ösként reklámozott OS belső verziószáma 7.10.7720.68.

Az OK gombra kattintva a Blend létrehozza a projektet. Ismerkedjünk meg a felülettel, amelynek elemeit a 3-3 ábra mutatja be!



**3-3 ábra: Az Expression Blend felületének részei**

## 0: A menüsor

### 1: A toolbox (szerszámoszláda)

Ez jelentősen eltér a Visual Studióban megszokottól: nem egyszerűen a vezérlők listája, hanem minden a felület kialakításával kapcsolatos eszközt tartalmaz. A tetején a kiválasztáshoz szükséges kurzorok foglalnak helyet (ezek használatával tudunk valamit közvetlenül a tervezőfelületre kattintással kiválasztani, szerkeszteni). Lejjebb, a kéz ikonnal tudjuk egyszerűen görgetni a tervezőfelületet. A nagyító ikon segítségével nagyíthatunk a felület egy részére. Lejjebb, a harmadik csoportban találjuk a primitív vezérlőket szimbolizáló egyszerű négyzetet. Alatta találhatók az elrendezés-vezérlők (Grid, négy részre osztott négyzet ikon), a szövegvezérlők (keretes T betű), illetve egyéb gyakran használt vezérlők (gomb ikon).

A *toolbox* legelső eleme jelen pillanatban egy dupla, jobbra mutató nyíl. Erre kattintva érjük el az ún. *Asset Library*-t, ahonnan egyéb, a toolboxon nem szereplő vezérlőket és egyéb felhasználható típusokat érhetünk el. Ha kiválasztunk valamit az Asset Library-ban, annak ikonja a duplanyíl alatt jelenik meg.

Ha közelebbről megnézzük, néhány ikon jobb alsó sarkában egy kis nyilat vehetünk észre – ez azt jelenti, hogy nem egy eszközzel van szó, hanem egy csoportról. Azoknál az ikonoknál, amelyeknél van ilyen nyíl, a

jobb egérgombbal kattintva előhívhatjuk a helyi menüt, és azzal kiválaszthatjuk az eszközt, amelyet használni szeretnénk. Kattintsunk a jobb gombbal például a keretes T betűre (szövegvezérlők)! Az ikon mellett megjelenő lista négy eleme: TextBlock, TextBox, RichTextBox, PasswordBox. Kattintsunk a PasswordBoxra (ezúttal a bal egérgombbal), és látható, hogy a keretes T betű, mely a TextBlockot hivatott jelképezni, eltűnt, és helyette a PasswordBox ikonja jelent meg. Ennek annyi a jelentősége, hogy az itt kiválasztott vezérlőt tudjuk létrehozni, ha a tervezőfelületen kijelöljük az új vezérlő helyét.

#### 2: A fejlesztési ablak

Legelső füle (a sorrend megváltoztatható) a Projects, ami a VS Solution Explorerének helyi verziója: ebben találjuk meg a projektet alkotó összes fájlt és könyvtárat, valamint referenciákat a külső szerelvényekre. Második füle az Assets, mely a Silverlight felülettel kapcsolatos típusokat gyűjti össze, és rendszerezi. A States fülön találhatjuk meg és szerkeszthetjük a kiválasztott vezérlő vizuális állapotait – erről később szó lesz még. A Device fül segítségével pedig az alkalmazásainkat futtató emulátor orientációját és témáját állíthatjuk be, illetve itt adhatjuk meg azt is, hogy a teszteléshez az emulátort vagy egy fizikai eszközt szeretnénk-e használni.

#### 3: Objects and Timeline

Alapesetben itt láthatjuk az alkalmazásunk egy oldalának (PhoneApplicationPage) vizuális hierarchiáját. Ahogy a 3-3 ábrán is látható, egy **LayoutRoot** nevű elem a legfelső vezérlő, ebben helyezhetjük el a tartalmat. Egy sablon szerkesztése esetén itt láthatjuk a sablon felépítését, egy animáció vagy **VisualState** szerkesztésénél pedig itt jelenik meg az idővonalat (*timeline*) ábrázoló ablak, mellyel beállíthatjuk, hogy az animáció indulása utáni adott időpillanatban hogyan nézzen ki a felület.

#### 4: A tervezőfelület

Legfelül az éppen nyitva lévő fájlok listája látható (jelenleg csak a **MainPage.xaml**), alatta pedig az ún. „breadcrumb”, ami azt mutatja, hogy éppen milyen elemet választottunk ki (a fenti képen Nothing Selected, azaz éppen semmi sincs kiválasztva). A tervezőfelület jelentős részét az oldal interaktív megjelenítő felülete foglalja el. Ezen helyezhetjük el és méretezhetjük át a vezérlőket. A megjelenítőtől jobbra, felül három ikont találhatunk, ezek segítségével válthatunk a tervezési nézet, a XAML-nézet és az osztott nézet között.

#### 5: Tulajdonságok és adatok

A Properties fülön állíthatjuk be a kiválasztott vezérlő tulajdonságait, és itt iratkozhatunk fel az eseményekre. A Resources fülön találjuk az alkalmazás erőforrásainak listáját (ezekről később bővebben is szót ejtünk). A Data fül alatt a projekt adatforrásai találhatók, illetve itt hozhatunk létre teszteléshez használt adatforrást.

Az oldal tartalmaként mindössze egyetlen vezérlőelemet adhatunk meg. Ha úgy döntünk, hogy ez például egy gomb lesz, akkor a gomb kitölti majd a teljes oldal felületét – leszámítva az ApplicationBar és az ún. *system tray* által elfoglalt részeket. Ahhoz, hogy ne csak egyetlen vezérlőt helyezhessünk el az oldalon, elrendezés-vezérlőre (*layout control*) lesz szükségünk. Ezek a vezérlők azt teszik lehetővé, hogy több gyermekvezérlőt is elrendezhessünk az elrendezés-vezérlő által lefoglalt területen.

Alapesetben, a Silverlight Toolkit használata nélkül háromféle elrendezés-vezérlő áll rendelkezésre a Windows Phone 7-ben: a táblázatos elrendezést lehetővé tevő **Grid**, a Windows Formsból (vagy korábbiaktól) már ismerős, pixel alapú pozicionálást lehetővé tevő **Canvas**, és a vezérlőket egymás alá vagy egymás mellé, minimális helyigény alapján felsorakoztató **StackPanel**. Mivel az elrendezés futásidejű működése megegyezik az „asztali” Silverlightéval, itt nem térünk ki rá bővebben.

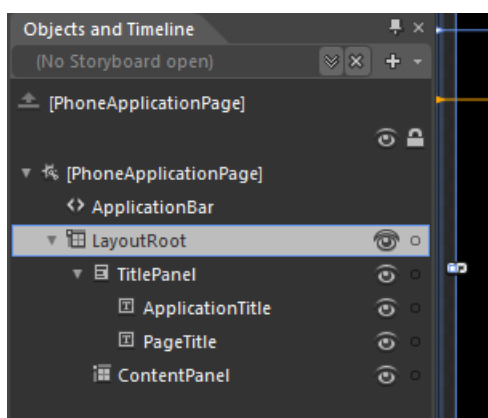
Az elrendezési rendszerről és vezérlőkről szóló további információkért érdemes az alábbi linkről elindulni:  
[http://msdn.microsoft.com/en-us/library/cc645025\(v=vs.95\).aspx](http://msdn.microsoft.com/en-us/library/cc645025(v=vs.95).aspx)

Tipp: ha utólag rájövünk, hogy egy nem megfelelő típusú elrendezés-vezérlőt helyeztünk el a felületen, nem kell XAML-t túrni a cseréhez. Az Objects and Timeline ablakban kattintsunk jobbal a lecserélni kívánt vezérlőre, és válasszuk ki a Change Layout Type pont alatt a megfelelő típust!

Mivel viszonylag ritka az a helyzet, hogy valaki ténylegesen egyetlen vezérlőt akarjon elhelyezni alkalmazása felületén, a Silverlight Phone Application projektsablon rögtön egy teljes vizuális hierarchiát épít fel a felületen. Ha az Objects and Timeline ablakra nézünk, láthatjuk ennek gyökérelemét, a LayoutRoot nevű vezérlőt, amely történetesen egy **Grid**. Ez az egyetlen tartalma az oldalunknak.

A tervezőfelületre pillantva érezhető, hogy itt ennél bonyolultabb struktúrának kell lennie – a LayoutRoottól balra lévő apró nyílra kattintva megtekinthetjük a Grid belsejében lévő felületi elemeket is. Ahogy az a 3-4 ábrán látható, a LayoutRoot két elemet tartalmaz.

Bár a kezdeti időkben nem túl gyakran kerülnek elő, érdemes tudni, hogy a vezérlőktől jobbra látható két piktogrammal tudunk eltüntetni egy-egy vezérlőt a tervezőfelületről, illetve zárolni, azaz védeni a véletlen módosításoktól.



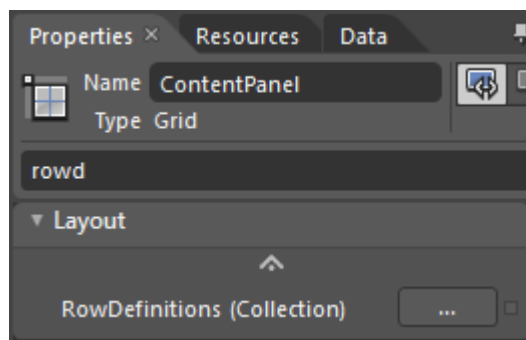
**3-4 ábra: Az Objects and Timeline panel**

A **TitlePanel** nevű gyermekelem egy **StackPanel**, amely az alkalmazás felső részét írja le: az alkalmazás nevét („MY APPLICATION”) tartalmazó **TextBlock**ot, illetve az alatta lévő oldalnevet („page name”). Ha nincs különleges igényünk, esetleg szeretnénk, hogy az alkalmazás illeszkedjen a sok másik által kialakított jól felismerhető sémába, akkor elég, ha megváltoztatjuk a két **TextBlock** tartalmát, de a struktúrához nem kell hozzányúlnunk.

A **ContentPanel** nevű gyermekelem a leginkább érdekes rész: ebben a **Grid**ben helyezhetjük el a tényleges tartalmat adó és funkciót szolgáló vezérlőket.

Észrevehetjük, hogy a vezérlőkre kattintgatva az Objects and Timeline panelen (vagy valamelyik nyíl alakú kurzorral magán a tervezőfelületen), a jobb oldalon lévő Properties panel tartalma azonnal frissül. Mivel több vezérlőt is szeretnénk elhelyezni az alkalmazáson – pontosabban a **ContentPanel**en – belül, meg kell adnunk a **ContentPanel**nek, hogy hány oszlopból és sorból álljon. Ezt úgy tehetjük meg, hogy kijelöljük azt, majd a Properties panelen megkeressük annak **RowDefinitions** illetve **ColumnDefinitions** tulajdonságait. Mielőtt azonban elvesznénk a millió tulajdonság között, érdemes madártávlatból is szemügyre venni a Properties panelt!

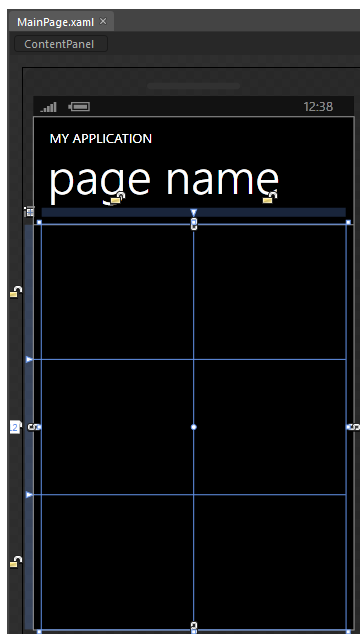
Ahogy az a 3-5 ábra mutatja, legfelül a kijelölt objektum nevét láthatjuk – ezzel a névvel hivatkozhatunk a vezérlőre a mögöttes kódban, illetve egyéb helyeken a XAML-ben. Előfordulhat, hogy a „<No Name>” felirat szerepel ebben a szövegdobozban, ugyanis a Silverlightban nem feltétlenül kell egy vezérlőnek nevet adni. A név mellett találjuk a tulajdonságok és az események között váltó két gombot, alatta pedig a kijelölt elem típusát. Mindezek alatt pedig a keresődoboz helyezkedik el. Érdemes ezt használnunk, mivel a szűrés kellőképpen intelligens (nemcsak a tulajdonságok nevének kezdete alapján szűrhetünk), illetve mert a Silverlight vezérlői tényleg **nagyon sok** tulajdonsággal rendelkeznek.



**3-5 ábra – A Properties panel**

A keresődoboz alatt találjuk a tulajdonságokat – értékeikkel egyetemben — csoportokra osztva. Itt két dologról is érdemes szót ejteni, amelyeket elsőre könnyű nem észrevenni. Az egyik, hogy a csoportok alján található egy lefelé mutató nyíl; ezzel érhetjük el a ritkábban használt tulajdonságokat. A másik, hogy sok tulajdonságtól jobbra egy kis négyzet található. Ez nemcsak dizájnelem, hanem funkcióval is bír! Az Advanced options „gomb” segítségével hozhatunk létre adatkötést (esetleg erőforráshoz kötést) az adott tulajdonság kapcsán, illetve egyebek mellett alapértelmezett értékre is visszaállíthatjuk a tulajdonságot. (Az adatkötésről és az erőforrásokról a későbbiekben lesz szó.) A gomb színe is jelentőséggel bír: ha szürke, akkor a tulajdonság az alapértelmezett értékén áll, ha fehér, akkor kézzel állították be valamilyen értékre, ha sárga, valamilyen adatkötésből kapja az értékét, és így tovább.

Visszatérve a **ContentPanel**re, ha hozzáadunk a **RowDefinitions** és a **ColumnDefinitions** tulajdonságokhoz két oszlopot és három sort, látható, hogy a tervezőfelületen is megjelennek a cellák határoló vonalai, hat egyforma cellára bontják a felületet, ahogyan azt a 3-6 ábra mutatja.



**3-6 ábra: Hat egyforma cellára bontott ContentPanel Grid**

Ennyi áttekintés után vegyük sorra azt a néhány alapvető vezérlőt, amelyek nélkül nincs élet a Silverlightban!

## Parancsvezérlők

### A Button vezérlő használata

A Toolboxon duplán a **Button** ikonjára kattintva (3-7 ábra), megjelenik egy gomb az alkalmazás felületén. Alapértelmezés szerint abban a vezérlőben jelenik meg az új elem, melyet az Objects and Timeline ablakban kijelöltünk.

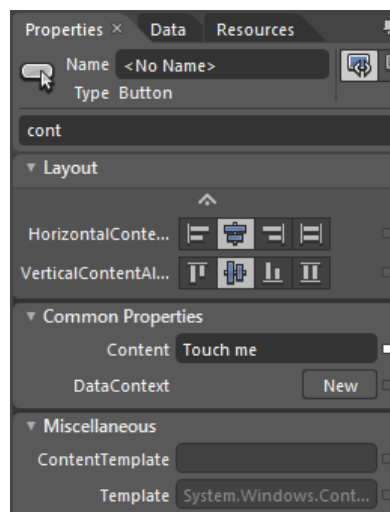


3-7 ábra: A Buttont jelölő gomb a Toolboxon

Vezérlőt úgy is létrehozhatunk, hogy egyszer kattintunk az ikonjára, majd a pluszjel formájúra változott kurzorral megrajzoljuk a felületen. Ennek a módszernek többek között az az előnye, hogy rögtön meghatározhatjuk a vezérlő pontos pozícióját, illetve méretét. Hátránya viszont, hogy egy összetettebb vizuális hierarchiában a Blend néha eltéveszti, hogy pontosan melyik elrendezés vezérlőbe, egy **Grid** esetén melyik cellá(k)ba is akartuk helyezni az új vezérlőt. **Canvas** használata esetén viszont ez a módszer nagyon hasznos.

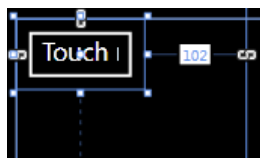
A **Button** az elsődleges parancsvezérlő: funkciója általában mindössze annyi, hogy a felhasználó rákattinthat (illetve, mivel érintőképernyős eszközről beszélünk: megérintheti, rátapinthat), és ennek hatására lefuttathatunk egy eseménykezelő metódust a mögöttes kódban.

Az első, amit szokás átállítani egy gombon, annak felirata. A **Button** a **ContentControl** őssztályból származik. A **ContentControl**-ból származó vezérlők közös ismérve, hogy egyetlen tartalmat tudnak fogadni – **Content** tulajdonságukon keresztül –, ez a tulajdonság viszont **System.Object** típusú, tehát bármilyen objektumot felvehet értékéül. A Properties panelen a **Content** tulajdonság melletti szövegdobozban írhatjuk át a gomb feliratát, ahogyan azt a 3-8 ábra is mutatja:



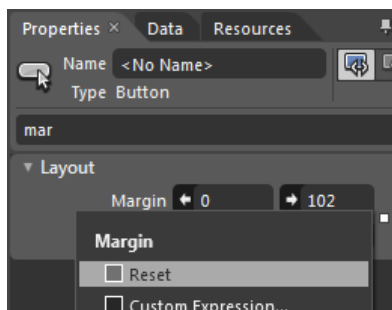
3-8 ábra: A Button Content tulajdonságának átállítása a Properties panelen

Észrevehetjük, hogy ha túl hosszú feliratot rendelünk a gombhoz, a tervezőfelületen nem jelenik meg a teljes szöveg. Ennek oka, hogy alapbeállításként a Blend a **Margin** tulajdonság segítségével méretezi a vezérlőt, vagyis meghatározza, hogy a tartalmazó vezérlő (a **ContentPanel** nevű **Grid** [0,0] cellája) széleitől mekkora távolságot tartson a vezérlő (3-9 ábra).



**3-9 ábra: A túl kicsi gomb és a Margin esete**

Ha az Advanced options segítségével alaphelyzetbe állítjuk a **Margin** (3-10 ábra), a gomb szélesebb lesz, szélétében kitölti a cellát (3-11 ábra).



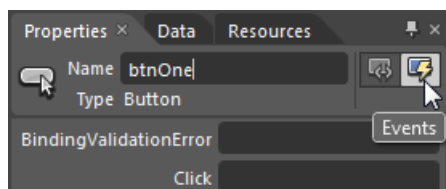
**3-10 ábra: A Margin alaphelyzetbe állítása az Advanced options Reset parancsának segítségével**



**3-11 ábra: Gomb, Marginok nélkül**

Azt is észrevehetjük, hogy gombunk még nem rendelkezik névvel. Alapértelmezés szerint nem kell, hogy nevet kapjanak a vezérlők, viszont ha hivatkozni szeretnénk rájuk a mögöttes kódban, akkor érdemes nevet adni nekik. (Esetünkben ez **btnOne** lesz.)

Ahhoz, hogy valamilyen kód lefusson, amikor a felhasználó megnyomja a gombot, a **Button Click** eseményéhez kell hozzárendelnünk egy eseménykezelő metódust. A Properties panel tetején válthatunk át eseménynézetbe, és itt a **Click** felirat melletti szövegdobozba való dupla kattintással készíthetjük el az eseménykezelőt a Blenddel (3-12 ábra).



**3-12 ábra: A Properties panel átkapcsolása esemény-nézetbe**

Ahogy elkészül a metódus – melynek neve meg is jelent a **Click** mellett –, a Blend átvált az aktuális oldal mögöttes kódjára. Itt kapcsolhatjuk azt a kódot a felülethez, amellyel megvalósíthatjuk az elvárt működést.

A bonyolultabb programoknál érdemes szeparálni a felület mögötti logikát és a tényleges üzleti logikát. Ebben az MVVM (Model-View-ViewModel) alkalmazásfejlesztési minta lehet segítségünkre. Ennek leírása túlmegy a könyv határain, illetve több keretrendszer is létezik implementálására. Josh Smith WPF-guru cikke az MVVM-ről WPF alatt itt található: <http://msdn.microsoft.com/en-us/magazine/dd419663.aspx>. Egy önállóan megírható alkalmazás elkészítésének leírása Silverlight MVVM-mel itt van részletesen leírva: <http://www.silverlight.net/learn/advanced-techniques/the-mvvm-pattern/using-the-mvvm-pattern-in-silverlight-applications>. Az ismertebb MVVM-keretrendszerek leírása a [http://en.wikipedia.org/wiki/Model\\_View\\_ViewModel](http://en.wikipedia.org/wiki/Model_View_ViewModel) linken található. (Az Open source MVVM frameworks c. rész alatti linkeket érdemes megnézni.)

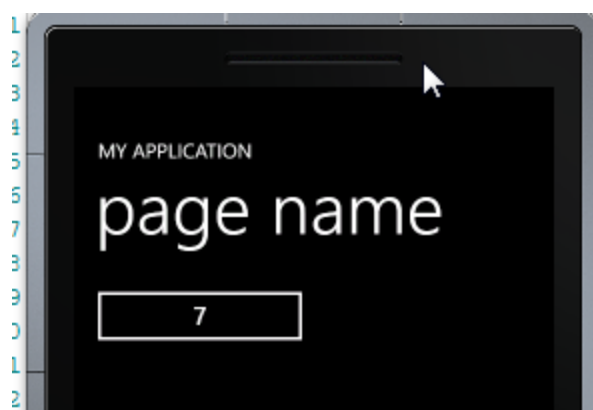
Itt annyit fog tenni a gomb, hogy amikor megnyomják, növel egy számlálót, aztán annak aktuális értékét megjeleníti a gombon. Ehhez először létrehozok egy **System.Int32** típusú változót az oldal mögöttes kódjában, majd az előbb elkészült eseménykezelőben megnövelem az értékét, és átadom a gombnak mint tartalmat. A teljes kód – leszámítva a **using** direktívákat – így néz ki:

```
namespace BaseControlsSample
{
    public partial class MainPage : PhoneApplicationPage
    {
        int taps = 0;

        public MainPage()
        {
            InitializeComponent();
        }

        private void btnOne_Click(object sender, System.Windows.RoutedEventArgs e)
        {
            taps++;
            btnOne.Content = taps;
        }
    }
}
```

Az alkalmazás futtatásához vagy megnyomjuk az F5 billentyűt, vagy kiválaszthatjuk a Project menüből a Run Project menüpontot. A Devices panel beállításai szerint a Blend vagy a WP7 emulátort indítja el, vagy a PC-hez csatlakoztatott telefonra telepíti az alkalmazást. Ahogy az alkalmazás elindult, a gombot nyomogatva növekszik az érték (3-13 ábra).



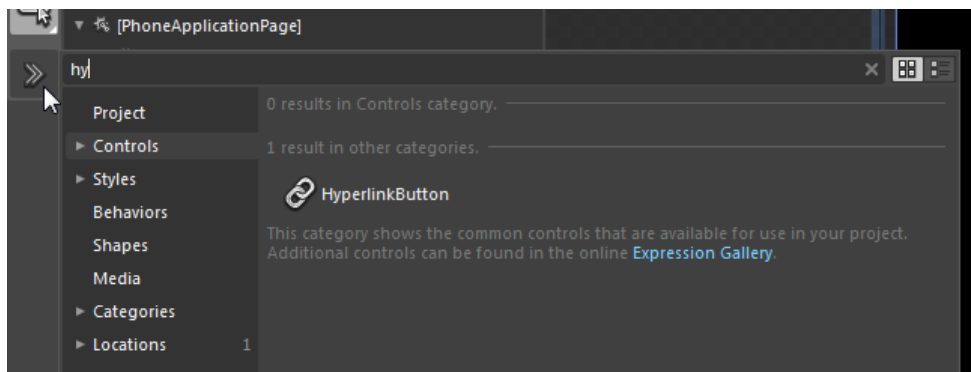
3-13 ábra: A program az emulátoron futtatva

Nem telepíthetünk minden telefonra alkalmazásokat a Marketplace megkerülésével. Be kell regisztrálnunk magunkat az App Hubon (<http://create.msdn.com/>) Windows Phone fejlesztőként, és megadni telefonunk gyári azonosítóját. Ezzel feloldjuk a zárolást, fejlesztői telefonttá tesszük az eszközt, ezután arra a Marketplace megkerülésével is telepíthető alkalmazás.

#### A **HyperlinkButton** vezérlő használata

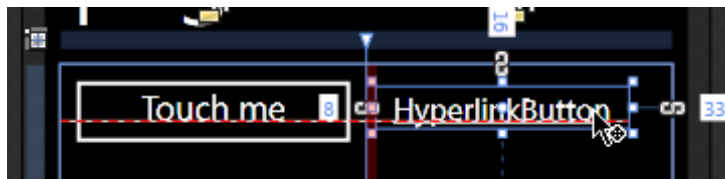
A **HyperlinkButton** vezérlő a webes hivatkozások megfelelőjének tekinthető. Tulajdonképpen működése nem sokban különbözik a **Button**étól: valami, ami kattintás hatására végrehajtja a hozzárendelt kódot.

A **HyperlinkButton** alapértelmezésben nem található meg a Toolboxban, ezért az Asset Library segítségével kell előkeríteni. A Toolbox alján lévő dupla nyílra kattintva megjelenik a fent nevezett ablak; a keresődoboz segítségével gyorsan megtalálható a **HyperlinkButton** vezérlő, amely ezután egy kattintással helyezhető el a Toolboxon, ahogyan azt a 3-14 ábra mutatja.



**3-14 ábra: Az Assets panel megnyitása, és a HyperlinkButton megkeresése**

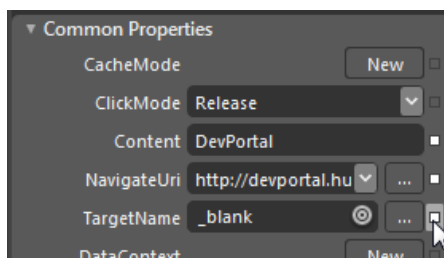
A Toolbox új elemére duplán kattintva megjelenik egy **HyperlinkButton** a felületen, amelyet aztán a kurzor segítségével elhelyezhetünk (3-15 ábra).



**3-15 ábra: A HyperlinkButton elhelyezése a felületen**

Akárcsak a **Button**, ez a vezérlő is a **ContentControl**ból származik; tartalmát a **Content** tulajdonsággal szabályozhatjuk. Szintén a Properties Panel Common Properties csoportjában található a **NavigateUri** tulajdonság. Ennek segítségével állítható be az oldal, ahová navigálni szeretnénk. Alapértelmezésként itt az alkalmazásban lévő PhoneApplicationPage-eket látjuk, de beírhatunk egy webcímet is. (Protokollal együtt kell megadni, tehát pl.: <http://devportal.hu>.) Ahhoz, hogy a navigáció működjön, be kell állítanunk a **TargetName** tulajdonságot is „\_blank”-re. Közvetlenül ezt nem tehetjük meg, de az Advanced options négyzetre kattintva kiválaszthatjuk a Custom Expression menüpontot, amellyel szabadon szerkeszthető az érték (3-16 ábra).





3-16 ábra: A NavigateUri és a TargetName beállítása

Ha ezek után elindítjuk a programot, és megérintjük a linket, megnyílik az Internet Explorer, és betöltődik a NavigateUri-ban megadott webcím.

Ha nem elég, hogy így, deklaratívan megadjuk a navigáció címét, mert például azt dinamikusan szeretnénk előállítani, akkor a **Button**nál látott módon feliratkozhatunk a **Click** eseményre, és a mögöttes kódban tetszőleges kódot végrehajthatunk.

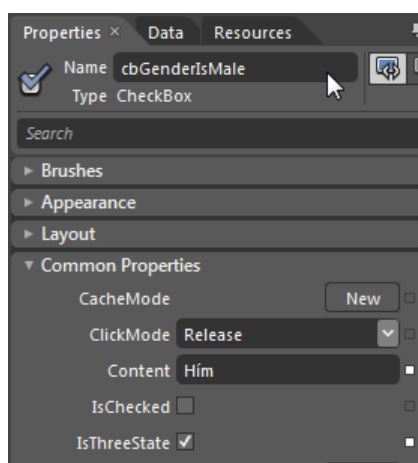
### A CheckBox használata

Ha szeretnénk felkínálni a lehetőséget a felhasználónak, hogy szabályozza a program működését, engedélyezzen vagy letiltson valamilyen opciót, a legjobb választás a **CheckBox**, azaz jelölőnégyzet. A Toolboxon ugyanabban a menüben található, ahol a **Button**. A **CheckBox** felirata, ahogy a már bemutatott vezérlőknél is, a **Content** tulajdonságon keresztül változtatható meg.

Legfontosabb tulajdonságai az **IsChecked** és az **IsThreeState**. Előbbi megmondja, hogy a jelölőnégyzetet bejelölték-e. Utóbbival azt szabályozhatjuk, hogy a jelölőnégyzet kétállapotú legyen, vagy felvehessen-e egy harmadik, határozatlan (*indeterminate*) állapotot is.

Ha az **IsThreeState** tulajdonság igaz (**true**) értékű, a jelölőnégyzet három állapotot vehet fel. Ha hamis (**false**) értékű, akkor is felveheti a harmadik állapotot, de csak programozottan; a felhasználó már nem állíthatja vissza köztes állapotba. Ez hasznos lehet, ha például nem akarjuk, vagy nem tudjuk előre megmondani, hogy melyik állapotba kellene állítani alapértelmezetten a jelölőnégyzetet, ugyanakkor nem szeretnénk azt sem, hogy a felhasználó utólag visszaállíthassa köztes állapotba.

Ha szeretnénk valamilyen kódot futtatni, amikor a jelölőnégyzet állapota megváltozik, a **Checked** (bejelölés esetén), **Indeterminate** (eldöntetlen, köztes állapotba kerülés esetén) és **Unchecked** (bejelölés törlése esetén) eseményekre iratkozhatunk fel.



3-17 ábra: A CheckBox beállítása

```
private void cbGenderIsMale_Checked(object sender, RoutedEventArgs e)
{
    MessageBox.Show("Herbert Garrison");
}
```

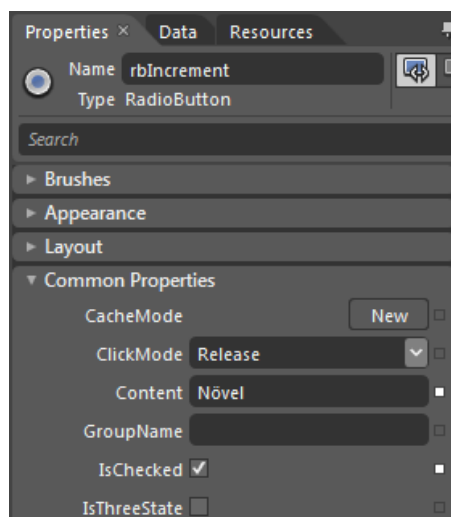
```
private void cbGenderIsMale_Unchecked(object sender, RoutedEventArgs e)
{
    MessageBox.Show("Sheila Broflovski");
}

private void cbGenderIsMale_Indeterminate(object sender, RoutedEventArgs e)
{
    MessageBox.Show("Liane Cartman");
}
```

#### A **RadioButton** használata

A **CheckBox** vezérlőkkel egyes lehetőségek egymástól független ki- vagy bekapcsolását adhatjuk a felhasználó kezébe. Előfordul azonban, hogy azt szeretnénk, ha a felhasználó több lehetőségből egyet választhatna. Ilyen esetekben használhatjuk a **RadioButton** (rádiógomb).

A **RadioButton** legfontosabb tulajdonságai és eseményei nevükben és működésükben is megegyeznek a **CheckBox**nál látottakéval (**Content**, **IsChecked**, **IsThreeState**, **Checked**, **Unchecked**, **Indeterminate**).



3-18 ábra: A **RadioButton** tulajdonságainak beállítása

Ha több csoportba szeretnénk rendezni a rádiógombokat, úgy, hogy minden csoportban kölcsönösen kizárják egymást a lehetőségek, de a különböző csoportokban lévők között már ne legyen kapcsolat, **GroupName** tulajdonságuk segítségével csoportosíthatjuk őket.

A mögöttes kódban a **RadioButton** **IsChecked** tulajdonságát figyelembe véve írhatjuk meg a program logikáját. Az alábbi példában a korábban elkészített **Button** eseménykezelőjét módosítjuk úgy, hogy ha az **rbIncrement** választógombot jelöli be a felhasználó, növeli az értéket, ha pedig nem, csökkenti azt (3-19 ábra).

```
private void btnOne_Click(object sender, System.Windows.RoutedEventArgs e)
{
    if ((bool)rbIncrement.IsChecked) taps++;
    else taps--;
    btnOne.Content = taps;
}
```

Mind a **CheckBox**, mind a **RadioButton** **IsChecked** tulajdonsága **Nullable<Boolean>** típusú, azaz a **true** és **false** értéken kívül **null** értéket is felvehet. Ezért kell előbb Booleanné alakítani, hogy true/false-ként vizsgálhassuk az értékét.



3-19 ábra: Parancsvezérlők a felületen

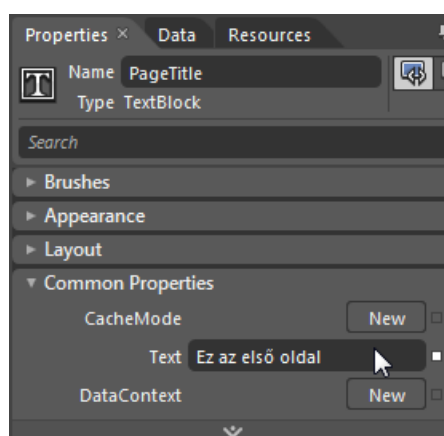
## Szövegmegjelenítési vezérlők

Nagyon kevés olyan program van, amelynek nem kell szöveget megjelenítenie. A Silverlightban erre elsősorban két vezérlőt használunk: egyszerű szövegmegjelenítésre a **TextBlock**ot, bonyolultabban formázott megjelenítésre pedig a **RichTextBox**ot.

### Egyszerű szövegmegjelenítés a TextBlockkal

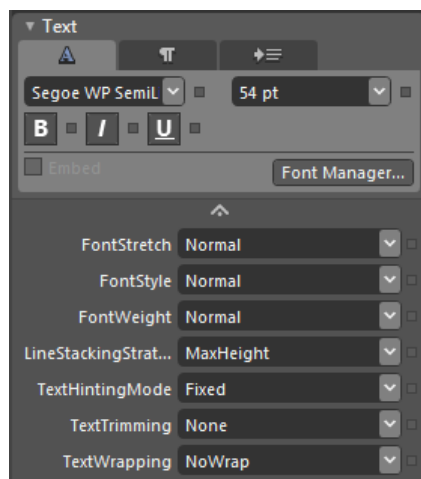
Ahogy korábban láttuk, a Windows Phone Application sablon rögtön két **TextBlock**ot is beépít alkalmazásunk felületébe. Vegyük szemre a **PageTitle** nevűt!

A **TextBlock** által megjelenített szöveget a Text tulajdonság segítségével adhatjuk meg (3-20 ábra).



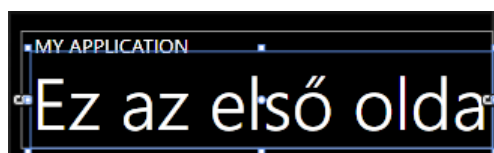
3-20 ábra: A TextBlock szövegének beállítása

A Text csoportban található tulajdonságok segítségével szabhatjuk testre a szöveg kinézetét. Egyebek között átállíthatjuk a betűtípust, a betűk méretét, illetve a szokásos (félkövér, dőlt, aláhúzott) betűformázási lehetőségeket is megkapjuk (3-21 ábra).



3-21 ábra: A TextBlock szövegének tulajdonságai

Előfordulhat, hogy a szöveg túl hosszú, és nem fér ki a **TextBlock** rendelkezésére álló helyen. Ilyenkor alapértelmezésként egyszerűen „kicsúszik”, vagyis a vége nem jelenik meg (3-22 ábra). Általában érdemes elkerülni, hogy túl sok szöveg kerüljön egy **TextBlock**-ba, de erre nem mindig van lehetőség.



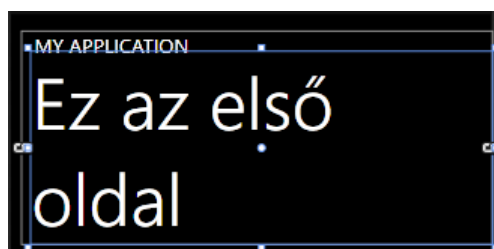
3-22 ábra: Túl hosszú szöveg

Két dolgot tehetünk, hogy kiküszöböljük ezt a problémát. Az első, hogy a **TextTrimming** tulajdonságot **None** értékről **WordEllipsis** értékűre állítjuk. Ennek eredményeképpen, amikor a **TextBlock** túl sok szöveget tartalmaz, levágja azokat a szavakat, amelyek már nem férnek ki, és három ponttal helyettesíti őket (3-23 ábra).



3-23 ábra: Túl hosszú szöveg TextTrimminggel kezelve

A másik lehetőség, hogy a **TextWrapping** tulajdonságot **Wrap** értékre állítjuk. Ennek hatására a **TextBlock** több sorban próbálja meg elhelyezni a szöveget. Amikor egy szó már nem fér ki, a következő sorba kerül (3-24 ábra).

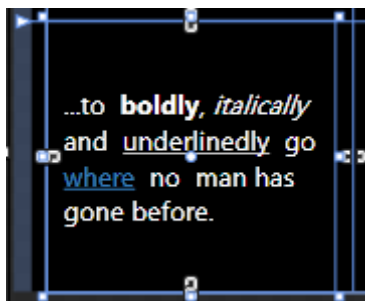


3-24 ábra: Túl hosszú szöveg TextWrappinggel kezelve

### Formázott szövegek megjelenítése a RichTextBoxsal

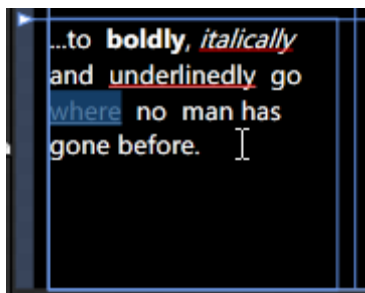
A **RichTextBox** neve azt sugallja, hogy egy szövegbeviteli vezérlőről lehet szó, a helyzet azonban — legalább is jelen pillanatban — nem ez. A RichTextBox WP7-es implementációja nem teszi lehetővé szöveg bevitelét vagy szerkesztését a felhasználó számára, csupán arra ad lehetőséget, hogy programozottan töltsük fel szöveggel, illetve egyéb elemekkel — hivatkozások, képek stb. — a vezérlőt.

Amellett, hogy az egyszerű szövegeknél bonyolultabb elemeket is meg tud jeleníteni, a **RichTextBox** igazi előnye, hogy a **TextBlock**kal ellentétben nem csupán egyféle formázást adhatunk az egész szövegnek, hanem akár karakterenként mást és mást (3-25 ábra).

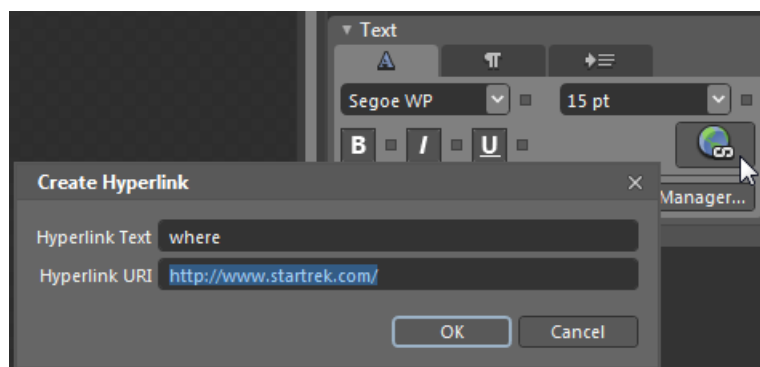


3-25 ábra: Többféleképpen formázott szöveg egy RichTextBoxban

A **RichTextBox**ot a szövegvezérlők csoportjában találjuk a Toolboxon. Amint elhelyezünk egyet a felületen, tartalma rögtön szerkeszthetővé is válik (3-26 ábra). A szöveg bármely része kijelölhető, és átállíthatóak jellemzői. Sőt, rögtön hivatkozást is befűzhetünk a szövegbe, ha a Text csoportban lévő földgömb gombra kattintunk (3-27 ábra).

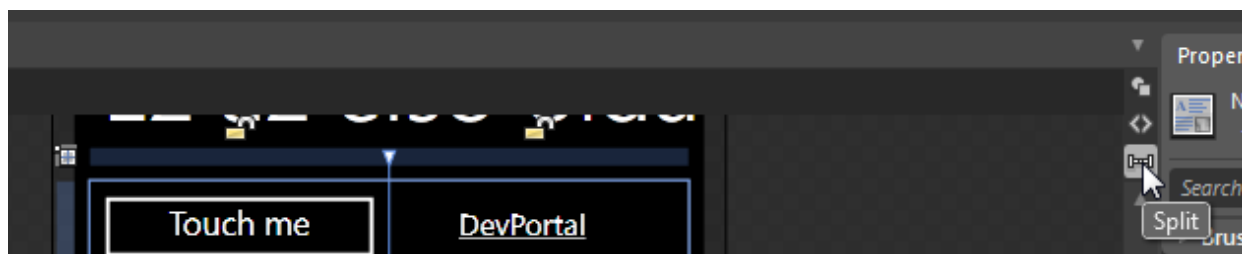


3-26 ábra: A szöveg egy részének kijelölése, hogy hivatkozássá alakítsuk



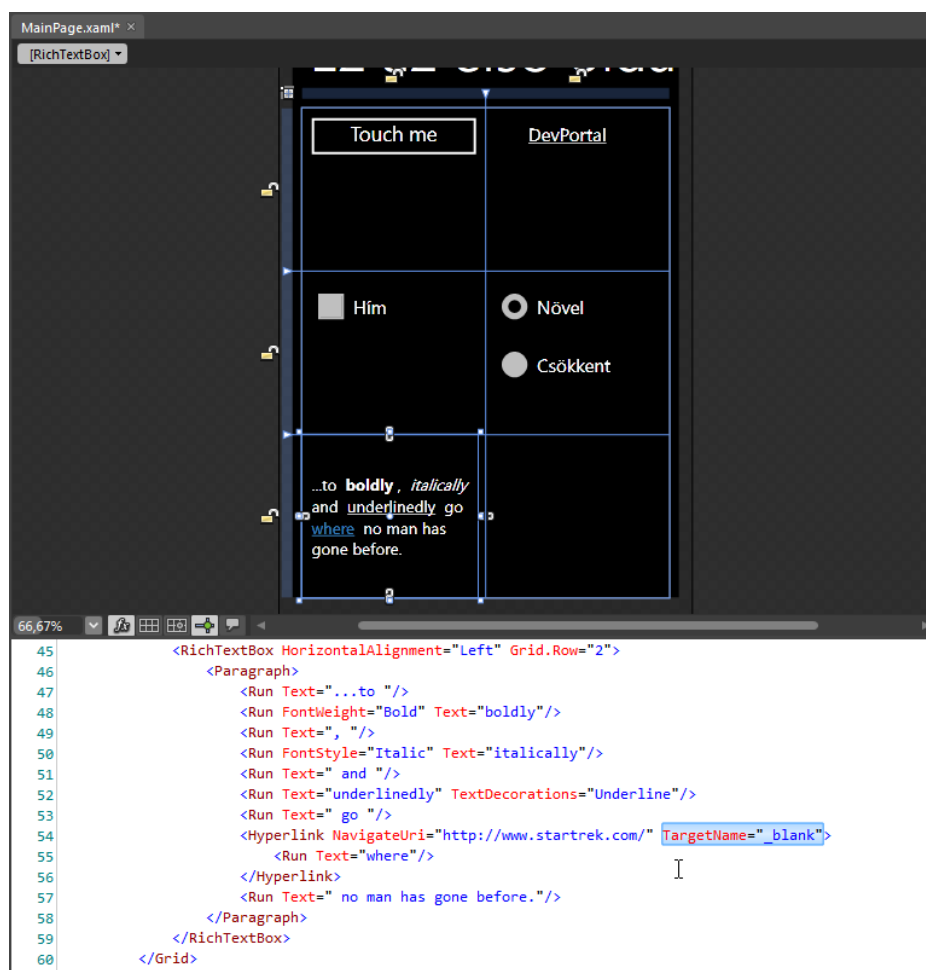
3-27 ábra – A Create Hyperlink gomb és ablak

Ahogy korábban a **HyperlinkButton** esetén, itt is meg kell adnunk a **TargetName** tulajdonságot, hogy a hivatkozás tudja, milyen linket kell megnyitni. Sajnos erre a tervezőfelületen belül nincs lehetőség, kézzel kell beleírni a XAML-be. A tervezőfelület jobb felső sarkában találjuk a nézetváltó ikonokat (3-28 ábra), a legalsóval válthatunk osztott nézetbe, ahol egyszerre látjuk a tervezőfelületet, és a XAML-t.



3-28 ábra: A tervezőfelület jobb felső sarkában található nézetváltó gombok

A XAML-ben meg kell keresni a **Hyperlink** elemet, és **TargetName** tulajdonságának a „\_blank” értéket kell adni. Ha ezek után fut az alkalmazás, a link már jól fog működni – ha megérintik, megnyílik az Internet Explorerben a megadott weblap.



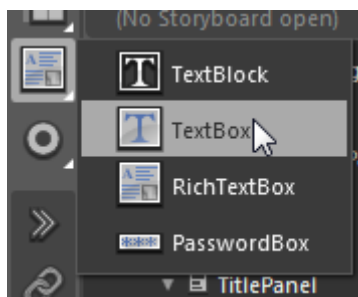
3-29 ábra – Osztott nézetben egyszerre látjuk a tervezőfelületet és a hozzá tartozó XAML-t

## Szövegbeviteli vezérlők

Szövegeket kiírni már tudunk, és a felhasználó különböző parancsokat futtathat, vagy éppen választhat lehetőségek közül. De gyorsan szembe kerülünk azzal az igénnyel, hogy ne csak a felkínált lehetőségek közül választhasson, hanem ő maga adhasson meg valamilyen bemenetet, amit a program fel tud dolgozni. A Silverlight számos speciálisabb vezérlőt kínál a fejlesztőknek, hogy megkönnyíthessék a felhasználói adatbevitelt, de legáltalánosabb adatbeviteli vezérlő mindig is a **TextBox** marad.

### Szövegbevitel a TextBoxszal

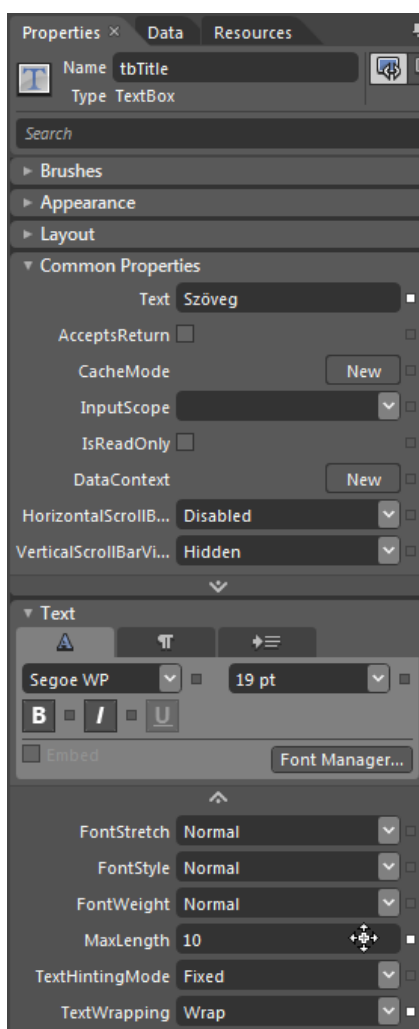
A **TextBox**, avagy szövegdoboz segítségével a felhasználó a telefon virtuális vagy fizikai billentyűzete segítségével adatokat vihet be az alkalmazásba. A TextBoxot, ahogy a két előző vezérlőnk is a szövegvezérlők csoportjában találjuk a Toolboxon (3-30 ábra).



3-30 ábra – A TextBox helye a szövegvezérlők között, a Toolboxon

Nem meglepő módon a **TextBox** vezérlő **Text** tulajdonságán keresztül érhetjük el, illetve kódból itt állíthatjuk be a szövegdoboz szövegét. Az **IsReadOnly** tulajdonságával tehetjük csak olvashatóvá a szövegdobozt – ilyenkor a felhasználó nem állíthatja át a beleírt szöveget.

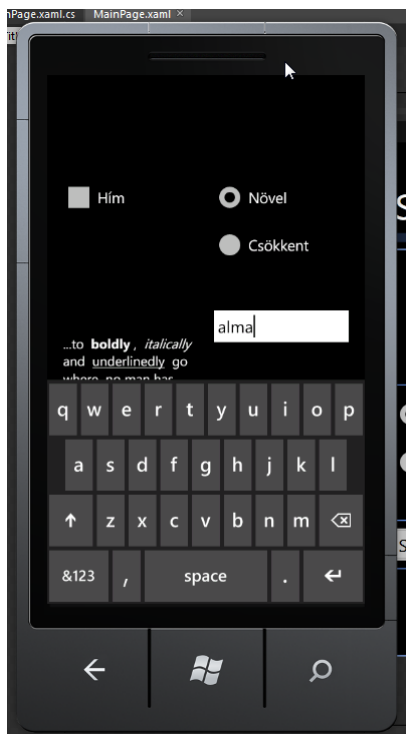
A **MaxLength** tulajdonsággal megszabhatjuk, hogy legfeljebb hány karaktert írhatson be a felhasználó a szövegdobozba. Programozottan ennél többet is meg lehet adni; a **MaxLength** csak a felhasználói adatbevitelt szabályozza.



3-31 ábra A TextBox tulajdonságainak beállítása a Properties panelen

Amikor a **TextBox** szövege megváltozik, elkaphatjuk a **TextChanged** eseményt. Az alábbi kódrészlet segítségével a felhasználó írhatja be az oldal címét – a korábban meghatározott **MaxLength** miatt legfeljebb 10 karakterben. Ahogy gépel, az oldal címe automatikusan frissül.

```
private void tbTitle_TextChanged(object sender, TextChangedEventArgs e)
{
    PageTitle.Text = tbTitle.Text;
}
```



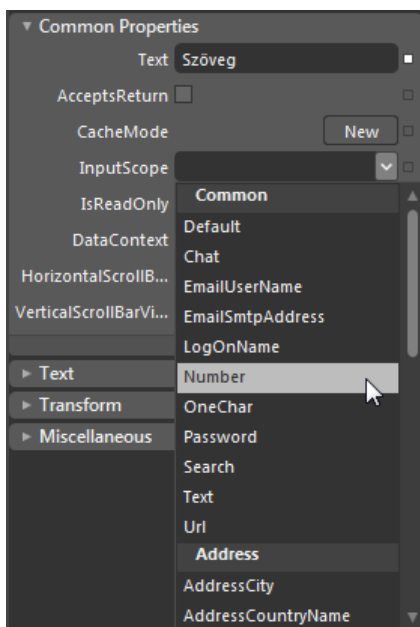
**3-32 ábra: Adatbevitel a TextBoxon keresztül**

#### ***A virtuális billentyűzet beállítása***

Egy néhány colos billentyűzeten gépelni sosem lesz annyira kényelmes és gyors, mint egy teljes értékű és méretű fizikai billentyűzeten tenni ugyanezt. Ha például tudjuk, hogy a felhasználó egy szövegdobozba csak számokat fog gépelni, miért nehezsük meg a dolgát egy teljes billentyűzettel? Elég lenne a számbillentyűket megjeleníteni!

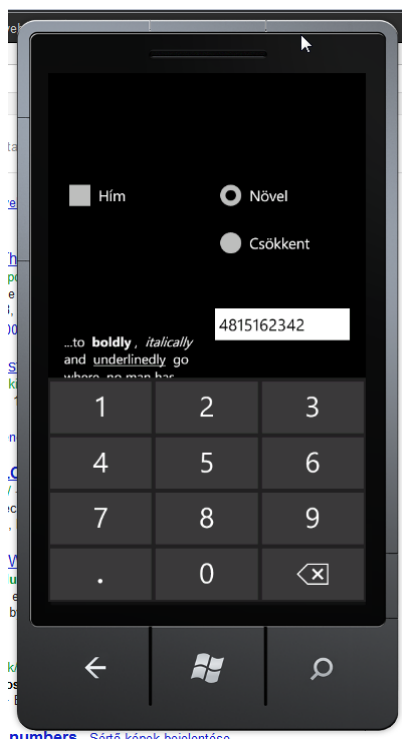
Éppen ezért, illetve, hogy esetleg megakadályozzuk a felhasználót érvénytelen adatok bevitelében, a Silverlight for Windows Phone 7 lehetővé teszi, hogy néhány kattintással pontosan beállíthassuk a **TextBox** vezérlők virtuális billentyűzetét. Ezt az **InputScope** tulajdonsággal tehetjük meg (3-33 ábra).





**3-33 ábra: Az InputScope kiválasztása egy TextBoxnál**

Ha például a **Number** beállítást választjuk, a felhasználó csak számokat tud majd beírni ebbe a szövegdobozba (3-34 ábra).



**3-34 ábra: Number InputScope-úra beállított TextBox**

Ugyanezt természetesen kódból is beállíthatjuk, így akár futás közben is változtatható egy TextBoxhoz rendelt billentyűzet:

```
InputScope ins = new InputScope();
InputScopeName insName = new InputScopeName();
insName.NameValue= InputScopeNameValue.Number;
ins.Names.Add(insName);
tbTitle.InputScope = ins;
```

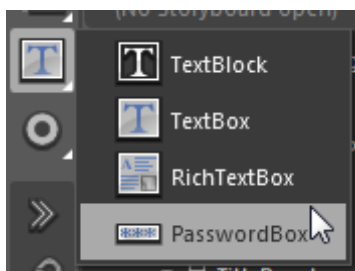
Az InputScope rengeteg lehetséges értéket felvehet. Az alábbi linken mindegyikről találhatunk egy rövid leírást: [http://msdn.microsoft.com/en-us/library/system.windows.input.inputscopenamevalue\(v=vs.95\).aspx](http://msdn.microsoft.com/en-us/library/system.windows.input.inputscopenamevalue(v=vs.95).aspx)

#### ***Jelszavak bevitele a PasswordBoxon keresztül***

Ha alkalmazásunk vagy egy hozzá kapcsolódó webes szolgáltatás jelszavas azonosítást igényel, legalább egyszer a jelszót is be kell kérni a felhasználótól. Ez történhet egy normál **TextBox**on keresztül is, azonban ennek az a hátránya, hogy bárki, aki éppen a felhasználó mögött áll, lelesheti a jelszót a kijelzőről. (Azt most ne firtassuk, hogy ehhez olyan közel kell állnia, hogy elkerülhetetlenül a felhasználó fülébe szuszog!)

Ilyen esetekre a megfelelő megoldás a **PasswordBox** használata, mely egy speciális **TextBox**. Miután a felhasználó beírt egy karaktert, az még egy másodpercig látszik, aztán egy pöttyre cseréli le a **PasswordBox**, ezzel megakadályozva, hogy a jelszó utólag visszaolvasható legyen.

A **PasswordBox** is a szövegvezérlők csoportban található (3-35 ábra).



**3-35 ábra: A PasswordBox kiválasztása a Toolboxról**

A **Password** tulajdonságon keresztül érhető el a beírt jelszó, a **PasswordChar** tulajdonsággal pedig az alapértelmezett pöttyöt cserélhetjük le egy nekünk tetsző helyettesítő karakterre.

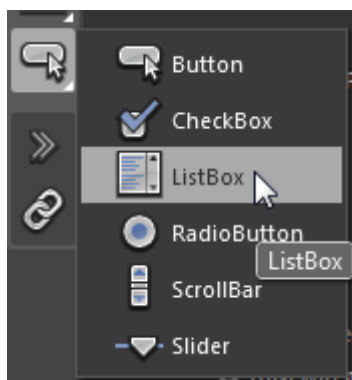
#### ***Listavezérlők***

A vezérlők, amelyekkel eddig foglalkoztunk, egyetlen tartalmat tudtak megjeleníteni. Bizonyos vezérlőknél ez a **Content** tulajdonságon keresztül megadott objektum, más vezérlőknél pedig a **Text** vagy **Password** tulajdonságban megadott egyszerű szöveg.

Mi a helyzet, ha nem egy elemet (szöveget, saját objektumot, bármit) szeretnénk megjeleníteni, hanem több azonos típusú elemet? Erre a problémára szolgáltatnak megoldást a listavezérlők. Ezekben közös, hogy rendelkeznek egy **Items** nevű tulajdonsággal, mely objektumok listája. Ezen keresztül adhatunk hozzá elemeket a listavezérlőhöz, mely megpróbálja azokat legjobb tudása szerint megjeleníteni. A megjelenítés testreszabható, erre a későbbiekben még kitérünk.

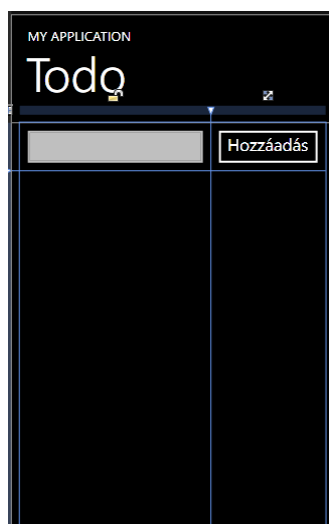
#### ***A ListBox használata***

A **ListBox** talán a legalapvetőbb elemvezérlő; a Toolboxon az általános vezérlők között találjuk (3-36 ábra).



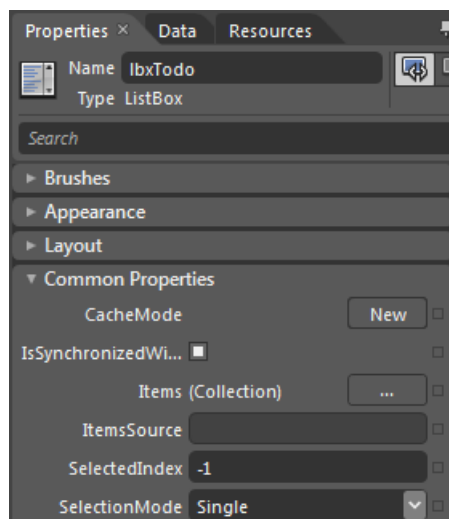
**3-36 ábra: A ListBox kiválasztása a Toolboxról**

A működés megértéséhez a 3-37 ábrán látható, végtelenig egyszerűsített tennivaló-listát valósítjuk meg. Mindössze annyit tud, hogy a **TextBox**ba írt szövegeket a gomb megnyomásával a **ListBox**hoz rendeli. (Az elemek törlése már feláras funkció.)



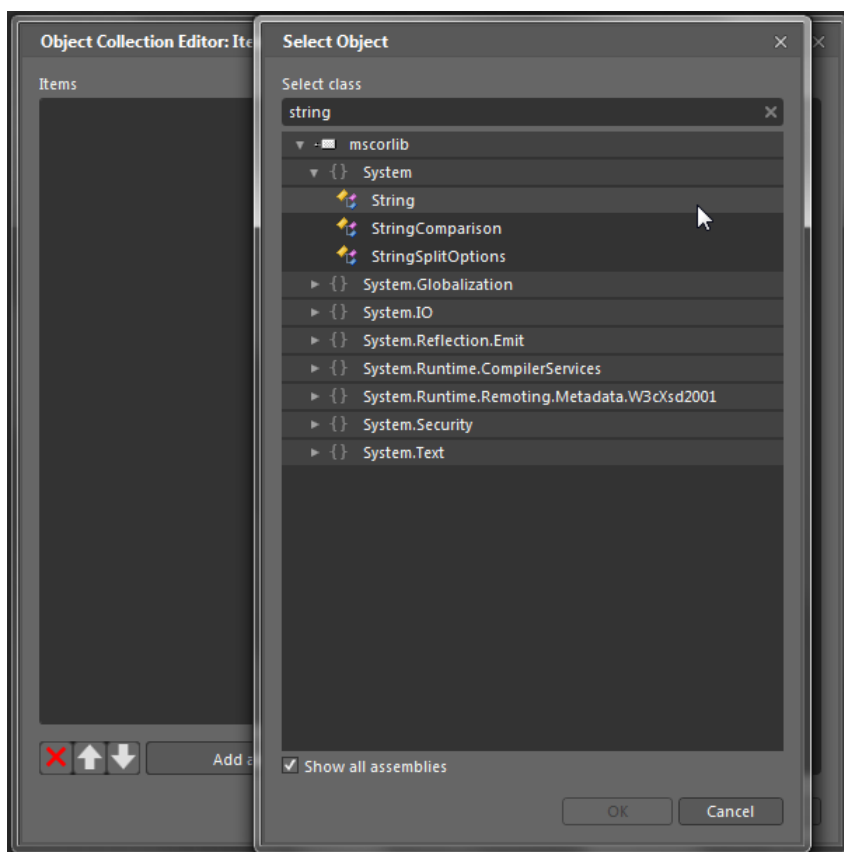
**3-37 ábra: A Todo-lista program kinézete**

A **ListBox** kijelölése után a Properties panelen szétnézve ismét a Common Properties csoport rejti a leggyakrabban használt tulajdonságokat. Az **Items** listán keresztül adhatunk elemeket a **ListBox**hoz, a **SelectionMode**-dal pedig megadhatjuk, hogy több elem is kijelölhető legyen-e. A **SelectedIndex** segítségével állíthatjuk be, hogy hányadik elem legyen alapértelmezetten kijelölve. A számozás 0-tól indul, a -1 azt jelenti, nincs kijelölt elem. (3-38 ábra)



**3-38 ábra: A ListBox tulajdonságainak beállítása a Properties panelen**

Ha az Items melletti „...” gombra kattintunk, az Object Collection Editor ablak jelenik meg, melyben szerkeszthetjük a **ListBox** tartalmát. Az „Add another item” gombra kattintva kiválaszthatjuk a **ListBox**hoz hozzáadni kívánt elem típusát. Egy stringet hozzáadva (az mscorlib|Systemben találjuk) a szerkesztőablak jobb oldalán megjelenik a karakterlánc, amit átírhatunk. Az egyes elemek sorrendjét az ablak alján lévő nyilakkal változtathatjuk meg, illetve az „X” gombbal törölhetünk egy elemet (3-39 ábra).



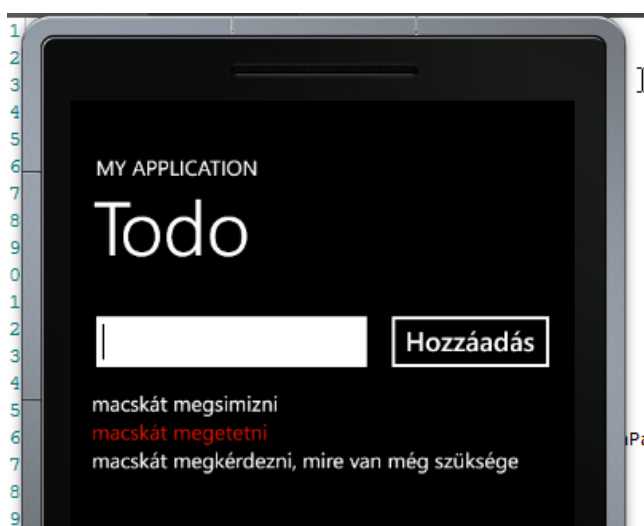
**3-39 ábra: Stringek hozzáadása a ListBox tartalmához**

A gomb **Click** eseményére felírt eseménykezelővel hozzáadhatjuk a felhasználó által megadott új elemet:

```
private void btnAddItem_Click(object sender, RoutedEventArgs e)
{
    if (!string.IsNullOrEmpty(tbNewItem.Text))
        lbxTodo.Items.Add(tbNewItem.Text);
    tbNewItem.Text = string.Empty;
}
```

A **ListBox SelectionChanged** eseménye akkor következik be, amikor megváltozik a kijelölt elem. A **SelectedItem** tulajdonságon keresztül érjük el a kiválasztott elemet (ha több is van, a **SelectedItem**sen keresztül kapjuk meg mindet). Az alábbi kód egy **MessageBox**ban jeleníti meg a felhasználó által kiválasztott elemet:

```
private void lbxTodo_SelectionChanged(object sender, SelectionChangedEventArgs e)
{
    MessageBox.Show(lbxTodo.SelectedItem.ToString());
}
```



**3-40 ábra: Az alkalmazás működés közben**

Ahogy a 3-40 ábrán is látható, a kijelölt elem a telefonon beállított „accent” színnel jelenik meg.

## Adatkötés

Az előző példákban szerepeltek olyan kódrészletek, amelyek pusztán arra szolgáltak, hogy frissítsék a felületet, megjelenítsék rajta az új, megváltozott adatokat. A gomb megnyomásának hatására lefutó eseménykezelő nemcsak megnövelt egy értéket, hanem utána explicit módon ki is írta a számot a gombra. A **TextBox**ba írt szöveg megváltozásakor lefutó **TextChanged** esemény minden karakter leütése után frissítette az oldal címét. Ezen kívül pedig ott volt még a **ListBox**, amelybe közvetlenül helyeztük bele az elemeket, holott, ha később szeretnénk volna feldolgozni azokat, érdemesebb lett volna egy háttérben lévő, típusos listába rakni őket, és ennek a listának a tartalmát megjeleníteni.

A XAML-ben mélyen be van ágyazva az adatkötés technológiája. Az adatkötés segítségével különböző objektumokat, vezérlőket tarthatunk automatikusan szinkronban. Ha valami megváltozik az egyik helyen, a változás azonnal megjelenik a másik helyen is, anélkül, hogy ehhez bármilyen kódot kellene írunk.

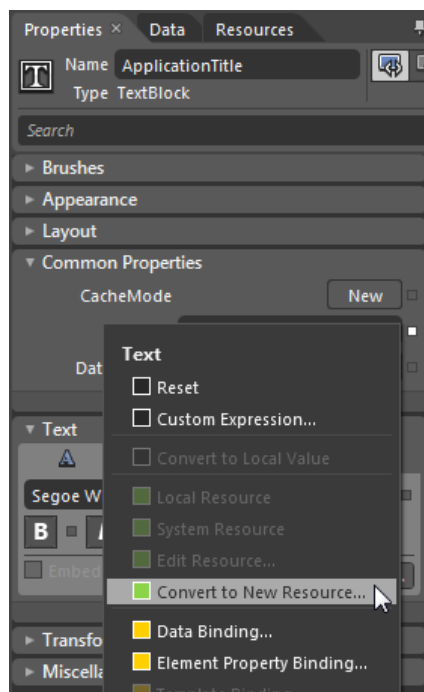
## Erőforrások

Vannak olyan értékek (számok, szövegek), objektumok, amelyeket nem csak egy helyen szeretnénk felhasználni egy programban. Érdemes lenne tehát egyszer létrehozni őket, és aztán az adott oldalon vagy az egész programban újra és újra felhasználni. Ez egyrészt a memóriefoglalást is csökkenti, másrészt pedig szinkronban tartja az összes olyan objektumot, mely felhasználja az erőforrást. Könnyebb lesz

megtartani a program konzisztenciáját, ha a változtatás igénye esetén csak egy helyen kell frissíteni egy adott objektumot.

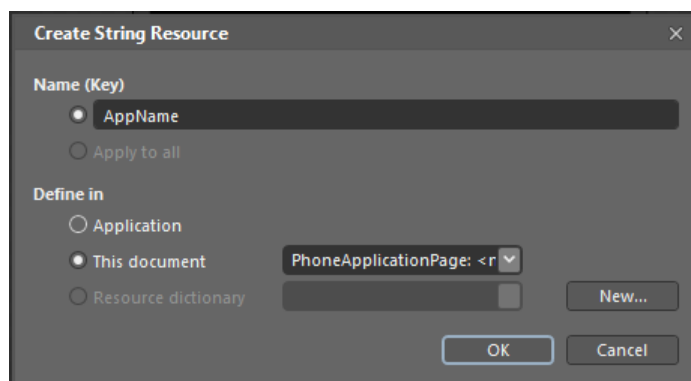
Ezeket a közös használatú objektumokat definiálhatjuk kódban, de tekintve, hogy gyakran magához a felülethez köthetők (például egy-egy vezérlő háttérszínét meghatározó Brush), XAML-ben is megadhatjuk őket. Így még jobban elválasztható a felület leírása a kódban megírt logikától.

Alkalmazásaink címe például nem változik, és ha több ablakból áll egy alkalmazás, valószínűleg mindegyik ablakban szeretnénk majd felhasználni azt. Az **ApplicationTitle** **TextBlock**ot a Properties panelen kiválasztva a Text tulajdonság mellett található Advanced options gombra kattintva megjelenik a menü, melynek egyik eleme a „Convert to New Resource...” névre hallgat (3-41 ábra).



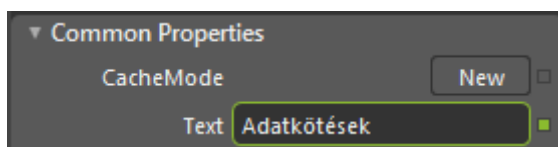
3-41 ábra: Erőforrás létrehozása egy meglévő értékből

Erre kattintva alakíthatjuk egy újrafelhasználható erőforrássá azt a stringet, amit éppen a Text tulajdonság tartalmaz. A megjelenő párbeszédablakban beállíthatjuk az erőforrás nevét (Name), illetve meghatározhatjuk, hogy hol kerüljön definiálásra az erőforrás: az oldalon vagy az alkalmazás szintjén. Ez meghatározza azt is, hogy hol lesz látható: előbbi esetben csak az aktuális oldalon belül, utóbbinál viszont ha vannak más oldalaink, azokban is elérhető lesz. A harmadik lehetőség, hogy egy erőforráskönyvtárban (*resource dictionary*) hozzuk létre. Az erőforráskönyvtárak különálló XAML-fájlok, melyeket csak erőforrások tárolására szokás létrehozni.



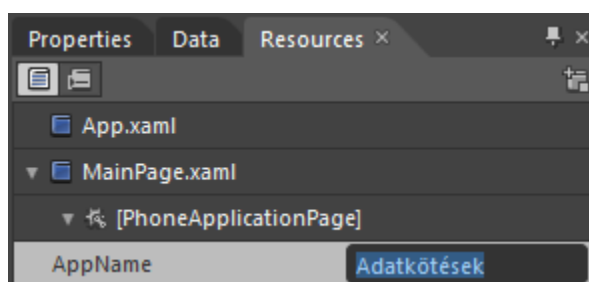
3-42 ábra: Az erőforrás nevének és tárolási helyének beállítása

Az is észrevehető, hogy a Blend nagyon szépen jelzi, hogy egy tulajdonság nem lokálisan kap értéket, hanem egy erőforráshoz kötöttük hozzá (3-43 ábra).



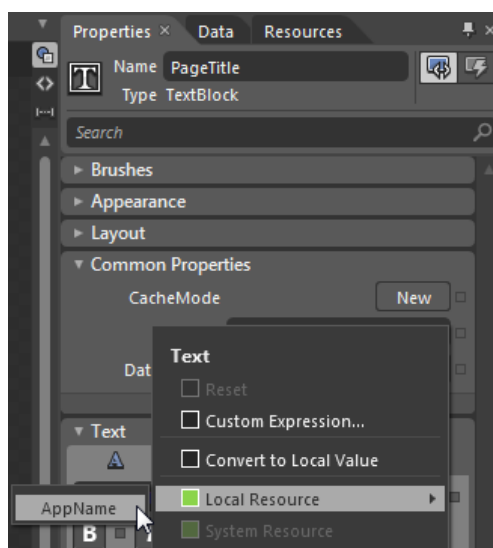
**3-43 ábra: A zöld keret és pötty jelzi, hogy a tulajdonság egy erőforrásból kap értéket**

Továbbá, ha átváltunk a Resources panelra, ott is megtalálható az erőforrások között az újonnan létrehozott string. A kész erőforrás innen azonnal, drag-and-drop módszerrel áthúzható egy felületi elemre, létrehozva ezzel az erőforráshoz kötést (3-44 ábra).



**3-44 ábra: Az adatforrások a Resources panelon**

Ugyanezt elérhetjük, ha a kiválasztott tulajdonság melletti Advanced optionsre kattintva a Local Resource menüpont alatt kiválasztjuk a megfelelő erőforrást (3-45 ábra).



**3-45 ábra: Már létező erőforrás kiválasztása egy tulajdonsághoz**

Ez egy nagyon hasznos lehetőség a XAML-ben, illetve a Silverlightban. Mi a helyzet akkor, ha nem egy előre elkészíthető erőforráshoz akarunk kötni, hanem valami máshoz, például egy CLR-objektumhoz, vagy kötést szeretnénk létrehozni a UI különböző elemei között? Erre szolgál megoldással az adatkötési rendszer.

## Több vezérlő kötése egyazon forráshoz

Tekintsük az alábbi osztályt:

```
public class Character
{
    public string Name {get; set;}
    public string City {get; set;}

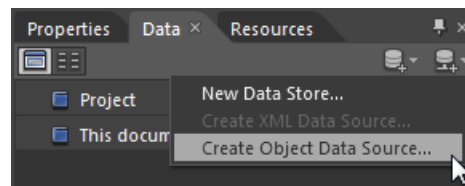
    public Character()
    {
        Name = "Hubert Farnsworth";
        City = "New New York";
    }
}
```

Hozzunk létre egy példányt ebből az oldalon, és jelenítsük meg mindkét tulajdonságát egy-egy **TextBlock**-ban (3-46 ábra)!



3-46 ábra: A program felülete

Ehhez kattintsunk a Data panelre, majd annak jobb felső sarkában a „Create data source” gomb alatt válasszuk a „Create Object Data Source...” pontot (3-47 ábra)!

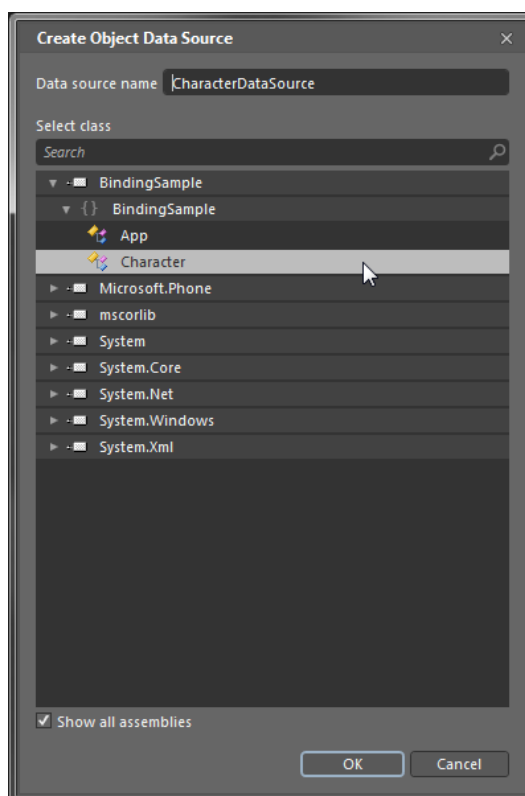


3-47 ábra: Adatforrás létrehozása a Data panelen

A Data panel tetején találunk egy másik gombot is, a „Create sample data” nevűt. Ezzel véletlenszerű mintaadatokat feltöltött adatforrásokat hozhatunk létre. Hasznos a fejlesztésnek abban a fázisában, ahol a tényleges adatok még nem állnak rendelkezésre.

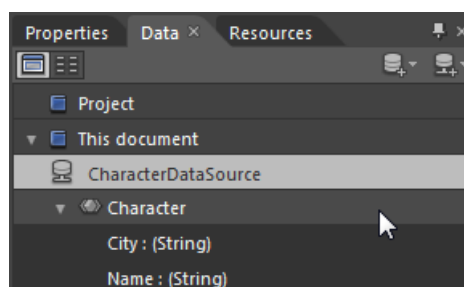
A megjelenő ablakban válasszuk ki a **Character** osztályt, illetve adjuk meg az adatforrás nevét! (Ha még nem jelenne meg az osztály, vagy hiányzik a program szerelvénye a listából, fordítani vagy futtatni kell, és megjelenik.)



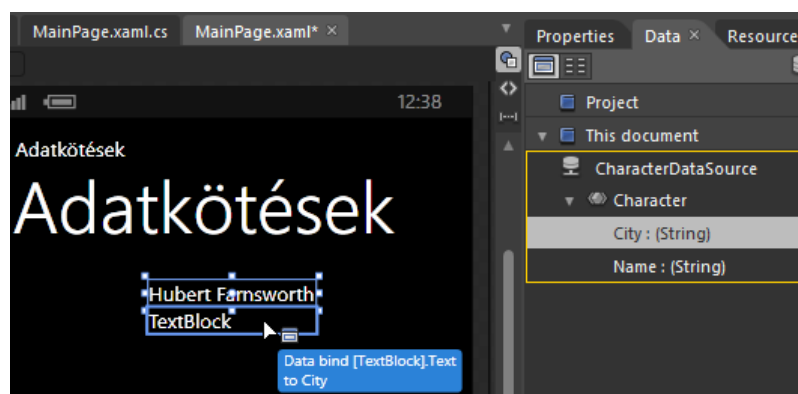


**3-48 ábra: Az adatforrás típusának kiválasztása**

A Data panelen megjelenik az új adatforrás (3-49 ábra). Drag-and-drop módszerrel az egész adatforrást vagy egyes részeit ráhúzhatjuk egy felületi elemre, és létrejön az adatkötés (3-50 ábra).



**3-49 ábra: Az új adatforrás tulajdonságai a Data panelen**



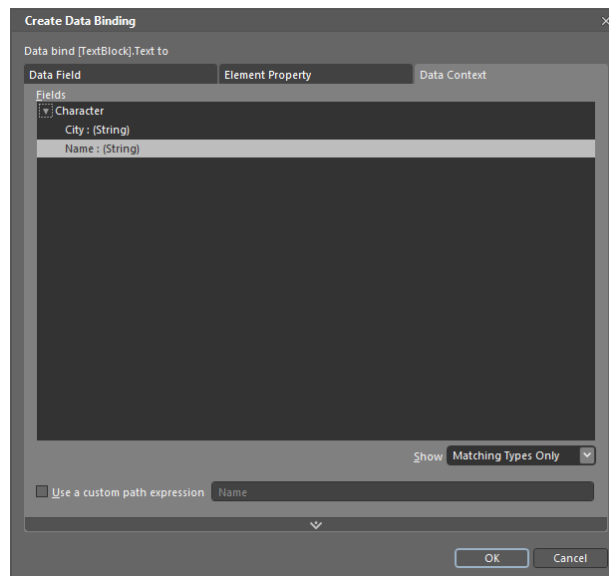
**3-50 ábra: Az adatforrás részeinek adatkötése drag-and-drop módszerrel**

Szép, szép, de ha sok tulajdonságot szeretnénk egyazon objektumról megjeleníteni, nem lehetne egy kicsit központosítani az adatkötést? Természetesen lehet. Vezérlőnk **DataContext** tulajdonsága szolgál

arra, hogy egy objektumot megadva neki beállítsuk, hogy a vezérlő vagy a benne lévő vezérlők tulajdonságai alapértelmezetten milyen objektumra legyenek rákötve. A **DataContext** öröklődik lefelé a vizuális fán, tehát ha például a **PhoneApplicationPage**-en állítjuk be, automatikusan mindegyik, az oldalon lévő vezérlő képes lesz adatot kötni hozzá.

A fenti két **TextBlock** egy **StackPanel** gyermeke. Logikus tehát, hogy a **StackPanel DataContext**-jét állítsuk be a **CharacterDataSource**-ra, azaz az előbb létrehozott **Character** objektumra, a két **TextBlock** objektumot pedig ennek egyes tulajdonságaihoz kössük hozzá. Ehhez első lépésként magát az adatforrást kell megfogni, és drag-and-drop módszerrel a **StackPanel**-re húzni. (Érdemes nem a felületen, hanem az Objects and Timeline ablakon elengedni a vontatott objektumot.) Ezek után az egyik **TextBlock Text** tulajdonsága melletti Advanced options-re kattintva a DataBinding menüpontot kell választani.

A megjelenő ablak automatikusan a **DataContext** fület nyitja meg, ahol kiválaszthatjuk, hogy a **StackPanel**-től örökölt kontextus melyik tulajdonságára szeretnénk a **Text**-et kötni (3-51 ábra).



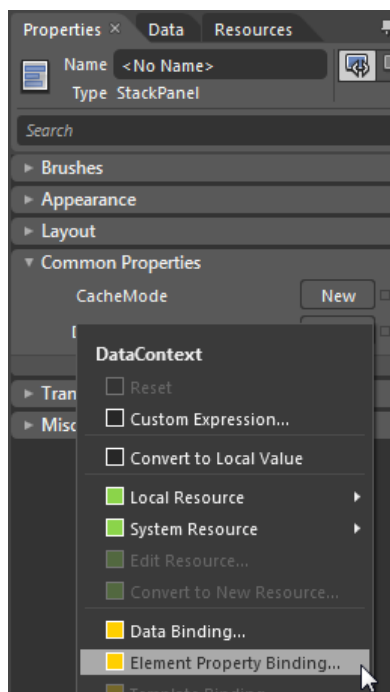
3-51 ábra: Adatkötés létrehozása egy szülő DataContextjén keresztül

Tekintve, hogy Blendben az ember hajlamos mindent „csak összekattintgatni”, itt még nem olyan nagy a **DataContext** előnye a sok, egyenként létrehozott **Binding**gal szemben, de ha kézzel szerkesztjük a XAML-t, felbecsülhetetlen lehet, hogy nem kell egy-egy bonyolultabb adatkötést újra és újra megírni, hanem elég egyszer létrehozni, majd jóval egyszerűbb szintaxissal a felület egyes részeihez kötni.

### Adatkötés UI-elemek között

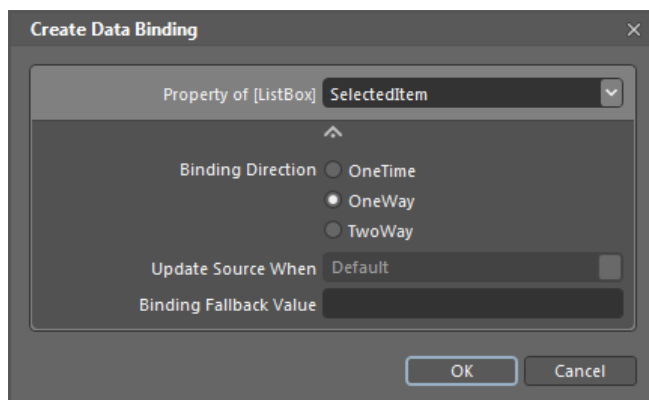
Az előző példát egészítjük ki! A **TextBlock**ok alatt egy **ListBox** foglal helyet, melyben **Character** típusú elemeket helyezünk el. Szeretnénk, ha a **TextBlock**okban a **ListBox** kiválasztott eleme jelenne meg. Tehát arra van szükség, hogy a **StackPanel DataContext**-jét most ne egy háttérben lévő objektumhoz kössük, hanem a **ListBox** kiválasztott eleméhez; vagyis az adatkötést két UI-elem között hozzuk létre.

A Blend ehhez is hathatós támogatást nyújt. Elég a **StackPanel DataContext**-jénél az Advanced options alatti „Element Property Binding...” pontot választani, és a **ListBox**-ra kattintani (3-52 ábra).



**3-52 ábra: Adatkötés egy vezérlőre**

A megjelenő ablakban kiválaszthatjuk a tulajdonságot, amelyre kötni szeretnénk (**SelectedItem**), valamint megadhatunk különböző beállításokat, például a kötés irányát. Silverlightban három kötési mód van: a **OneTime** egy egyszeri kötés, vagyis az adatok első, a forrásból történő kiolvasása után nem lesz szinkronizálás; a **OneWay**, ami egyirányú kötés, vagyis a forrás változása kihat a célra; és a **TwoWay**, ahol a változáskövetés kétirányú, tehát a cél változása is visszahat a forrásra. Mivel csak megjeleníteni szeretnénk a szöveget, tökéletesen elég az alapértelmezett **OneWay** kötés (3-53 ábra).



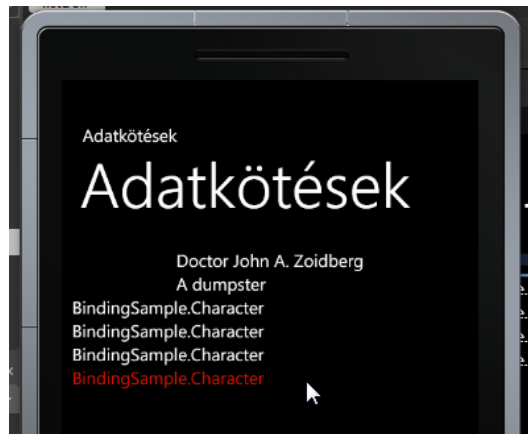
**3-53 ábra: Tulajdonság és irány kiválasztása adatkötésnél**

Ezután már csak a **TextBlock**ok **Text** tulajdonságát kell a korábbiakban megismert módon kötni. Mivel azonban a **ListBox**ban lévő elemekről statikusan semmit nem tudunk, a Create Data Binding ablakban nem kapunk segítséget, hogy kiválasszuk a **Name** vagy a **City** tulajdonságokat. Az ablak alsó részén található szövegdobozban adhatjuk meg manuálisan a tulajdonság nevét (3-54 ábra).



**3-54 ábra: Saját elérési út megadása a forrástulajdonságra**

A Blend és a Visual Studio XAML értelmezője olyannyira kifinomult, hogy ha **Name** tulajdonságot lát, nem enged szünet karaktert belerakni. Ha azt jelzi az IDE, hogy egy XAML-ben megadott string nem megfelelő a **Name** tulajdonságnak, vagy ne rakjunk bele szünetet, vagy változtassuk meg a **Name**-et másra! Emellett írhatunk morcos leveleket a Microsoftnak.



3-55 ábra: A **ListBox** egy elemének kiválasztásával a két **TextBlock** tartalma frissül

Ha teszteljük az alkalmazást, a két **TextBlock** felirata automatikusan frissül, ahogy más elemre állunk a **ListBox**-ban (3-55 ábra).

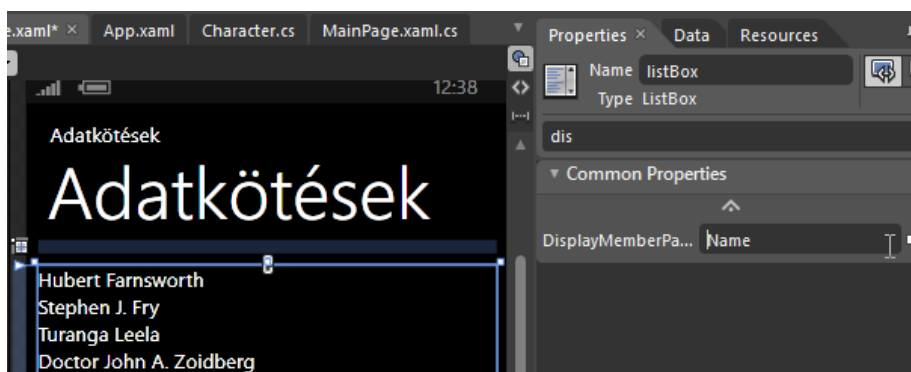
Az adatkötési keretrendszernek alapvető része a változásokról való értesítés. Ezt a vezérlők esetében ún. **DependencyProperty**-ekkel szokás megoldani, az egyéb típusoknál pedig az **INotifyPropertyChanged** interfész implementálásával. Utóbbiról egy leírás és példa: <http://msdn.microsoft.com/en-us/library/ms229614.aspx>

## Az adatok megjelenésének testreszabása adatsablonokkal

Amikor szövegeket helyeztünk el a **ListBox**-ban, azok gyönyörűen megjelentek. Most, hogy a szövegnél bonyolultabb objektumokat (a **Character** osztály már két stringet is tartalmaz), már nem működik annyira szépen a megjelenítés.

Ez nem meglepő: honnan tudná a Silverlight, hogy hogyan kell megjeleníteni egy **Character** típus példányát? Jobb híján csak annyit tehet, hogy visszatér a kályhához: meghívja a számára ismeretlen típusú objektum **ToString** metódusát. Ennek eredményét láttuk a képernyőn. Hogyan tehetjük ezt szebbé?

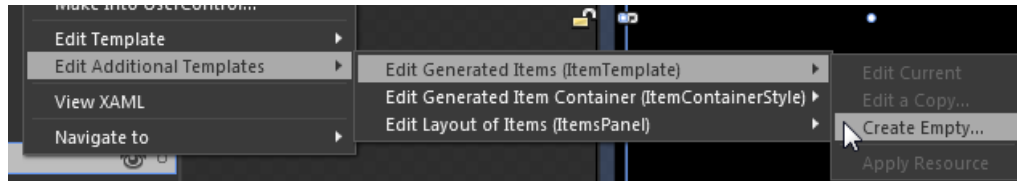
Listavezérlőink rendelkeznek egy **DisplayMemberPath** nevű tulajdonsággal, mellyel megadhatjuk, hogy az objektum melyik tulajdonsága jelenjen meg a **ToString** hívása helyett (3-56 ábra).



3-56 ábra: A **DisplayMemberPath** tulajdonság beállítása

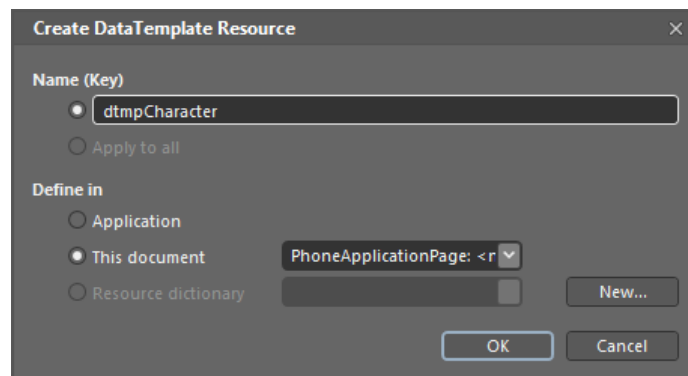
Mivel a mobilképernyő alapvetően kicsi, gyakran elég lehet, ha egyetlen tulajdonságot jelenítünk meg. Sokszor viszont több adatot is szeretnénk megjeleníteni egy elemről. Ilyenkor használhatunk adatsablonokat (*data template*).

A **ListBox**ban lévő elemek sablonját úgy változtathatjuk meg, hogy jobb gombbal a **ListBox**ra kattintunk, majd az „Edit Additional Templates” menüpont alatt az „Edit Generated Items (ItemTemplate)” parancsot választjuk. Ezen belül elsőre csak a „Create Empty...” választható, ezzel hozhatunk létre új adatsablont (3-57 ábra).

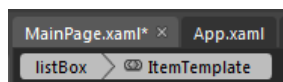


**3-57 ábra: Adatsablon létrehozása a ListBox elemeihez**

Ha a következő ablakban (3-58 ábra) megadjuk a sablon nevét, és azt, hol jöjjön az létre (az erőforrások létrehozásából már ismerős lehet), a tervezőfelület sablonszerkesztési módba vált (3-59 ábra).

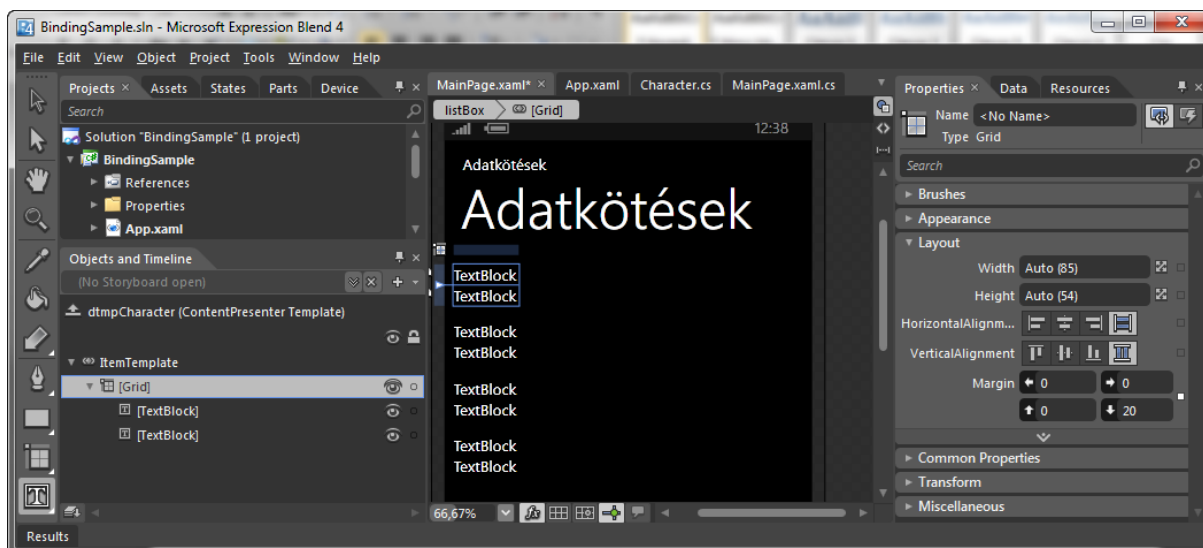


**3-58 ábra – Az új adatsablon nevének és helyének megadása**



**3-59 ábra – A tervezőfelület tetején látható „breadcrumb” jelzi, hogy sablont szerkesztünk**

Itt ugyanúgy rakhatjuk össze a felületet, mint ahogy az oldalakét is tettük.



3-60 ábra: A Gridből és az abban lévő két TextBlockból álló adatsablon

A 3-60 ábrán látható, hogy két **TextBlock** van a sablonban. A korábban megismert adatkötési eljárással ez is könnyedén megoldható. (Advanced options, Data Binding..., Data Context, custom expression.)



3-63 ábra: Az adatsablonnal megerősített program működés közben

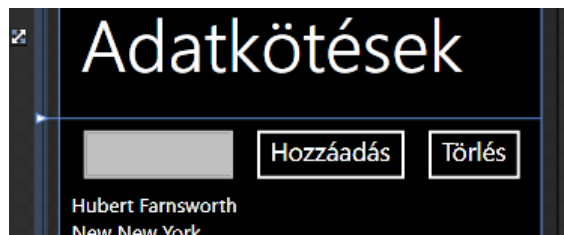
Az adatsablonok — ha erőforrásként hoztuk létre őket — újra felhasználhatóak, tehát a fent elkészítettet átadhatjuk egy másik oldal másik **ListBox** objektumának is.

## A vezérlők kinézetének testreszabása vezérlősablonokkal

Adatok megjelenését már testre tudjuk szabni. De mi a helyzet magukkal a vezérlőkkel?

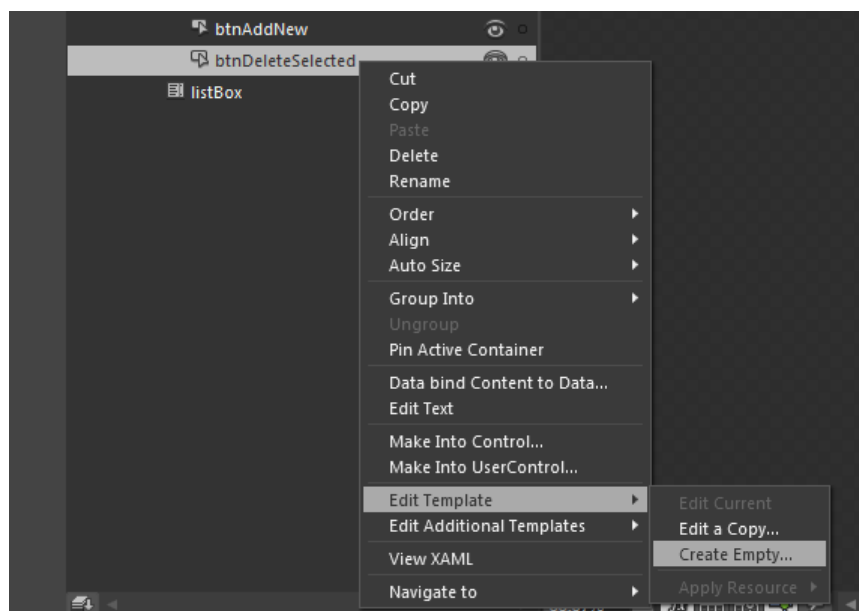
A Silverlight vezérlői jól testreszabhatóak pusztán tulajdonságaikon keresztül: elég belenézni a Brushes, Appearance és Layout csoportokba a Properties panelen. Azonban ha ez nem elég, akár teljesen ki is cserélhetjük megjelenésüket. A Silverlight (illetve eredetileg a WPF) vezérlői „kinézetmentes” (*lookless*) vezérlők; a vezérlőt leíró osztály csupán a működést definiálja, a kinézetet nem. Egy gomb mindössze, „valami, amire rá lehet kattintani” — esetünkben tapintani. Tartozik hozzá egy alapértelmezett kinézet –

legtöbbször *chrome*-ként hivatkoznak erre –, így, ha mi nem szabjuk meg, hogy hogyan nézzen ki egy gomb, ezt az alapértelmezett kinézetet kapja. Hogyan cserélhetjük le ezt az alapértelmezett megjelenést? Az előző, adatkötéses alkalmazást kiegészítettük még egy sorral, amivel új **Character**eket lehet hozzáadni a listához, illetve a lista kijelölt elemét törölni (3-62 ábra).



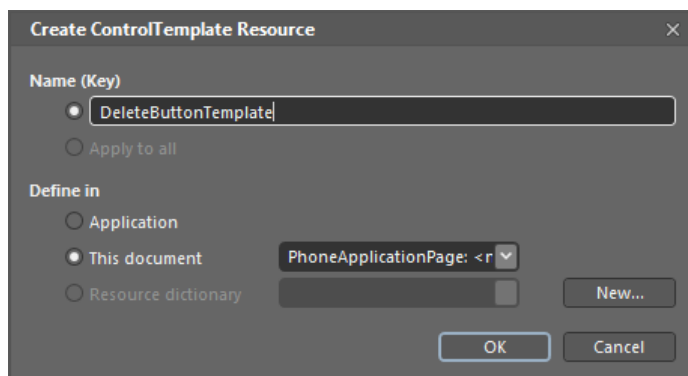
**3-62 ábra: Új vezérlők az előző példaalkalmazáshoz**

Szeretnénk, ha a gombok kisebb feliratokkal jelennének meg, és egy-egy kép is kapcsolódna hozzájuk. Ehhez tesztet kell szabnunk a vezérlőt, mert jelen pillanatban nem tud képet megjeleníteni. Jobb egérgombbal kattintva az egyik gombra, az „Edit Template” menüpontot kell választanunk. Itt vagy egy üres sablont hozhatunk létre („Create Empty...” – a példában ezt teszem), vagy a jelenlegit másolhatjuk le, és szerkeszthetjük. Ha már van egy saját sablon, akkor elérhetővé válnak az „Edit Current” (jelenlegi szerkesztése) és az „Apply Resource” (használat erőforrásként) parancsok is.



**3-63 ábra – Egy vezérlő sablonjának lecserélése üresre**

A megjelenő ablakban szokás szerint nevet adhatunk a sablonnak (erőforrásnak), és kiválaszthatjuk a helyét (3-64 ábra).

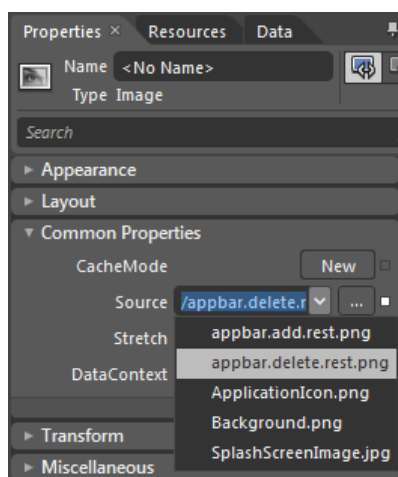


3-64 ábra – Az új vezérlősablon nevének és helyének megadása

Az üres **Grid**ben egy **Image** és egy **TextBlock** vezérlőt elhelyezve (és tulajdonságait megfelelően beállítva) az alapvető elrendezés néhány másodperc alatt összeállítható; ugyanúgy működik, mint a korábbi adatsablon létrehozása. Két dolog maradt már csak hátra:

- - kell egy kép az **Image**-nek, illetve
- - szeretnénk, ha a gomb szövege meg is jelenne.

Az első probléma egyszerűen megoldható. A projektbe fel kell vennünk a megjelenítendő képet, majd az **Image** vezérlő **Source** tulajdonságánál kiválasztani azt a lenyíló listából (3-65 ábra).

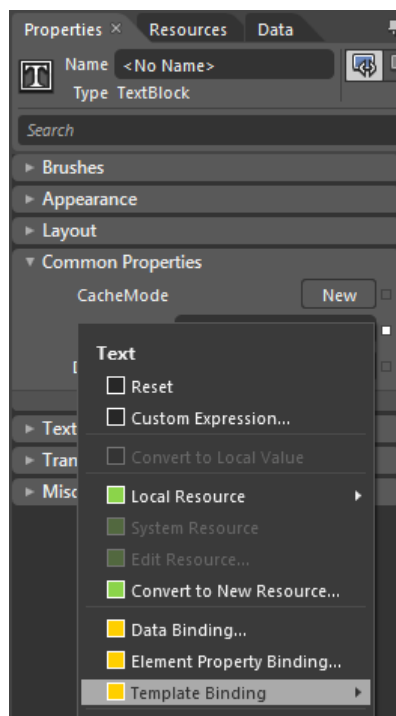


3-65 ábra: Kép kiválasztása egy **Image** elem forrásául

Ez egyszerű volt, mert előre rendelkezésre állt a kép. Mi a helyzet a felirattal? Azt esetleg változtatni szeretnénk, vagy többnyelvűvé tenni, esetleg egyszerűen csak nem akarjuk a vezérlősablonban rögzíteni. A **TextBlock**nek meg kell mondani, hogy **Text** tulajdonságában annak a **Button** vezérlőnek a **Content** tulajdonságát jelenítse meg, amelyre éppen alkalmazzák – elvégre egy sablont több gombra is rá lehet majd húzni. Ehhez a **TemplateBinding**ot használjuk fel.

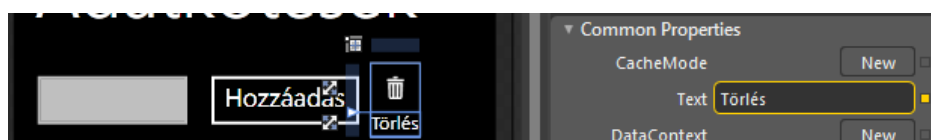
A **TextBlock** kijelölése után, a **Text** tulajdonság melletti Advanced optionsre kattintva a **Template Binding**ot kell választani. A megjelenő listában a sablon célvezérlőjének (esetünkben a **Button**) összes olyan tulajdonságát láthatjuk, melyre adatkötést lehet létrehozni (3-66 ábra).





**3-66 ábra: Vezérlősablon tulajdonságának a vezérlőhöz kötése a *TemplateBinding* segítségével**

A **Content** tulajdonság kiválasztása után meg is jelenik az adatkötést jelző sárga keret, illetve a tervezőfelületen látszik, hogy a sablon már a gombtól veszi a feliratát (3-67).



**3-67 ábra: A vezérlősablon átveszi a vezérlőtől az értéket**

Ugyanezeket a lépéseket eljátszva, a képet természetesen egy másikra állítva a Hozzáadás gomb is megszépíthető (3-68 ábra).

Érdemes a képeket Visual Studióban hozzáadni a projekthez, mert a Blendben sajnos nem tudjuk Build Action tulajdonságukat átállítani, ez pedig elengedhetetlen.



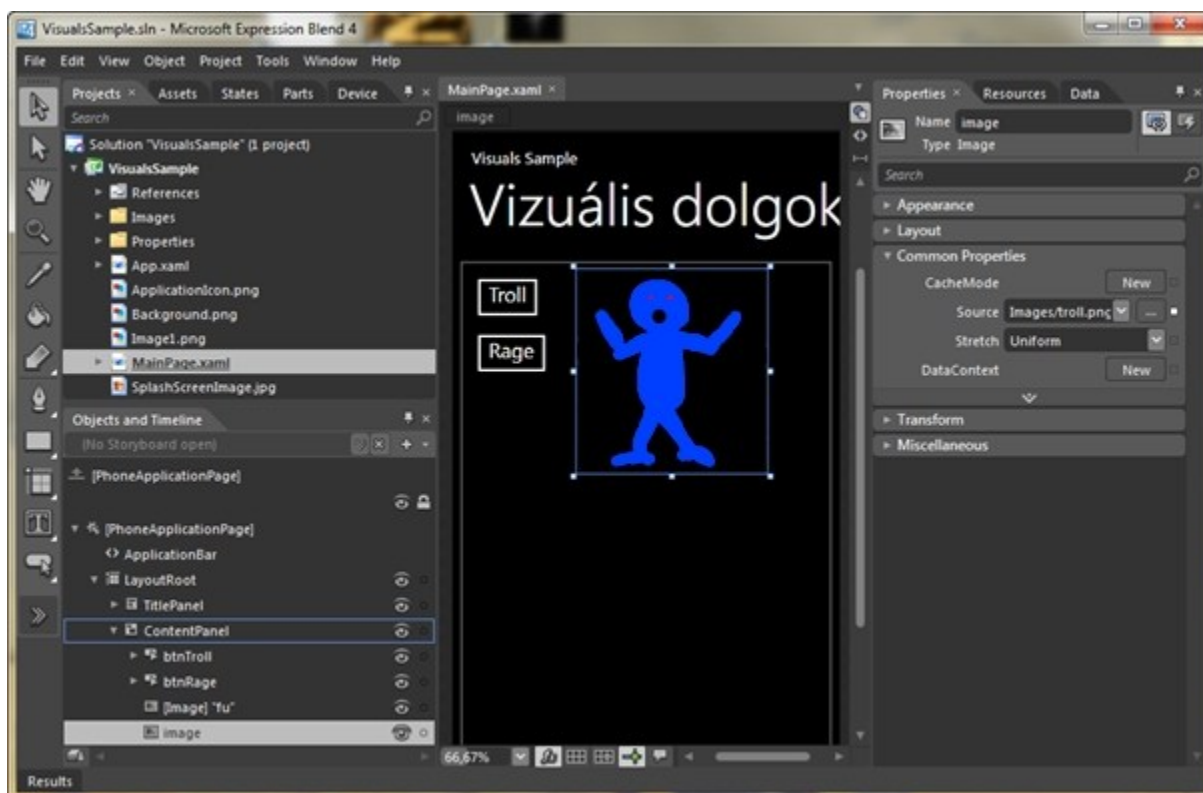
**3-68 ábra: Mindkét gomb külön sablonnal ellátva**

Mivel most a két gomb felépítése tulajdonképpen ugyanaz, csupán a képeik térnek el, érdemes saját vezérlőt készíteni, amelynek egy plusz **ImageSource** tulajdonságot adunk, és az **Image** vezérlő forrását erre kötjük rá a **TemplateBinding**gel.

## Az alkalmazás állapotainak létrehozása VisualState-ek segítségével

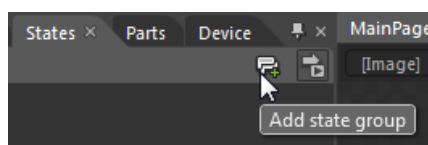
A Silverlight 3 óta alkalmazásainkban különböző vizuális állapotokat adhatunk meg, melyekben leírhatjuk a felület elvárt állapotát bizonyos helyzetekben. Ezzel központosíthatjuk a felületleírást, és az egyes állapotok közötti váltás is jelentősen leegyszerűsödik a korábbi módszerekhez képest.

Vegyük példának az alábbi egyszerű alkalmazást! Két gombból áll, amelyek két képhez kapcsolódnak olyan módon, hogy ha a felhasználó megnyomja az egyik gombot, az egyik kép tűnik elő, ha a másikat, a másik kép. Emellett pedig a megnyomott gomb letiltottá válik, a másik pedig engedélyezetté (3-69 ábra).



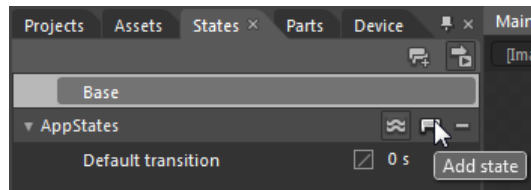
3-69 ábra: A VisualState-eket bemutató program kinézete és szerkezete

Ezt az egyszerű logikát leírhatjuk kódban is, de ha egy mód van rá, próbáljuk meg elkerülni, hogy olyan kódot írjunk, ami a felület megjelenését közvetlenül befolyásolja. Ehelyett a States panelen hozunk létre két állapotot! VisualState-jeink csoportokba rendezhetők, ezért először egy csoportot kell létrehozni (3-70 ábra).



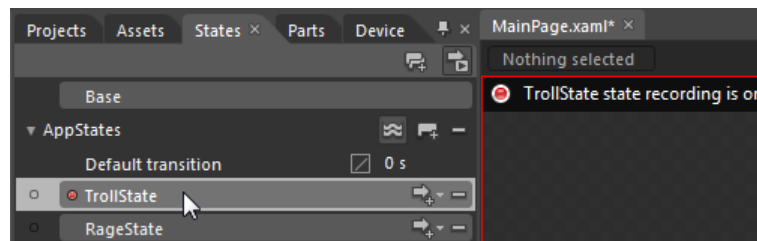
3-70 ábra: Új állapotcsoport hozzáadása a States panelen

A csoport létrehozása és elnevezése után létrehozhatjuk állapotainkat (3-71 ábra). Az egy csoportban lévő állapotok kölcsönösen kizárják egymást, tehát az alkalmazás egy **VisualStateGroup**-nak egyszerre csak egy **VisualState**-jében lehet.



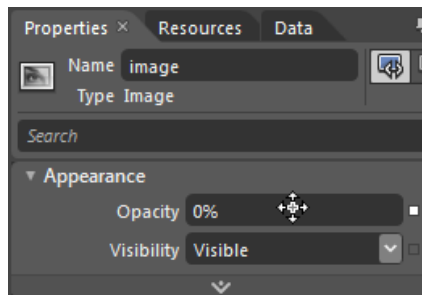
3-71 ábra: Új állapot hozzáadása egy állapotcsoporthoz

Vegyük észre, hogy amikor kijelölünk egy **VisualState**-et, a tervezőfelület állapotszerkesztési módba vált, vagyis bármit változtatunk a felületen, az nem az alkalmazás normál állapotában fog megjelenni, hanem csak akkor, amikor ebbe az állapotba lép az alkalmazás. A States panel tetején lévő Base gombra kattintva térhetünk vissza bármikor az állapotokon kívüli szerkesztéshez (3-72 ábra).



3-72 ábra: Egy állapot szerkesztése közben a tervezőfelület piros keretet kap

A vezérlők átlátszóságát az **Opacity** tulajdonsággal tudjuk állítani. Ha egy állapot szerkesztése közben 0%-ra állítjuk az **Opacity** értékét, a vezérlő eltűnik (3-73 ábra).

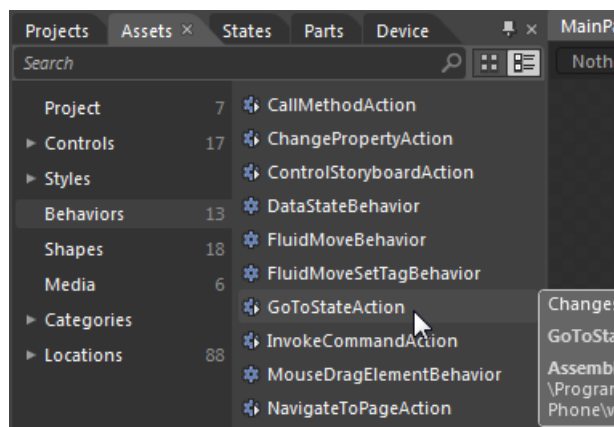


3-73 ábra: Az Opacity beállítása

A gombokat az **IsEnabled** tulajdonság állításával engedélyezhetjük, illetve tilthatjuk.

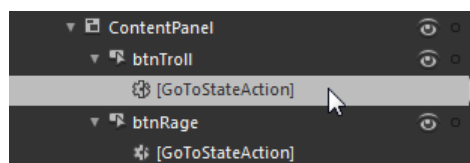
Meg kell oldani, hogy a gombnyomás hatására lefusson az állapotváltás. Ezt megtehetjük kódban (a **Click** eseménykezelőben meghívjuk a **VisualStateManager GoToState** metódusát, átadva neki a célállapot nevét), de ha tartjuk magunkat ahhoz, hogy deklaratív módon szeretnénk a felületet leírni, akkor más módszerre lesz szükség.

A **Behavior**ök segítségével meglévő vezérlőkhöz rendelhetünk hozzá extra viselkedéseket. Ha például arra van szükségünk, hogy egy gomb valamilyen eseménye hatására (nem kell feltétlenül a **Click**nek lennie) egy állapotváltást végezzen, a **GoToStateAction**ot aggregálhatjuk rá a gombra. A Behavioröket az Assets panel Behaviors listájában találjuk (3-74 ábra).



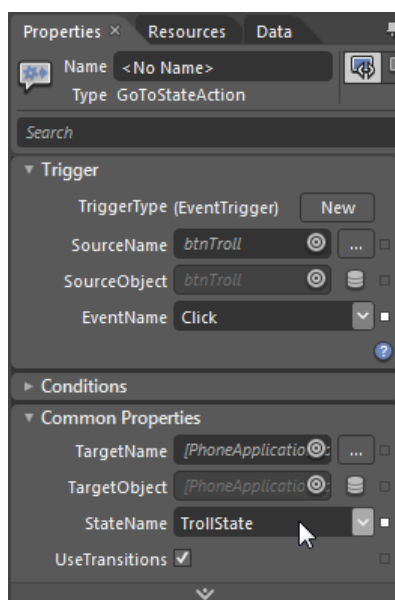
3-74 ábra: A *GoToStateAction Behavior* az *Assets* panelen

Innen egyszerűen ráhúzhatjuk őket a gombokra. Ennek eredményét az Objects and Timeline ablakban is láthatjuk (3-75 ábra).



3-75 ábra: A *GoToStateAction Behavior*, miután rádobtuk egy gombra

A **GoToStateAction**ot kiválasztva a Properties panelen beállíthatjuk, hogy mely esemény hatására (**EventName**) mely állapotba szeretnénk átváltani (**StateName**).



3-76 ábra: A *Behavior* beállítása

Ha futtatjuk az alkalmazást, és rákattintunk valamelyik gombra, az alkalmazás felülete rögtön abba a vizuális állapotba vált, amelyet meghatározunk a gomb számára (3-77 ábra).



3-77 ábra: A program felülete „rage” állapotban

## Animációk hozzáadása

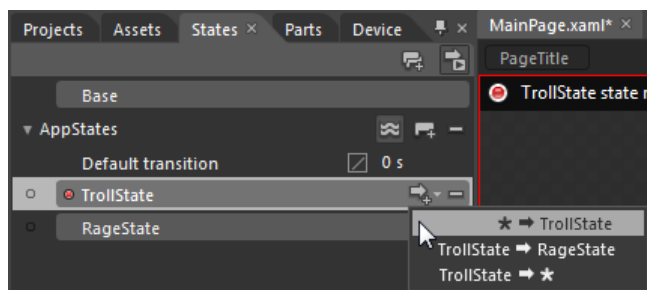
Vannak esetek, amikor ennyi nem elég. Szépen, gyorsan, mindenféle kód nélkül össze tudjuk állítani a felületet a különböző állapotokban, de érezhetően kicsit fapados a kinézet működése. Szebb lenne, ha például a kép nem egy pillanat alatt váltana át, hanem egy másodperc alatt elhalványulna, „átúszna” a másik képbe. Vagy például szeretnénk azt elérni, hogy mindig az a gomb legyen felül, amit a felhasználó megnyomhat – és a gombok helycseréje se pillanatszerű legyen, hanem látványos, animált.

Maga az animáció egy tulajdonság értékének adott idő alatti megváltoztatását jelenti. Korábban a legtöbb fejlesztőeszköznél ennek a leprogramozása teljes egészében a fejlesztő feladata volt: másodpercenként harmincszor feldobni egy eseményt, abban pedig kiszámolni a vezérlő aktuális elhelyezkedését, kinézetét a kiinduló- és a végállapot között, és utána (gyakran pixelről pixelre) kirajzolni a vezérlőt.

Mivel ennek az implementálása értékes időt vett el a fejlesztőtől, a Microsoft a WPF-fel előtérbe helyezte azt a megoldást, hogy az ehhez hasonló látványelemeket ne imperatív kóddal valósítsa meg a fejlesztő, hanem csak írja le (deklarálja), hogy mit is szeretne elérni. Vagyis az animációkat elég XAML-ben megfogalmazni: „milyen tulajdonságot, mennyi idő alatt, milyen végértékre szeretnék hozni”. Ennek előnye, hogy gyorsan leírható, és a háttérben az erre optimalizált keretrendszer elvégzi helyettünk a bonyolult kód megírását. A XAML előnye pedig, hogy könnyen generálható...

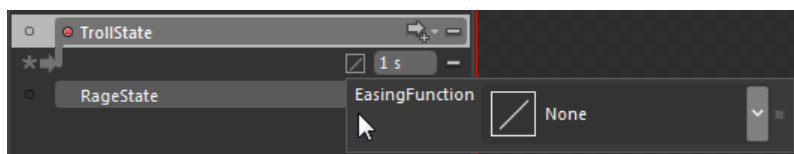
A két korábban létrehozott VisualState egyikében cseréljük fel a gombok helyzetét! A lényeg, hogy minden állapotban a másik állapotba váltó, engedélyezett gomb legyen felül! Ez a tervezőfelületen, drag-and-drop módszerrel érhető el, de nem árt arra ügyelni, hogy a megfelelő állapot legyen kijelölve, amikor a változtatásokat végzi az ember.

Ezután csak annyit kell tenni, hogy az egyes állapotokat kiválasztva (States panel) az állapoton lévő „Add transition” gombbal hozzárendelünk egy állapotátmenet-animációt egy adott átmenethez. (A \* → Célállapot nevű átmenet a bármely más állapotból az adott célállapotba történő animálást jelenti.)



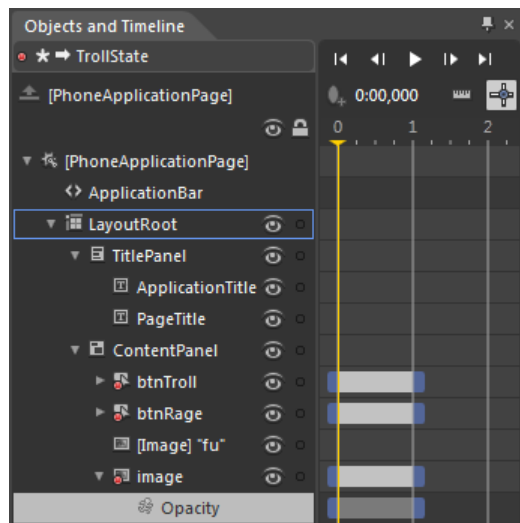
**3-78 ábra: Állapotátmenet hozzáadása egy állapothoz**

Ha létrejött az animáció, megadhatjuk, hogy mennyi ideig tartson (1s), illetve hogy milyen görbe mentén történjen (EasingFunction), ahogyan azt a 3-79 ábra jelzi.



**3-79 ábra: Az idő és az EasingFunction beállítása**

Érdemes egy pillantást vetni az Objects and Timeline ablakra is. Amikor kijelölünk egy állapotátmenetet a States panelen, itt jelzi a Blend, hogy a felületen lévő vezérlők közül melyek animáltak, és pontosan mely tulajdonságaik. Ha valamelyik tulajdonságot mégsem szeretnénk animálni, egyszerűen törölhetjük innen az animációt (3-80 ábra).



**3-80 ábra: Az animáció megjelenítése az Objects and Timeline panelen**

Mindezeket beállítva a program tesztelhető: az állapotok közötti átmenet nem pillanatszerűen megy végbe, hanem egy másodperc alatt változik az Opacity értéke 1-ről 0-ra és vissza, és a gombok is egy másodperc alatt csúsznak egyik helyről a másikra.

A VisualState-ek témaköre ennél természetesen jóval terebélyesebb. Néhány link a továbblépéshez:

Mike Taulty bevezető screencastja a VisualState-ekhez:

<http://active.tutsplus.com/tutorials/silverlight/understanding-visual-states-in-silverlight/>

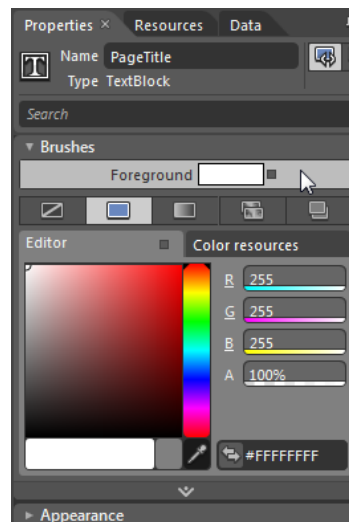
VisualState-ek használata egyéni vezérlőkben: <http://www.timmykokke.com/2010/11/using-visual-states-in-custom-controls-in-silverlight/>

## Témák használata

Ha szeretnénk, hogy alkalmazásaink illeszkedjenek a Windows Phone 7 megjelenésébe, a chooserek, launcherek (választók és indítók) használata, illetve egyéb operációs rendszerszintű integrációk elvégzése előtti első lépés, hogy a felületet konzisztenssé tesszük magával az operációs rendszerrel. Mi a helyzet például, ha egy adat- vagy vezérlősablonban szeretnénk felhasználni azt a színt, amelyet a felhasználó beállított a rendszer alapszínének? Vagy szeretnénk egy szöveget pontosan azzal a betűtípussal és betűmérettel megjeleníteni, amelyet az oldalak címénél használnak az alkalmazások?

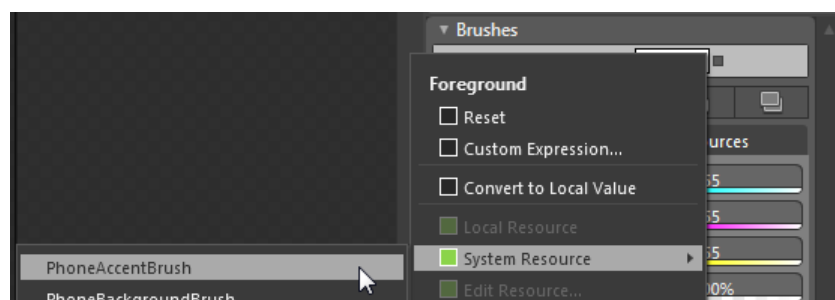
Nem kell pontosan tudnunk, hogy ezek a méretek, színek, betűtípusok milyenek is. A Silverlight az összes rendszerszintű alapbeállítást erőforrásként elérhetővé teszi számunkra. Mindössze annyit kell tennünk, hogy a kiválasztott tulajdonság kapcsán néhány kattintással beállítjuk, hogy az egy ilyen erőforrást használjon. Az alábbi példában a legutóbbi program (troll és rage) oldalcímének színét állítjuk be a telefonon kiválasztott alapszínre.

A **PageTitle TextBlock** kijelölése után a Brushes csoportban található **Foreground** tulajdonsággal állítható be egy szöveg betűszíne (3-81 ábra).



3-81 ábra: A színek beállítása a Properties panelen

Ahogy szinte minden, ez is adatköthető. A mellette lévő „Advanced options” gombra kattintva ezúttal a System Resources listából kell választani. Ez a lista felsorol minden olyan rendszerszintű erőforrást, amelynek típusa megegyezik a beállítani kívánt tulajdonság típusával. A **PhoneAccentBrush** az a szín, amely az operációs rendszernek a felhasználó által bármikor átállítható alapszíne (3-82 ábra).



3-82 ábra: A PhoneAccentBrush rendszerszintű erőforrás kiválasztása

Ennek kiválasztása után – ha elindítjuk az alkalmazást – látható, hogy a **PageTitle**-ben megjelenő felirat mindig az operációs rendszer színsémáját követi (3-83 ábra).



**3-83 ábra:** A **PageTitle** a telefon aktuális „accent” színével jelenik meg

## Összefoglalás

Ebben a fejezetben röviden megismerkedhettünk a Silverlight grafikus felhasználói felületének alapvető elemeivel. Számba vettük a leggyakrabban használt parancsvezérlőket, szövegmegjelenítési és szövegbeviteli vezérlőket, valamint a több elem megjelenítésére képes listavezérlőket.

Ezután áttekintettük, hogy a beépített vezérlőtípusokon milyen szolgáltatásokat tesz elérhetővé a Silverlight, hogy gazdag tartalmú és tudású felületeket építhessünk. Megismerkedtünk az adatkötés fogalmával és az adatkötési keretrendszer használatával. Megnéztük, hogyan lehet adat- és vezérlősablonokkal testre szabni egy adattípus megjelenését, illetve hogy hogyan cserélhetjük le egy vezérlő kinézetét a nekünk megfelelőre. Ezután ízelítőt kaptunk a vizuális állapotok használatából, az animációból, és megnéztük, hogyan lehet felhasználni az operációs rendszer vizuális erőforrásait saját alkalmazásainkban.

Mindezt a Visual Studio nélkül, kizárólag Expression Blend felhasználásával tettük meg, szinte teljes egészében elhagyva a saját kód megírását a felhasználói felület elkészítésénél.



## 4. Haladó alkalmazásfejlesztés Windows Phone-on

Az előző fejezetekben megismerkedhettünk azokkal az elvekkel és vezérlőkkel, amelyek mindenképpen szükségesek egy Windows Phone 7 alkalmazás fejlesztéséhez. Ebben a fejezetben olyan eszközöket fogunk áttekinteni, amelyek a platform lehetőségeit még jobban kihasználják – a Metro felhasználói felülettel konzisztens módon.

A fejezet első részében a térképek megjelenítését, testreszabhatóságát vesszük át, majd rátérünk a tartalmak rendszerezésére. Erre két módszert alkalmazunk: a tartalomegységeket az egyik esetben külön oldalakon jelenítjük meg – az ezek közti információcsere biztosításával –, míg a másik esetben egy oldalon, a horizontális strukturálást lehetővé tevő Panorama és Pivot vezérlők segítségével. Végül megismerkedünk a Silverlight Toolkit Windows Phone 7-hez tartozó részeivel, amelyek amellett, hogy számos új elemmel egészítik ki eszköztárunkat, segítenek a platformhoz illő felhasználói felületek és interakciók megteremtésében is.

### Térképek kezelése

A Microsoft az utóbbi években folyamatosan növekvő erőfeszítéseket fektet online szolgáltatásainak továbbfejlesztésébe, amelyek egyik megtestesülése a Bing Maps szolgáltatáscsomag. Ez a csomag azon túl, hogy teljes világtérképet tartalmaz, amit kedvenc keresőnket alkalmazva címkeresésre, útvonaltervezésre, forgalomadatok szerzésére használhatunk, lehetővé teszi, hogy ezeket a térképeket – és a kiegészítő szolgáltatásokat – saját alkalmazásainkban is elérhetővé tegyük. Például, egyszerűen tudunk kiemelt helyeket (Point of Interest, POI) megjeleníteni, ezek között optimális útvonalat tervezni, a webes verzióban is fellelhető kétféle nézet között váltogatni, és mindezt a Silverlightban megszokott teljes testreszabhatóság megtartásával tehetjük.

Egyszerű DataTemplate definiálásával létrehozhatunk akár egy olyan oldalt, amely megadott pozíciókat „pushpin”-ekkel (gombostű, rajzszeg) emel ki, mégpedig úgy, hogy ez a jelölés távolról csupán egy szövegdobozból álljon, de amint a felhasználó ráközelít a területre, ez kiegészül egy képpel, akár teljesen kicserélődik egy videolejátszóra. De természetesen itt a fejlesztő teljes szabadságot kap, ha akarunk, böngészőket vagy a Mango óta XNA játékokat is megjelenítünk a kiemelt helyeken, de valószínűleg nem baj, ha ezekkel a lehetőségekkel nem mindig élünk.

### *Feliratkozás a szolgáltatásra*

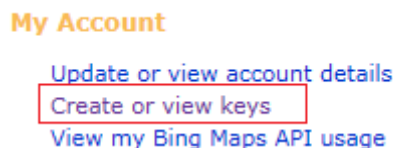
A szolgáltatás igénybevételéhez először regisztrálnunk kell a <https://www.bingmapsportal.com/> weblapon. Egy Live Id-t felhasználva jelentkezünk be és fogadjuk el a feltételeket! Itt kivételesen érdemes elolvasni a feltételeket tartalmazó dokumentumot, mivel igen szigorú szabályok vonatkoznak például a telefonos felhasználásra, amikbe jobb nem egy – esetleg több hónapon át tartó – fejlesztés végén belefutni.

Pár ezek közül:

- Nem írhatunk valós idejű navigációt megvalósító alkalmazást.
- A programunk nem működhet együtt más térképszolgáltatásokkal.
- A közlekedési adatok nem használhatók televízió, rádió, újság, azaz média esetében.
- Nem tárolhatunk közlekedési adatokat.
- Nem használhatunk pornográf pushpineket.

## 4. Haladó alkalmazásfejlesztés Windows Phone-on

Mindenesetre ha ezek után is akarunk térképeket használni, és elfogadtuk a feltételeket, akkor létrejön számunkra egy felhasználói fiók. Ehhez a fiókhoz alkalmazásokat regisztrálhatunk, ha a menü **Create or view keys** elemére klikkelünk, ahogyan azt a 4-1 ábra is mutatja.



4-1 ábra: Bing Maps kulcs létrehozása

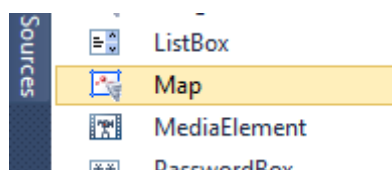
A kitöltést követően meg is kapjuk az alkalmazás kulcsát, ez az, amire majd szükségünk lesz a Bing Maps vezérlő létrehozásakor (lásd a 4-2 ábrán).

Application name	Key / URL	Update Key
WP7 Ebook	Ajdyq5FaN2RZvzvogm_X8UpS6vgC1pj_fO1lnAuux3JBr Not-for-profit	Update

4-2 ábra: A kapott kulcs, erre szükségünk lesz a kliens hitelesítéséhez

### A Bing Maps vezérlő használata

Hozzunk létre egy új Windows Phone Application projektet Visual Studioban, nevezzük el „Advanced Development”-nek! A Toolboxot kinyitva láthatjuk, hogy már az alap SDK-ban is megtalálható a számunkra fontos **Map** vezérlő (4-3 ábra), tehát a referencia hozzáadásával sem kell majd bajlódunk.

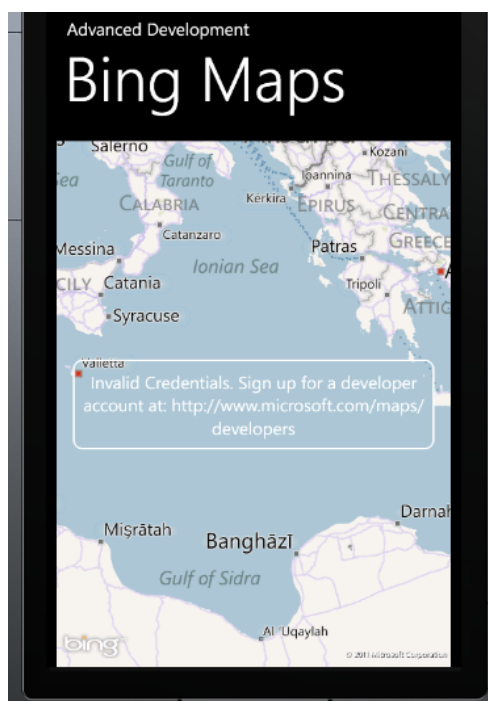


4-3 ábra: A Map vezérlő megtalálható a Toolboxban

Ha drag-and-drop módszerrel hozzáadjuk ezt a vezérlőt a megnyitott oldalunkhoz, akkor annak névtére automatikusan belekerül a XAML kódba. Ezt természetesen kézzel is megtehetjük az alábbi kód segítségével:

```
xmlns:maps="clr-namespace:Microsoft.Phone.Controls.Maps;assembly=Microsoft.Phone.Controls.Maps">
...
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <maps:Map />
</Grid>
```

A kód futtatásakor már láthatjuk is, hogy a Bing Maps vezérlő megjelenik. Ha nagyítunk, megpróbáljuk mozgatni a térképet, akkor a vezérlő elkezd kommunikálni a szolgáltatással, aminek első feladata a kliens hitelesítése lesz. Mivel még nem adtuk meg a regisztrációkor kapott kódot, ez meg fog hiúsulni, amit a térkép közepén megjelenő hibaüzenet tudat velünk, amint az a 4-4 ábrán látható.



4-4 ábra: A térkép, közepén a hitelesítés hiányát jelölő hibaüzenettel

Állítsuk be tehát a hitelesítéshez szükséges kulcsot a vezérlő tulajdonságai között, és nevezzük is el a térképet!

```
<maps:Map Name="myMap"
  CredentialsProvider="Ajdyq5FaN2RZvzvogm_...."/>
```

Ennek hatására a figyelmeztetés eltűnik, és rendesen használhatjuk a vezérlőt. Adjunk meg egy középpontot a térképen a **Center** tulajdonság segítségével! Ez a tulajdonság **GeoCoordinate** típusú, és XAML kódból a pozíciót leíró koordinátákat is felhasználhatunk az értékadás során. A vezérlő viszont még ezt követően is a teljes világtérképet fogja megjeleníteni, függetlenül a **Center** pozíciótól, hiszen nem állítottuk még be a nagyítás fokát. Ezt a **ZoomLevel** megadásával változtathatjuk meg:

```
<maps:Map Name="myMap"
  CredentialsProvider="Ajdyq5FaN2RZvzvogm_... "
  Center="47.639655, -122.129404"
  ZoomLevel="17" />
```

Az egyszerűbb felhasználás kedvéért és a tesztelés miatt is érdemes megjeleníteni a **ZoomBar** gombjait. Ezek az emulátoron különösen hasznosak tudnak lenni a kicsinyítés és nagyítás gombokkal való vezérelhetősége miatt:

```
ZoomBarVisibility="Visible"
```

A térkép két különböző módon jelenhet meg:

- **RoadMode:** Ebben az esetben nem a műholdképekkel találkozhatunk, hanem az ezek alapján készített, csak az utakra összpontosító nézetet kapunk. Ez az alapértelmezett mód.
- **AerialMode:** Teljes műholdképek, amelyeken az előző nézet útvonalai természetesen ugyanúgy ki vannak emelve.

A módok közötti váltáshoz egy kicsit többet kell dolgoznunk, ugyanis a térkép **Mode** tulajdonsága egy olyan objektumpéldányt vár, ami a **MapMode** osztályból származik – ez a megjelenítéshez szükséges

#### 4. Haladó alkalmazásfejlesztés Windows Phone-on

értékeket, beállításokat tartalmazza. Az előbb leírt módok használatához szerencsére léteznek előre definiált osztályok, azokat csak példányosítani kell, például az alábbi módszerrel:

```
using Microsoft.Phone.Controls.Maps;  
//...  
public MainPage()  
{  
    InitializeComponent();  
    myMap.Mode = new AerialMode();  
}
```

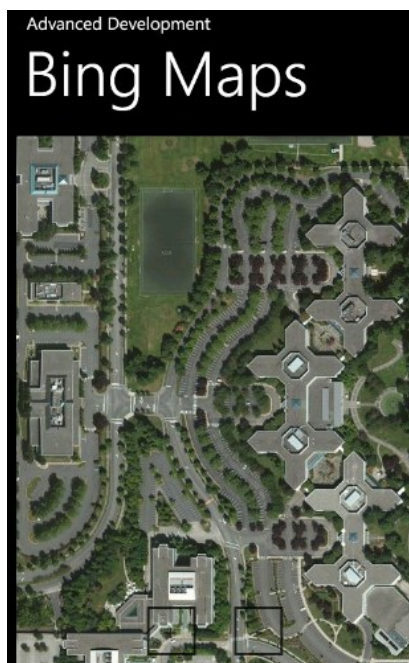
Természetesen erre ugyanúgy lehetőségünk van a XAML leírásból is:

```
<maps:Map Name="myMap"  
    CredentialsProvider="Ajdyq5FaN2RZvzvogm_..."  
    Center="47.639655,-122.129404"  
    ZoomLevel="17"  
    ZoomBarVisibility="Visible">  
    <maps:Map.Mode>  
        <maps:AerialMode />  
    </maps:Map.Mode>  
</maps:Map>
```

Tűntessük még el a Bing logóját és copyright feliratát a **LogoVisibility** és **CopyrightVisibility Collapsed**-re állításával! Ha mindennel végeztünk, akkor a teljes eddigi kódunk (a névtér definíciója mellett) csupán ennyi:

```
<maps:Map Name="myMap"  
    CredentialsProvider="Ajdyq5FaN2RZvzvogm_..."  
    Center="47.639655,-122.129404"  
    ZoomLevel="17"  
    ZoomBarVisibility="Visible"  
    LogoVisibility="Collapsed"  
    CopyrightVisibility="Collapsed">  
    <maps:Map.Mode>  
        <maps:AerialMode />  
    </maps:Map.Mode>  
</maps:Map>
```

A kód hatására meg is jelenik egy teljes funkcionalitással bíró térkép, középen a Microsoft főhadiszállásával, amint az a 4-5 ábrán látható.



**4-5 ábra: A térkép pozicionálva és nagyítva Aerial módban**

## ***Pushpinek***

Egy térkép mit sem érne kiemelt információk, címek, megjelölt pozíciók nélkül. Ennek eszköze a szintén a **Maps** vezérlő névtérhez tartozó **Pushpin** osztály. Ez egyszerűen, csupán koordináták és kiírandó szöveg megadásával használható vezérlő, viszont szükség esetén a Silverlightben megszokott könnyedséggel testre is szabhatjuk azt. Ezeket a pushpineket akár egyesével, XAML vagy C# kódból hozzáadogathatjuk a térképhez, de akár dinamikus listákat is köthetünk a vezérlőhöz.

Egy Pushpin felvétele XAML-ben az alábbi módon történhet:

```
<maps:Pushpin Location="47.639655, -122.129404" Content="Microsoft" />
```

Ugyanez a feladat kódban az osztály példányosításával és a térkép Children kollekciójához való hozzáfűzéssel jár:

```
myMap.Children.Add(new Pushpin
{
    Location = new System.Device.Location.GeoCoordinate(47.63876, -122.12862),
    Content = "C#"
});
```

Az így kapott jelöléseink végtelenül egyszerűek, csupán a tartalomnak beállított szöveget mutatják az adott pozíción, amint azt a 4-6 ábra is mutatja.



**4-6 ábra: Egyszerű Pushpinek a térképen**

A megjelenítés felüldefiniálása a XAML leírásból egyszerűbb, ugyanis itt csupán a **Content** attribútumszerű értékadását kell kicserélni az ún. „property element” szintakszissal való leírásra:

```
<maps:Pushpin Location="47.639655,-122.129404" Background="Transparent">
    <StackPanel>
        <TextBlock Text="MS" HorizontalAlignment="Center" />
        <Image Source="http://www.deviantart.com/download/128133698/Windows_Vista_Icon_HD_by_magbanuamicah.png" Width="50" />
    </StackPanel>
</maps:Pushpin>
```

Ezzel a kis módosítással már teljesen kézben tarthatjuk a pushpin megjelenítését, az itt alkalmazott kód például a 4-7 ábrán látható kinézetet eredményezi.



4-7 ábra: Testreszabott Pushpin

### Rétegek alkalmazása

Sokszor elvárt funkcionalitás a kiemelt helyek szűrése egy adott kategória alapján (például a környező kocsmákat jelenítsük meg, a benzinkutakat rejtjük el), vagy akár teljes kikapcsolása. Ez a POI-k egyszerű hozzáadásával körülményes lenne, azonban rétegekbe rendezve a megjelenített tartalmat a feladat sokkal egyszerűbbé válik.

Ennek kipróbálására hozzunk létre egy **POI** osztályt, ami tartalmaz egy pozíciót, valamint a hely megjelölését. Ennek példányaikat két listába, a benzinkutak és a bárak listájába fogjuk sorolni:

```
public class POI
{
    public GeoCoordinate Location { get; set; }
    public string Name { get; set; }
}
```

A példányokat egy privát, csak olvasható lista tartalmazza, amit az oldal létrehozásakor töltünk fel pár elemmel, és egy publikus, csak get metódussal rendelkező tulajdonságon keresztül érünk el. Egy valós projektben a listákat természetesen bármilyen adatforrás vagy webszolgáltatás előállíthatja. Az adatkötés miatt használunk **ObservableCollection** típust, bár itt ennek előnyeit nem fogjuk kihasználni:

```
private readonly ObservableCollection<POI> _pubs = new ObservableCollection<POI>
{
    new POI
    {
        Location= new System.Device.Location.GeoCoordinate(47.640055,-122.120204),
        Name = "Old Man's"
    },
    new POI
    {
        Location = new System.Device.Location.GeoCoordinate(47.63876, -122.12862),
        Name = "Oportó"
    }
};
```

```
public ObservableCollection<POI> Pubs
{
    get { return _pubs; }
}
```

Most, hogy előállítottuk a megjelenítendő listát, hozzunk létre egy új réteget a térképen belül:

```
<maps:MapLayer Name="lPubs">
    <maps:MapItemsControl ItemsSource="{Binding Pubs}">
        <maps:MapItemsControl.ItemTemplate>
            <DataTemplate>
                <maps:Pushpin Location="{Binding Location}">
                    <StackPanel>
                        <TextBlock Text="{Binding Name}"
                            HorizontalAlignment="Center"/>
                        <Image Source="Images/Beer.png"
                            Width="50" />
                    </StackPanel>
                </maps:Pushpin>
            </DataTemplate>
        </maps:MapItemsControl.ItemTemplate>
    </maps:MapItemsControl>
</maps:MapLayer>
```

A réteget egy **MapLayer** példány reprezentálja, ennek egy nevet is adtunk, hogy később tudjunk rá hivatkozni. Ezen belül a **MapItemsControl** teszi lehetővé azt, hogy ne csak egy-egy elemet jelenítsünk meg, hanem az egész lista tartalmát, még hozzá úgy, ahogyan azt az **ItemTemplate** részben meghatározzuk. A kódban egy **PushPin**-t tűzünk ki a **Location** által meghatározott pozícióra. Ez tartalmazza a hely nevét, valamint egy olyan képet, amit korábban az **Images** mappába mentettünk.

Képeknél a Build Actiont érdemes Contentre állítani, mivel ekkor a médiaelem nem lesz a szerelvény része, hanem a **.xap** csomag egy mappájába kerül. Így a relatív hivatkozások révén könnyebb azt felhasználni programunkban.

Ha kódunkat így próbáljuk futtatni, akkor a POI-k még nem fognak megjelenni a térképen, hiszen hivatkoztunk ugyan a bekötendő adatokra, de magát az adatkötést még nem végeztük el. Legegyszerűbb, ha ezt most az oldal konstruktorában tesszük meg az alábbi kóddal:

```
public MainPage()
{
    InitializeComponent();

    this.DataContext = this;
}
```

Most futtatva a programot a definiált POI-k már meg is jelennek (4-8 ábra). Az előző módszerhez hasonlóan hozzunk létre egy listát, ami például benzinkutakat tartalmaz:



4-8 ábra: A pozíció-listák a térképen megjelenítve

Ahhoz, hogy kihasználjuk a rétegek előnyeit, helyezzünk el két gombot az ApplicationBar-on, melyekkel a megfelelő réteg megjelenítését tudjuk ki-be kapcsolni:

```
<phone:PhoneApplicationPage.ApplicationBar>
  <shell:ApplicationBar IsVisible="True" IsMenuEnabled="True">
    <shell:ApplicationBarIconButton IconUri="/Images/beer.png"
                                   Text="Bárok"
                                   Click="Pubs_Click"/>
    <shell:ApplicationBarIconButton IconUri="/Images/petrol.png"
                                   Text="Benzin"
                                   Click="PetrolStations_Click"/>
  </shell:ApplicationBar>
</phone:PhoneApplicationPage.ApplicationBar>
```

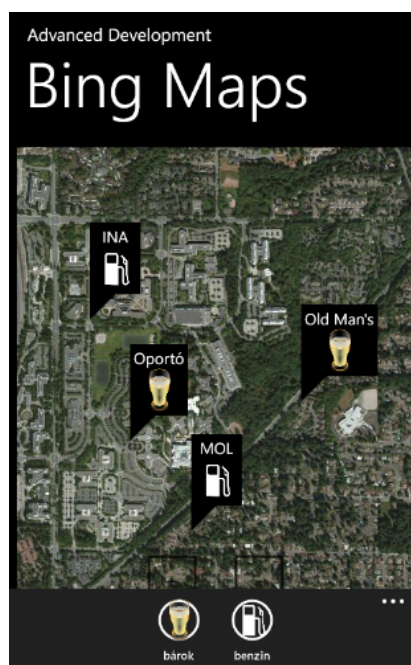
**A gombokhoz tartozó eseménykezelők elvégzik a réteg Visibility tulajdonságának átbillentését:**

```
private void Pubs_Click(object sender, EventArgs e)
{
    if (lPubs.Visibility == System.Windows.Visibility.Visible)
    {
        lPubs.Visibility = System.Windows.Visibility.Collapsed;
    }
    else
    {
        lPubs.Visibility = System.Windows.Visibility.Visible;
    }
}
```

A benzinkutak rétegéhez kapcsolódó **PetrolStations\_Click** kódja teljesen hasonló ehhez.

Eddigi munkánk végeredménye tehát egy olyan térkép, amely két külön rétegen jelenít meg testreszabott pushpineket, valamint a lenti vezérlőgombokkal ezeknek a rétegeknek a megjelenítését váltogathatjuk, amint azt a 4-9 ábra mutatja.





4-9 ábra: Gombok az ApplicationBaron a rétegek megjelenítésének szabályozására

## Egyéb szolgáltatások

A térképen kívül számos egyéb rendkívül hasznos szolgáltatást kínál számunkra a Bing Maps. Gyakorlatilag az összes főbb funkció, amit egy térképprogramtól elvárhatunk, szabványos SOAP webszolgáltatásokon keresztül elérhető a fejlesztők számára. Néhány fontosabb ezek közül:

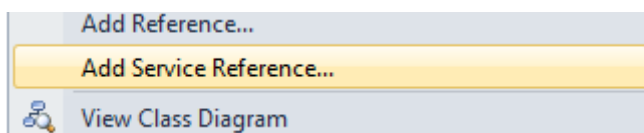
- **Geocode Service** – Címek, nevezetes helyek, földrajzi nevek keresése, koordinátákká alakítása vagy koordináták alapján ezek kinyerése.
- **Route Service** – Útvonalak tervezése általunk megadott köztes pontok között. Azaz nem csupán kezdő- és végpontot adhatunk meg, hanem érintett pozíciók egész listáját. Ezen a listán a tervező algoritmus sorban végigmegy, majd ezt a gráfot bejáró ponthalmazt ad vissza nekünk. Lehetőségünk van az utazás módjának beállítására (gyalog / autóval), valamint meghatározhatjuk, hogy milyen szempontból keressen optimális eredményt (sebesség, távolság, sőt akár a forgalmi helyzettel való számolást is kérhetjük).
- **Search Service** – Nevezetes helyek, például éttermek, mozik keresése – ez pozícióhoz is köthető.

A fenti szolgáltatások az alábbi címeken érhetők el:

Szolgáltatás	Cím
Geocode Service	<a href="http://dev.virtualearth.net/webservices/v1/geocodeservice/geocodeservice.svc">http://dev.virtualearth.net/webservices/v1/geocodeservice/geocodeservice.svc</a>
Route Service	<a href="http://dev.virtualearth.net/webservices/v1/routeservice/routeservice.svc">http://dev.virtualearth.net/webservices/v1/routeservice/routeservice.svc</a>
Search Service	<a href="http://dev.virtualearth.net/webservices/v1/searchservice/searchservice.svc">http://dev.virtualearth.net/webservices/v1/searchservice/searchservice.svc</a>

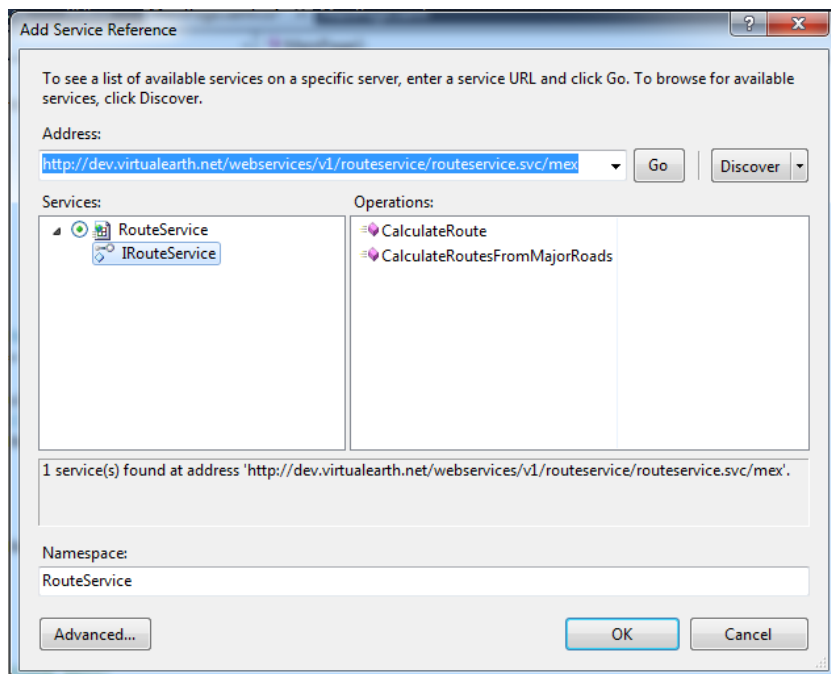
A szerverrel való kommunikációról bővebben a 9. fejezet szól, itt csak a feladathoz szükséges mélységig érintjük a témát.

A szolgáltatások hozzáadásához kattintsunk a projektre az egér jobb gombjával, majd a felugró listán válasszuk az **Add Service Reference** opciót (4-10 ábra)!



4-10 ábra: Szolgáltatás referenciájának hozzáadása

Az itt felugró ablakon adjuk meg a választott szolgáltatás címét, majd kattintsunk a **Go** gombra (4-11 ábra)! Ekkor a Visual Studio lekéri a cím mögött lévő szolgáltatás WSDL leírását, és kiolvassa belőle az elérhető interfészeket és azokon belül a felkínált funkciókat. A dialógusban lévő **Discover** gomb lehetőséget biztosít, hogy a környezet a mi saját Solutionünkön belül keressen elérhető szolgáltatásokat, de mi itt csak kliens alkalmazást fejlesztünk, így erre nem lesz szükségünk.



4-11 ábra: A szolgáltatás leírása alapján a Visual Studio felismeri az elérhető funkciókat

### Útvonaltervezés

A táblázatból válasszuk ki a Route Service-t, és ennek címét adjuk meg az Add Service Reference dialógusban! A generált osztály névtére legyen **RouteService**! Az OK gombra kattintva létrejönnek a kapcsolatot felépítő és a kommunikációt biztosító osztályok. Ezt követően példányosítással létrehozhatunk egy proxy objektumot, amelynek metódusai a szolgáltatást címzik meg, így a távoli metódusokat egyszerű függvényhívásokkal érhetjük el programunkból.

Hozzuk is létre ezt a proxyt a **MainPage** konstruktorában! Ahhoz, hogy a keretrendszer megtalálja ezt az osztályt, természetesen a **RouteService** névteret fel kell vennünk egy **using** taggal a file elején:

```
public MainPage()
{
    InitializeComponent();
    this.DataContext = this;
    RouteServiceClient proxy = new RouteServiceClient();
}
```

Ezeknél a szolgáltatásoknál szintén szükségünk lesz arra az azonosítóra, amit a Bing Maps regisztrációjánál kaptunk. Ezt érdemes egy külön változóban tárolni, a **MainPage** osztály elején. Itt már nem használhatjuk a XAML-ben rendelkezésünkre álló egyszerűsített értékadást, ehelyett egy **Credential** típusú objektumot is létre kell hoznunk:

```
private const string credStr = "Ajdyq5FaN2Rzvzvogm_... ";
private readonly Credentials cred = new Credentials { ApplicationId = credStr};
```

Térjünk vissza ezután a konstruktorba, és állítsuk össze az útvonaltervezéshez szükséges kérést! Szükségünk lesz egy olyan objektumra, ami a kalkuláció szempontjait definiálja. Erre a **RouteOptions** osztályt alkalmazhatjuk, amelyben az egyéb beállítások mellett arra mindenképp figyeljünk, hogy a **RoutePathType** tulajdonságot állítsuk a **RoutePathType** enumeráció **Points** elemére, mivel ez alapértelmezés szerint **None**, azaz az érintett pontok lekérdezését külön igényelnünk kell!

```
var options = new RouteOptions
{
    // a pontok listáját igényelnünk kell!
    RoutePathType = RoutePathType.Points,
    // leggyorsabb útvonal keresése
    Optimization = RouteOptimization.MinimizeTime,
    // forgalom alapján is optimalizáljon az időre
    TrafficUsage = TrafficUsage.TrafficBasedTime,
    // autóval közlekedünk
    Mode = TravelMode.Driving
};
```

Ezzel a konfigurációval tehát olyan útvonalat keresünk, amely a majd átadott pontok között autóval a lehető leggyorsabban bejárható úgy, hogy a jelenlegi forgalmi helyzetet is figyelembe vesszük. A kérésre adott válasz az egyéb fontos információkon felül tartalmazni fogja az érintett pontok listáját is. Ez a lista az út „töréspontjait” írja le, vagyis azokat a pontokat, amelyeket csupán egyenesekkel össze kell kötnünk és fel kell rajzolnunk a térképre.

Hozzuk most létre a bejárandó pontok listáját! Ez egy szintén a **RouteService** névtérben (vagy amilyen nevet a szolgáltatás referenciájának hozzáadásánál választottunk) lévő osztály, a **WayPoint** példányait jelenti. Itt tartalmazzon csak két elemet, a kiindulási és az érkezési pontot:

```
// érintett pozíciók
var points = new ObservableCollection<Waypoint>
{
    // kezdőpont
    new Waypoint{Location = Pubs[0].Location},
    // végpont
    new Waypoint{Location = PetrolStations[0].Location}
};
```

A konfigurációs objektumok létrehozása után már előállíthatjuk azt a kérést, amit a szolgáltatás már fel tud dolgozni:

```
var request = new RouteRequest
{
    Credentials = cred,
    Options = options,
    Waypoints = points
};
```

Mielőtt megpróbálkoznánk az útvonal előállításával, készítsük fel a megjelenítő felületet az adatok fogadására. A XAML-ben keressük ki újra a térkép leírását, és egészítsük ki az alábbi sorokkal:

```
<TextBlock Name="tbDistance" />

<maps:MapPolyline Name="lRoute"
    Stroke="#FF2C76B7"
    Opacity="0.85"
    StrokeThickness="6" />
```

A **tbDistance** fogja megjeleníteni az útvonal hosszát, az **lRoute** nevű **MapPolyline** objektum pedig felrajzolja azt. Ehhez csupán a vezérlő **Locations** tulajdonságát kell beállítani egy **ObservableCollection<Waypoint>** példányra, aminek elemeit használva rajzolja majd ki az út szakaszait.

### A szolgáltatás meghívása

WP7 esetében a WCF automatikusan aszinkron műveleteket generál a szolgáltatással való kommunikációra. Ennek nagy előnye, hogy a felhasználói felület nem áll le arra az időre, amíg a kérésünkre előállított válasz megérkezik, azaz a készülék végig válaszra kész marad. Ennek következménye, hogy azonnali visszatérési értékkel nem rendelkeznek a függvények, ehelyett az eredmény elkészültét jelző események argumentumaiként juthatunk hozzá az igényelt adatokhoz. Ennek szellemében írjuk meg a lekérdezést:

```
proxy.CalculateRouteCompleted += (s, e) =>
{
    tbDistance.Text = e.Result.Result.Summary.Distance.ToString();
    var lc = new LocationCollection();
    foreach (var item in e.Result.Result.RoutePath.Points)
    {
        lc.Add(item);
    }
    lRoute.Locations = lc;
    // pozícionáljuk a térképet az útvonalra!
    myMap.SetView(e.Result.Result.Summary.BoundingBox);
};
```

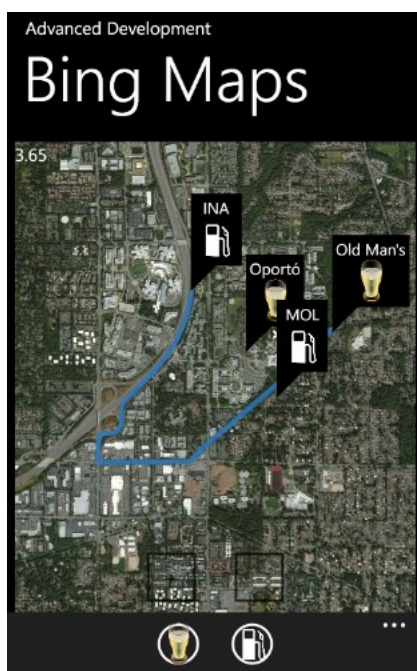
A **CalculateRouteCompleted** eseménykezelője az egyszerűség kedvéért lehet helyben definiált anonim metódus is. Ennek **e** paramétere egy **CalculateRouteCompletedEventArgs** típusú objektum, ami tartalmazza a művelet végrehajtását leíró adatokat (pl. sikeres volt-e), valamint a kalkulált útvonal részleteit. A **Summary** tulajdonságon keresztül elérhetjük a szakaszok együttes hosszát, az ennek megtételéhez szükséges időt, valamint egy **BoundingBox** nevű elemet, ami annak a téglalapnak az északkeleti és délnyugati koordinátáját tartalmazza, amely magába foglalja a teljes útvonalat.

A pontok halmaza sajnos más típusban érkezik, mint amit a megjelenítő elem vár, de ezt egy konverziós ciklussal orvosolhatjuk. Ezen kívül minden adat kötése egyértelmű. Miután ezeket a műveleteket elvégeztük, állítsuk a térkép nézetét a visszakapott **BoundingBox**-re!

Nincs más hátra, indítsuk el az aszinkron kérést!

```
// kérés indítása
proxy.CalculateRouteAsync(request);
```

Fordítsuk le és futtassuk a projektet, és nézzük meg, hogy hogyan jelenik meg mindez az emulátoron (4-12 ábra)!



4-12 ábra: Az útvonaltervezést is tartalmazó kész alkalmazás

Az ábrán látható, hogy a teljes útvonal kékkel kiemelve látszik, a bal felső sarokban pedig megjelenik a szakaszok együttes hossza.

## Navigáció

A tartalom rendszerezésének egyik magától értetődő eszköze önálló oldalak létrehozása. Az oldalak különálló egységeket alkotnak, azonban ahogyan weblapok esetében is, itt is gyakran szükséges valamilyen kommunikációt biztosítani a lapok között. Ebben a részben ezeket a lehetőségeket fogjuk áttekinteni.

### Oldalak

Minden Windows Phone 7 projekt a létrehozásakor két XAML állománnyal rendelkezik: egy **MainPage** nevű oldallal és egy alkalmazásszintű erőforrásokat és eseményeket tartalmazó **App.xaml** fájljal. Ha csak a Solution Explorerben nézünk rá a fájlrendszerre, akkor ezek hasonlóan tűnhetnek, de egészen másról van szó, ugyanis a **MainPage** szülő osztálya a **PhoneApplicationPage**, míg az **App** az **Application**-ból származik. A **PhoneApplicationPage** elképzelhető úgy is, mint egy weblap, egy önálló entitás, amely elemeket jelenít meg, és rendelkezik a más oldalak eléréséhez szükséges eszközökkel.

Az oldalak közti navigáció megvalósítására két lehetőségünk van:

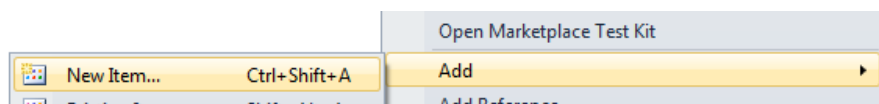
- **HyperlinkButton** – egyszerűen használható vezérlő, ami egy linkhez hasonlóan jelenik meg, de felüldefiniálhatjuk a kinézetét.
- **NavigationService** – ez egy osztály, amin keresztül könnyen kézben tarthatjuk a navigációs folyamatokat. Az egyszerű oldalváltáson túl függvényeket, eseményeket kínál számunkra.
- A **NavigationService** fontosabb metódusai, tulajdonságai és eseményei:
- **GoBack()** – Visszanavigál az előző oldalra, ugyanúgy, mint a Back gomb lenyomásakor.
- **Navigate(Uri)** – Átnavigál az Uri által meghatározott címre.
- **Navigating** – A navigáció indítványozásakor következik be.
- **Navigated** – A navigáció végét jelző esemény.
- **CurrentSource** – Az éppen mutatott oldal címe.

- **Source** – A jelenleg aktív vagy mindjárt aktívvá váló oldal címe.
- **CanGoBack** – Tudunk visszafelé navigálni? Azaz van-e bejegyzés az oldalak közti lépkedést rögzítő veremben?

A **NavigationService** rendelkezik egy **CanGoForward** tulajdonsággal is, de ez a WP7 esetében mindig **false** értékű lesz, valamint az ehhez kapcsolódó előrenavigálás, azaz a **GoForward(Uri)** függvény kivételt dob.

A **PhoneApplicationPage** navigációs eszköztára két fontosabb részből áll, a **NavigationContext**-ből és az előbb említett **NavigationService**-ből. Az első segítségével férhetünk hozzá a **QueryString**-hez, ami csakúgy, mint weblapok esetében, az URL-ben átadható paraméterek kulcs-érték szótárát jelenti. A **NavigationService** a felsorolt lehetőségein túl elérhetővé teszi a korábbi oldalak listáját (**BackStack**) és egy eljárást is, amivel ezt módosíthatjuk (**RemoveBackEntry**).

Ezek kipróbálására egészítsük ki az előző alkalmazást egy új oldallal, ami egy főmenüt fog tartalmazni, amelyen keresztül elérhetjük a többi lapot. Ehhez a projekt gyorsmenüjéből válasszuk az Add ⇒ New Item funkciót (4-13 ábra)!



4-13 ábra: Új elem hozzáadása a projekthez

Az Add New Item ablakban válasszuk a **Windows Phone Portrait Page** sablont, az oldal neve pedig legyen **MainMenuPage**! A többi sablon is ehhez hasonló szerkezetű, csupán néhány alapbeállításban térnek el egymástól, mint például a lap orientációja.

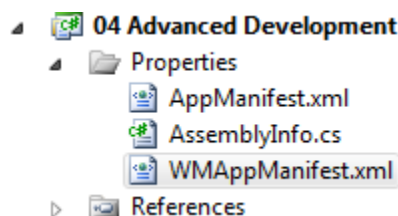
Ezen az új oldalon adjunk a **ContentPanel** nevű Gridhez egy StackPanelt, azon belül pedig helyezzünk el egy gombot:

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <StackPanel>
        <Button Name="btnMap"
            Content="Térkép"
            Click="btnMap_Click" />
    </StackPanel>
</Grid>
```

Menjünk át a kódot tartalmazó forráskód állományhoz (legegyszerűbb az F7 lenyomásával a XAML fájlhoz tartozó forráskódra ugrani). Itt a gomb **Click** eseménykezelőjében használjuk fel a már említett **NavigationService**-t az oldalt váltásra:

```
private void btnMap_Click(object sender, RoutedEventArgs e)
{
    NavigationService.Navigate(new Uri("/MainPage.xaml", UriKind.Relative));
}
```

A **Navigate** függvény egy URI-t vár, amely alapértelmezés szerint abszolút, nekünk viszont most egyszerűbb lesz, ha ezt relatívrá cseréljük. Ezzel a paranccsal címezzük meg a fejezet első részében létrehozott **MainPage.xaml**-t! Az oldalak közti váltás már működne, azonban egy probléma még van: a **MainPage** nyílik meg először, onnan pedig sehova nem tudunk navigálni! A megoldáshoz nyissuk ki a **Properties** mappát, azon belül pedig keressük ki a **WMAppManifest.xml**-t (4-14 ábra)!



4-14 ábra: A WAppManifest.xml a projekt Properties mappájában

Azért is érdemes megismerkednünk ezzel az állománnyal, mert itt tudjuk átírni alkalmazásunk metaadatait (például a kiadó nevét, címét, leírását, az operációs rendszer verzióját), beállítani a kezdőoldalt, a Splash képet és az ikonokat. Ezenfelül a **Capabilities** szekcióban találhatjuk az igényelt képességek listáját. Ha például nem használunk Push Notificationt, akkor érdemes az erre vonatkozó elemet eltávolítani (bár ennek elmulasztásával sem veszünk semmit). Most elegendő a kezdőoldalt kicserélnünk, ehhez írjuk át a **Tasks**-on belül a **DefaultTask**-ot:

```
<Tasks>
  <DefaultTask Name = "_default" NavigationPage="MainMenuPage.xaml"/>
</Tasks>
```

Most már tesztelhetjük programunkat! A kezdőképernyő a „Térkép” gombot fogja mutatni, amire kattintva átjutunk a korábban létrehozott útvonaltervezőnkre, ahonnan pedig a hardveres Back gombbal térhetünk vissza.

## Adatátvitel az oldalak között

Egyszerű, adatkezelést nélkülöző programoknál elég lehet az, hogy független oldalak között váltogatunk. Ugyanez már kevés lehet, ha például egy lista valamely elemének részleteit szeretnénk egy külön lapon megmutatni. Ugyanis valahogyan meg kell mondanunk a részleteket tartalmazó oldalnak, hogy melyik elemet választotta a felhasználó. Ennek legegyszerűbb megoldása a **QueryString**ek használata. Ekkor hasonlóan, mint weblapok esetében, csak az URL-t kell kiegészítenünk az átadandó kulcs-érték párokkal:

```
private void btnMap_Click(object sender, RoutedEventArgs e)
{
    NavigationService.Navigate(new Uri("/MainPage.xaml?myParam=test", UriKind.Relative));
}
```

A paraméter kiolvasása a másik oldalon a **NavigationContext** segítségével egyszerűen megoldható:

```
string myParam;
if (NavigationContext.QueryString.TryGetValue("myParam", out myParam))
{
    MessageBox.Show(myParam);
}
```

## Navigációs metódusok

A **PhoneApplicationPage** őssztálya, a **Page** a **NavigationService** osztályon túl tartalmaz virtuális metódusokat, amelyek közvetlenül az oldal létrejötte után vagy a deaktiválódása előtt futnak le, így adva lehetőséget arra, hogy a navigációval kapcsolatos feladatokat (például argumentumok kiolvasása, animálás) elvégezzünk. Mivel ezek virtuális eljárások, a saját oldalainkon felül tudjuk definiálni őket az **override** kulcsszó használatával:



```
protected override void OnNavigatedTo(System.Windows.Navigation.NavigationEventArgs e)
{
    base.OnNavigatedTo(e);
    // saját kód
}
```

Ehhez hasonlóan hozzáférhetünk az oldal elhagyását megelőző **OnNavigatingFrom** és **OnNavigatedFrom** függvényekhez is.

### BackStack

Az alkalmazásunkon belüli lapváltásokat egy verem tárolja, amiről a hardveres Back gomb leveszi a legfelső elemet, és visszavigál erre az oldalra. Ehhez a veremhez mi is hozzáférhetünk, sőt a visszalépést is egyszerű függvényhívással megvalósíthatjuk.

Hozunk létre egy új oldalt **Navigation** névvel, amit készítsünk fel egy kapott paraméter és a korábbi címek kiírására a **ContentPanel** kiegészítésével!

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <StackPanel>
        <TextBlock Name="tbMyParam" Text="Nem érkezett paraméter"/>
        <Button Name="btnDelete" Content="Törlés" Click="btnDelete_Click" />
        <ItemsControl Name="icNavigationList"
            DisplayMemberPath="Source"/>
    </StackPanel>
</Grid>
```

Az **icNavigationList** fogja megjeleníteni a látogatott oldalak vermét. A **DisplayMemberPath** tulajdonság beállításával jelezzük, hogy a későbbiekben bekötött elemek **Source** tulajdonságát akarjuk kiírni! Vegyünk fel még egy gombot is, amivel a verem legutóbbi elemét el tudjuk távolítani, és ezzel módosíthatjuk a navigációs szolgáltatás működését!

A kódfájlban írjuk felül az **OnNavigatedTo** eljárást, hogy az beállítsa a XAML-ben definiált vezérlők hiányzó értékeit, és töltsük ki a gomb eseménykezelőjét is!

```
protected override void OnNavigatedTo(System.Windows.Navigation.NavigationEventArgs e)
{
    base.OnNavigatedTo(e);
    // saját kód
    string myParam;
    if (NavigationContext.QueryString.TryGetValue("myParam", out myParam))
    {
        tbMyParam.Text = myParam;
    }
    icNavigationList.ItemsSource = NavigationService.BackStack;
}

private void btnDelete_Click(object sender, RoutedEventArgs e)
{
    NavigationService.RemoveBackEntry();
    /* nem használtunk adatkötést, úgyhogy új objektumra kell állítanunk az ItemsSource-ot! */
    icNavigationList.ItemsSource = NavigationService.BackStack.ToList();
}
```

Ha felveszünk valahova egy linket, ami erre az oldalra mutat, már megfigyelhetjük a **NavigationService** működését. Az **OnNavigatedTo** működése magától értetődő, a gomb eseménykezelője viszont egy kis magyarázatot érdemel. A **RemoveBackEntry** függvény törli a navigációs verem felső elemét. Mi viszont az egyszerűség kedvéért nem használtunk adatkötést, azaz a **BackStack** nem ad magáról értesítést a megjelenítő vezérlőnek arról, hogy frissült. Ilyenkor nem elegendő az **ItemsSource**-ot újra a **BackStack**-



re állítani, hiszen a referencia ekkor nem változik, úgyhogy a vezérlő sem frissül. Ehelyett egy új példányra, például a **BackStack**-ből előálló listára kell állítanunk az **INavigationList**-et.

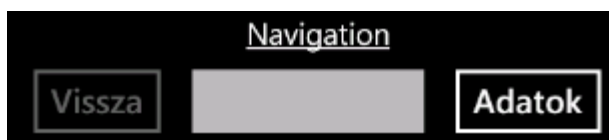
A dolog varázsa az, hogy ezzel a törléssel a hardveres Back gomb működését is befolyásoljuk, hiszen ugyanezen verem alapján dolgozik az is. Így ha töröljük a legfelső elemet, akkor az előtte lévőre fog a gomb visszavezetni.

## Az alkalmazás kiegészítése navigációs vezérlőkkel

Az előző példa kevés oldal esetén még nem igazán látványos. Több lap esetében pedig érdemes lehet a minden oldalon megvalósítandó funkciókat a folyamatos másolgatás helyett egy **UserControl**-ba szervezni. Ez azért is érdekes téma, mert a **UserControl** osztály nem tartalmaz **NavigationService**-t. Komplexebb navigáció megvalósításához olyan felhasználói vezérlőt kell készítenünk, amely használja a **NavigationService**-t.

Ennek megvalósításához az Add New Item menüben válasszuk a Windows Phone User Control sablont, és nevezzük el vezérlőnket **NavigationBar**-nak!

Ezt a vezérlőt úgy alakítjuk ki, hogy bárhonnan a **Navigation.xaml**-re navigál – akár paraméter átadásával –, valamint felkínál egy gombot, ami a hardveres vissza gombbal megegyező funkcionális. A kész vezérlő megjelenítése a 4-15 ábrán látható. A **Navigation** vezérlő egy **Hyperlink**, ami a paraméter nélküli oldalváltást teszi lehetővé. Az **Adatok** gomb ugyanezt valósítja meg, azzal a kiegészítéssel, hogy a középső **TextBox** értékét átadja a megcímzett oldalnak. A **Vissza** gomb pedig csupán akkor aktív, amikor a **BackStack** nem üres, azaz az alkalmazáson belül is tudunk még visszafelé navigálni.



4-15 ábra: Az elkészítendő navigációs vezérlő

Az ehhez tartozó XAML kód:

```
<UserControl x:Class="_04_Advanced_Development.NavigationBar"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d"
    FontFamily="{StaticResource PhoneFontFamilyNormal}"
    FontSize="{StaticResource PhoneFontSizeNormal}"
    Foreground="{StaticResource PhoneForegroundBrush}"
    d:DesignHeight="480" d:DesignWidth="480"
    Height="100" Width="450">
    <Grid x:Name="LayoutRoot" HorizontalAlignment="Center">
        <StackPanel>
            <HyperlinkButton Content="Navigation" NavigateUri="/Navigation.xaml" />
            <StackPanel Orientation="Horizontal"
                HorizontalAlignment="Center">
                <Button Name="btnBack"
                    Content="Vissza"
                    Click="btnBack_Click" />
                <TextBox Name="tbNameParam" Width="200"/>
                <Button Name="btnNavigationPage"
                    Content="Adatok"
                    Click="btnNavigationPage_Click" />
            </StackPanel>
        </StackPanel>
    </Grid>
</UserControl>
```

A felhasználói vezérlőkben a navigáció kulcsa az, hogy referenciát szerzünk egy olyan objektumra, ami rendelkezik az ehhez szükséges képességekkel. A WP7-es programokban szerencsére mindig található ilyen, hiszen az **App** osztály tartalmaz egy publikus **PhoneApplicationFrame** típusú, **RootFrame** nevű mezőt. Ez egy fix keret, ami a beleágyazott oldalak cserélgetését segíti, többek között a navigációhoz szükséges feltételek (**Source** tulajdonság, navigációs függvények, események) biztosításával.

Menjünk most a **NavigationBar** kódfájljába, és valósítsuk meg a **Frame** tulajdonságot, amely egy hivatkozás a **RootFrame**-re!

```
private Frame _frame;
public Frame Frame
{
    get
    {
        if (_frame == null)
        {
            _frame = (App.Current.RootVisual as PhoneApplicationFrame);
        }
        return _frame;
    }
}
```

A **Frame** a navigációs feladatok során úgy kezelhető, mintha az egy **NavigationService** példány lenne, nagyjából ugyanazokat a funkciókat érhetjük így is el. Ezek tudatában fejezzük be a forrásfájl kiegészítését!

```
public NavigationBar()
{
    InitializeComponent();

    this.Loaded += (s, e) =>
    {
        btnBack.IsEnabled = Frame.CanGoBack;
    };
}

private void btnNavigationPage_Click(object sender, RoutedEventArgs e)
{
    string target = "/Navigation.xaml";
    if (!string.IsNullOrEmpty(tbNameParam.Text))
    {
        Frame.Navigate(new Uri(string.Format("{0}?myParam={1}", target, tbNameParam.Text),
        UriKind.Relative));
    }
    Frame.Navigate(new Uri(target, UriKind.Relative));
}

private void btnBack_Click(object sender, RoutedEventArgs e)
{
    Frame.GoBack();
}
```

A vezérlő ezzel elkészült, már csak fel kell használni azt! Ehhez a XAML fájlokat kell csak módosítanunk, először vegyük fel a projekt névterét például **local** névvel:

```
xmlns:local="clr-namespace:_04_Advanced_Development"
```

Majd helyezzük el az egyéb tartalmak között, bárhol az oldalon:

```
<local:NavigationBar VerticalAlignment="Bottom" />
```

Próbáljuk ki alkotásunkat úgy, hogy bejárjuk az összes oldalt, majd a végén meglátogatjuk a navigációt összegző lapot, törölünk párat, és megnyomjuk a vissza gombokat! Ha minden jól működik, akkor sikerült elsajátítani az oldalak közti navigáció trükkjeit (4-16 ábra)!

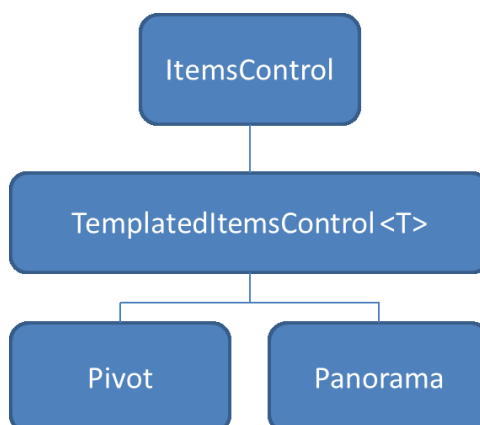


4-16 ábra: A BackStack tartalma és az átadott paraméter is látszik a Navigation oldalon

## Pivot és Panorama

Az előző részben láttuk, hogyan tudjuk alkalmazásunkat oldalakkal strukturálni. Szorosabban kapcsolódó adatok megjelenítésére vagy egyszerűen látványosabb UI létrehozására is két remek vezérlő áll rendelkezésünkre, amelyeket ebben a részben ismerhetünk meg.

A **Pivot** és a **Panorama** két nagyon hasonló, horizontálisan elnyúló vezérlő. Mindkettő az **ItemsControl**-ből, pontosabban a **TemplatedItemsControl**-ből származik (4-17 ábra). A WP7 programok kinézetének meghatározó formálói, nem véletlen, hogy a beépített alkalmazások többsége is ezek felhasználásával készült el.



4-17 ábra: A Pivot és Panorama kapcsolata az osztályhierarchia alapján

Példák a vezérlők alkalmazására az Office programból:



4-18 ábra: Balra az Office alkalmazás Panorama vezérlője, jobbra a leveleket csoportosító Pivot látható

### Különbségek

Pivot	Panorama
Lapokra osztott rendszerezés az egyes lapok közti átmenet nélkül.	Egy horizontálisan elnyúló, folyamatos oldal érzetét kelti.
A háttérkép horizontálisan fix (a tabokra definiálhatunk természetesen külön hátteret).	A háttérkép szélesebb, mint a kijelző, így ezen is lapozhatunk.
Csak az aktív lap és a mellette lévők kerülnek leképzésre a megjelenítéshez.	Az összes elem egyszerre kerül leképzésre a megjelenítéshez.
Egyszerre csak egy lap látszik, ezért ennek területét teljesen kihasználhatjuk.	Érdemes nem telezsúfolni a lapokat, mivel a következő lap jobbról belóg az aktívba.
Adatok rendszerezésére, kategóriák kezelésére, esetleg hierarchikus elrendezésre érdemes használni.	Figyelemkeltő kezdőoldalak, funkciók bemutatása esetén hasznos.
Pl. RSS olvasóban kategóriák szétválasztásakor használjuk.	Pl. Alkalmazás kezdőoldala.

Fontos megjegyezni, hogy Panorama vezérlő esetén kerülni kell az ApplicationBar használatát, ha erre szükségünk van, mindenképp használjunk Pivotot! Emellett a Panorama és a Pivot egymásba ágyazva nem használható!

A vezérlők népszerűségét jelzi az is, hogy külön oldalsablont építettek be mindkettő számára (sőt, alkalmazássablont is találhatunk). Hozunk létre ezekből egy-egy példányt, azaz egy **PanoramaPage** és egy **PivotPage** nevű oldalt! Az elemeket persze bármilyen oldalon létrehozhatjuk a vezérlő szokásos hozzáadásával is, nem kell ragaszkodnunk a sablon használatához.

A sablonból generált **PanoramaPage** tartalmaz egy Panorama vezérlőt, egy címmel és két előre létrehozott elemmel. Ezek az elemek jelenleg csak üres Gridet jelenítenek meg, de ha átírjuk őket, akkor bármilyen komplex lapot használhatunk.

```
<Grid x:Name="LayoutRoot">
    <controls:Panorama Title="my application">

        <!--Panorama item one-->
        <controls:PanoramaItem Header="item1">
            <Grid/>
        </controls:PanoramaItem>
    </controls:Panorama>
</Grid>
```

```

</controls:PanoramaItem>

<!--Panorama item two-->
<controls:PanoramaItem Header="item2">
    <Grid/>
</controls:PanoramaItem>
</controls:Panorama>
</Grid>

```

A dolog látványossá tételéhez csupán egy széles háttérkép hiányzik. Ilyeneket könnyen találunk az interneten, beágyazásuk pedig a **Background** tulajdonság hozzáadásával történik (4-19 ábra):

```

<controls:Panorama.Background>
    <ImageBrush ImageSource="Images/bgPanorama.jpg" />
</controls:Panorama.Background>

```



**4-19 ábra: Panorama egy háttérkép megadása után**

A táblázatban kiemelt különbségeken túl a Pivot használata teljesen megegyezik a Panorama vezérlőjével, így ennek kipróbálására most nem térünk ki.

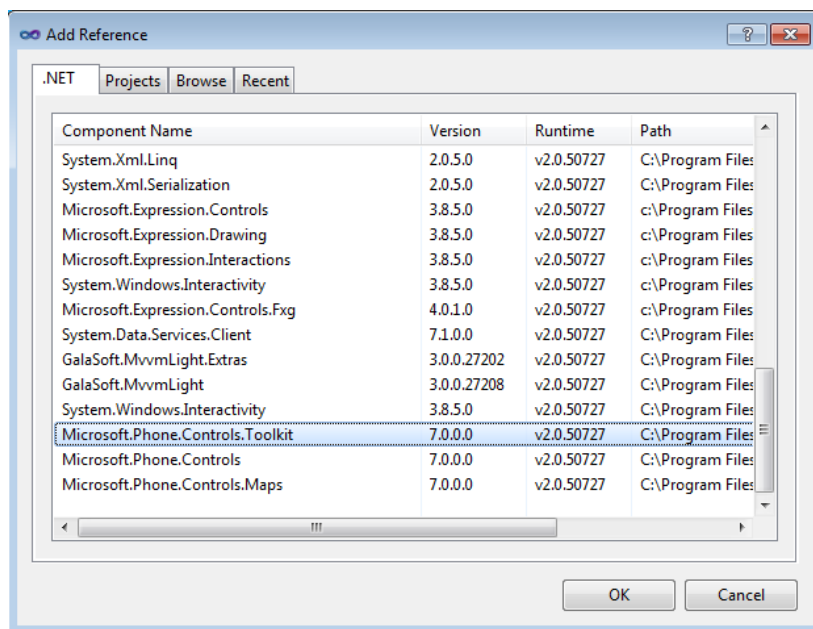
## Silverlight Toolkit for Windows Phone7

Az eddig bemutatott vezérlőkkel már egy gazdag eszközkészletből válogathatunk, de számos ismétlődő feladat még így is ránk hárul. Ezek elvégzését könnyíti meg a Silverlight Toolkit, ami új vezérlők felkínálásán túl a Metro stílus kialakításában is hasznos társunk lehet.

### Telepítés

A készlet használatához töltsük le a telepítőt a projekt hivatalos weblapjáról:

<http://silverlight.codeplex.com>. Miután telepítettük a csomagot, az elérhető a Visual Studio Add Reference dialógusából (4-20 ábra).



4-20 ábra: A Toolkit szerelvénye a telepítés után elérhető

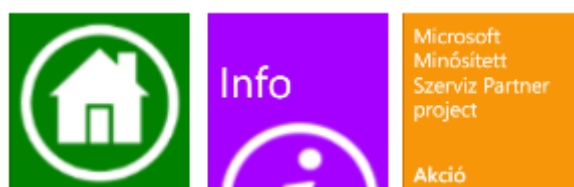
Ha mégsem találjuk itt, akkor keressük ki a Microsoftos SDK-k közül! Nálam ez a szerelvény itt található:  
**C:\Program Files\Microsoft SDKs\Windows Phone\v7.1\Toolkit\Aug11\Bin\Microsoft.Phone.Controls.Toolkit.dll**

Ahhoz, hogy az elemeket a XAML-ben is elérjük, a már ismert módon adjuk hozzá a névteret a leíráshoz:

```
xmlns:toolkit="clr-namespace:Microsoft.Phone.Controls;assembly=Microsoft.Phone.Controls.Toolkit"
```

### Fontosabb vezérlők

**HubTile** – A főoldal lapkáihoz hasonló dinamikus vezérlő, nagyszerűen használható menük kialakítására, amelyek az egyszerű navigáción túl információt, üzeneteket is szolgáltatnak a felhasználó számára (4-21 ábra).

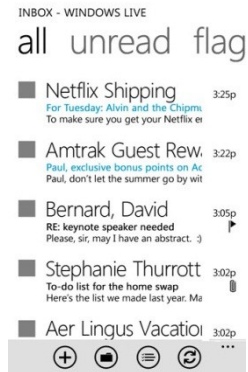


4-21 ábra: A HubTile állapotai

Ezt a vezérlőt az alábbi módon illeszthetjük be a felhasználói felületbe::

```
<toolkit:HubTile Name="hirekTile" Tap="hirekTile_Tap" Title="Hírek"
    Source="Images/news.png"
    Background="#E51400"
    GroupTag="TileGroup"
    Margin="6"
    Notification="1 új hír"
    DisplayNotification="True"
    Message="Microsoft Minősített Szerviz Partner project">
```

**MultiselectList** – Az Office alkalmazás levéllistájához hasonlóan lehetővé teszi, hogy a felsorolt elemek közül többet is kiválasszunk (4-22 ábra).

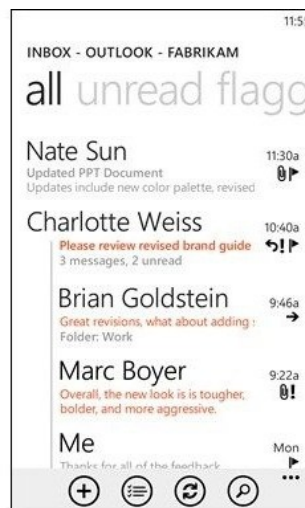


**4-22 ábra: A levelek kiválasztásakor is egy ugyanilyen funkcionális vezérlővel találkozunk**

Használata:

```
<toolkit:MultiselectList>
  <toolkit:MultiselectItem Content="1. elem" />
  <toolkit:MultiselectItem Content="2. elem" />
  <toolkit:MultiselectItem Content="3. elem" />
</toolkit:MultiselectList>
```

**ExpanderView** – Szintén a leveleknél (pontosabban a hosszabb levélváltásoknál) megszokott lista, amelynek egy elemét kiválasztva az lenyílik, és a részletei is láthatóvá válnak (4-23 ábra).



**4-23 ábra: Hasonló vezérlő a levélváltások részletezésénél**

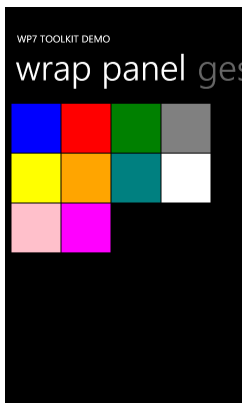
Ezt a vezérlőt az alábbi módon használhatjuk:

```
<toolkit:ExpanderView>
  <toolkit:ExpanderView.Expander>
    <Rectangle Height="80" Width="400" Fill="Blue" />
  </toolkit:ExpanderView.Expander>
  <toolkit:ExpanderView.Items>
    <TextBlock Text="Alelem 1" />
    <TextBlock Text="Alelem 2" />
  </toolkit:ExpanderView.Items>
</toolkit:ExpanderView>
```

**LockablePivot** – Kiegészíti a Pivotot egy zárolási lehetőséggel, ami lezárja az aktív lapot, azaz nem tudunk róla a telefon kezeléséhez használt gesztusok segítségével a környező lapokra navigálni. Ez hasznos lehet, ha horizontálisan görgethető elemeket használunk a lapon, hiszen ilyenkor nem egyértelmű, hogy például egy **ScrollBar** értékét kívánjuk módosítani, vagy egyszerűen csak lapoznánk.

**PageTransitions** – Az oldalak közti átmenetet kísérő animációk használatát megkönnyítő osztály. Segít a megszokott Metro stílusú átmeneteket egységes módon használni.

**WrapPanel** – A **Panel** osztályból származó konténertípus. A **StackPanel**hez hasonlóan egymás mellé vagy alá helyezi az elemeket az **Orientation** tulajdonságától függően. A **StackPanel**lrel szemben, ha eléri a sor (vagy oszlop) végét, akkor a kitöltést a következő sorban (oszlopban) folytatja. Ezzel egyszerűen tudunk ismeretlen elemszámú listákat, például galériákat megjeleníteni (4-24 ábra).

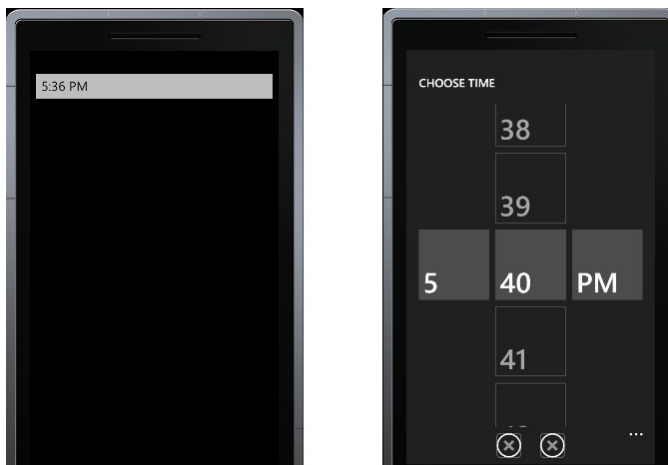


4-24 ábra: WrapPanel horizontális elrendezéssel

Használata:

```
<toolkit:WrapPanel>
  <toolkit:HubTile Background="Green" Title="Zöld" />
  <toolkit:HubTile Background="Red" Title="Piros " />
  <toolkit:HubTile Background="Blue" Title="Kék" />
</toolkit:WrapPanel>
```

**TimePicker** – Időpontok megadására használható vezérlő, az Alarm alkalmazásból ismerhetjük. Két nézettel rendelkezik, az egyik szövegesen kiírja a jelenleg beállított időpontot, erre kattintva pedig a másik, az időpontot beállító elrendezést láthatjuk (4-25 ábra).



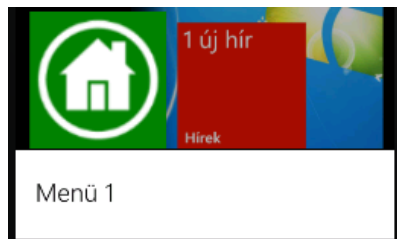
4-25 ábra: A TimePicker két nézete



Használata:

```
<toolkit:TimePicker Name="tpAlarm"/>
```

**ContextMenu** – A PC-ken megszokhattuk, hogy ha egy elemre kattintunk az egér jobb gombjával, akkor az elemre vonatkozó menü ugrik fel. A WP7 platformon ezt a viselkedést egy elem hosszan lenyomva tartásával érhetjük el, a menü előállításában pedig a **ContextMenu** vezérlő segít minket. Ez a menü mindig egy másik vezérlőhöz kapcsolódik, így gyakorlatilag egy szolgáltatást, a **ContextMenuService**-t kell felvennünk a funkcióval kiegészítendő kontrollba. A menü megjelenésén túl a hivatkozó vezérlő ki is emelkedik a többi közül, pontosabban ezt kivéve minden más kissé távolabbra kerül, így az animálással sem kell külön foglalkoznunk (4-26 ábra).



**4-26 ábra: ContextMenu 1 elemmel, a háttérben pedig látható a kiemelő effekt**

Használata:

```
<toolkit:HubTile Tap="hirekTile_Tap" Title="Minden"
    Source="Images/all.png"
    Margin="6"
    Background="#FF008000"
    GroupTag="TileGroup"
    Notification="Noti"
    Message="Microsoft Minősített Szerviz Partner project">
    <toolkit:ContextMenuService.ContextMenu>
        <toolkit:ContextMenu>
            <toolkit:MenuItem Header="Menü 1" Click="MenuItem_Click" />
        </toolkit:ContextMenu>
    </toolkit:ContextMenuService.ContextMenu>
</toolkit:HubTile>
```

## Példaprogram

Hozzunk létre egy olyan oldalt, ami a toolkit előnyeit felhasználva egy áttekinthető, informatív és esztétikus menüt tartalmaz. Egy ehhez hasonló lap remek belépési pont lehet bármely alkalmazás számára, hiszen tájékoztatja a felhasználót a fontos funkciókról, üzeneteket jeleníthet meg és látványos, animált felületet is biztosít.

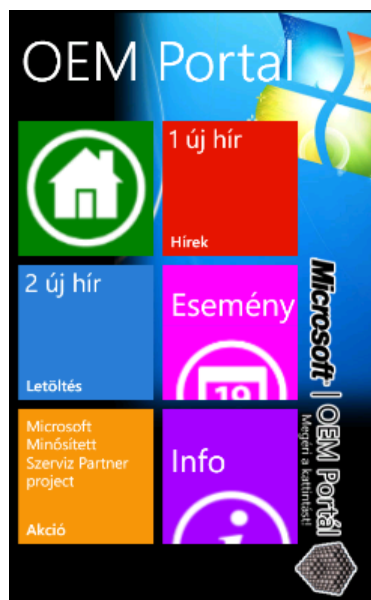
```
<Grid x:Name="LayoutRoot">
    <Grid.Background>
        <ImageBrush ImageSource="Images/tileBg.png" />
    </Grid.Background>
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto"/>
        <RowDefinition Height="*/>
    </Grid.RowDefinitions>

    <!--TitlePanel contains the name of the application and page title-->
    <StackPanel x:Name="TitlePanel" Grid.Row="0" Margin="12,17,0,28">
        <TextBlock x:Name="PageTitle" Text="OEM Portal" Margin="9,-7,0,0"
            Style="{StaticResource PhoneTextTitle1Style}"/>
    </StackPanel>
</Grid>
```

```
</StackPanel>

<!--ContentPanel - place additional content here-->
<toolkit:WrapPanel x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <toolkit:HubTile Tap="hirekTile_Tap" Title="Minden"
        Source="Images/all.png"
        Margin="6"
        Background="#FF008000"
        GroupTag="TileGroup"
        Notification="Noti"
        Message="Microsoft Minősített Szerviz Partner project">
        <toolkit:ContextMenuService.ContextMenu>
            <toolkit:ContextMenu>
                <toolkit:MenuItem Header="Menü 1" Click="MenuItem_Click" />
            </toolkit:ContextMenu>
        </toolkit:ContextMenuService.ContextMenu>
    </toolkit:HubTile>
    <toolkit:HubTile Name="hirekTile" Tap="hirekTile_Tap" Title="Hírek"
        Source="Images/news.png"
        Background="#E51400"
        GroupTag="TileGroup"
        Margin="6"
        Notification="1 új hír"
        DisplayNotification="True"
        Message="Microsoft Minősített Szerviz Partner project">
    </toolkit:HubTile>
    <toolkit:HubTile Tap="hirekTile_Tap" Title="Letöltés"
        Source="Images/downloads.png"
        Background="#297FD5"
        Margin="6"
        GroupTag="TileGroup"
        Notification="2 új hír"
        DisplayNotification="True"
        Message="Microsoft Minősített Szerviz Partner project">
    </toolkit:HubTile>
    <toolkit:HubTile Tap="hirekTile_Tap" Title="Esemény"
        Source="Images/calendar.png"
        Margin="6"
        Background="#FF00FF"
        GroupTag="TileGroup"
        Notification="Noti"
        Message="Microsoft Minősített Szerviz Partner project">
    </toolkit:HubTile>
    <toolkit:HubTile Tap="hirekTile_Tap" Title="Akció"
        Source="Images/akcio.png"
        Margin="6"
        Background="#FFF79608"
        GroupTag="TileGroup"
        Notification="Noti"
        Message="Microsoft Minősített Szerviz Partner project">
    </toolkit:HubTile>
    <toolkit:HubTile Tap="hirekTile_Tap" Title="Info"
        Source="Images/info.png"
        Margin="6"
        Background="#A500FF"
        GroupTag="TileGroup"
        Notification="Noti"
        Message="Microsoft Minősített Szerviz Partner project">
    </toolkit:HubTile>
</toolkit:WrapPanel>
</Grid>
```

A fenti kód pedig (természetesen a képek **Images** mappába másolása után) a 4-27 ábrán látható megjelenést eredményezi.



4-27 ábra: HubTile-ok WrapPanelbe csomagolva

## Összefoglalás

A fejezet elején részletesen megismerkedhettünk a Bing Maps szolgáltatás felhasználási lehetőségeivel, ezek között a kiemelt pozíciók jelölésével, az egyszerű címkereséssel és az útvonaltervezéssel. Ezt követően átnéztük a tartalmak rendszerezésére szolgáló műveleteket: az oldalak kezelését, azokon belül pedig a Panorama és Pivot vezérlőket. Végül a Silverlight Toolkit előnyeit ismerhettük meg.

Ezekkel a módszerekkel nemcsak hogy rendkívül egyszerűen valósíthatunk meg komplex alkalmazásokat, de könnyedén varázsolhatjuk a Metro stílushoz és a Mango felhasználói felület koncepciójához illeszkedő megjelenésűnek azokat.



## 5. Az alkalmazás életciklusa

Az előző fejezetek átolvasása után egyre boldogabban tarthatjuk kezünkben Windows Phone telefonunkat; most már mi magunk is láthattuk, hogy ezekre az eszközökre fejleszteni majdnem minden apró mozzanatát tekintve más, mint a többi, elterjedt mobilplatform esetében, és mégis rendkívül ismerős azoknak, akik jártak már a .NET világában. Mielőtt továbbhaladunk a funkcionalitás megismerésében, kicsit vissza kell vennünk a lendületből és tartani egy apróbb kitérőt. Minden új technológia rendelkezik saját karakterisztikával, mely egyedibbé, az addig megszokottól különbözővé teszi.

Ez Windows Phone 7.5 esetében is teljes mértékben igaz. Számos olyan megoldás van a kíváncsi szemek elől elrejtve, valahol az operációs rendszer lelkén belül, amely teljesítményben és funkcionalitásban rendkívül sokat hozzátesz a rendszer működéséhez. A teljes fejezet célja – jól áttekinthető formában – megismerkedni az alkalmazások működésének keretével, átnézni, mi minden történik a háttérben az alkalmazás indulásától a back vagy a Start gomb lenyomásáig.

A Windows Phone 7.5 (a továbbiakban csak a kódnevéen említve Mango) újdonságai között nagyon fontos újra megismerkedni egy régi ismerőssel, amely nem más, mint a *multitasking*, azaz a belső ütemező rendszer, amely lehetővé teszi, hogy egyszerre több alkalmazás osztozhasson a telefon biztosította erőforrásokon. Megnézzük, milyen feladatokat kell ellátnunk az alkalmazás indulásakor és kiléptetésekor, hogyan tudjuk elmenteni a felhasználó által kialakított állapotot és azt a következő indításkor helyreállítani, ezzel biztosítva, hogy a felhasználónak a munkáját ne kelljen előlről kezdenie, hanem onnan folytathassa, ahol abbahagyta. Ezek után tovább folytatjuk az ismereteink bővítését olyan újdonságok áttekintésével, mint a zenelejátszóval való integráció vagy a hardveres hangerő-szabályozó gombok kezelése.

Lehetőségünk van riasztások és naptárbejegyzések készítésére és felügyeletére is, amelyek elérhetőek saját alkalmazásunkból és a telefon beépített eszközeiből is.

Háttérben futó adatszinkronizációt is megvalósíthatunk alkalmazásunkban, amellyel apró darabokra szétbontva a csomagokat a háttérben végezhetünk szerver-kliens kommunikációt.

De miért is ebben a fejezetben tárgyaljuk a fent említett, egymástól teljes mértékben különböző dolgokat? A közös dolog bennük: ezek mind a háttérben képesek dolgozni!

### Multitasking

A fenti kifejezéssel valószínűleg már mindenki találkozott valahol. Egyetemi óra keretein belül, az internetet böngészve, de akár az operációs rendszer súgójának olvasása közben is belefuthatunk. Az alapötlet szerint a multitasking azért felelős, hogy az operációs rendszer egyszerre ne csak egy alkalmazást tudjon futtatni, hanem a központi feldolgozóegység (CPU) kapacitását kihasználva ezek a programok egymás mellett dolgozhassanak, megfelelően teret engedve egymásnak. A való világra levetítve a dolgot makroszkopikusan nézve még mondhatjuk, hogy igazodtunk az eredeti feltevéshez, azonban közelebről tekintve ez nem igaz. A felhasználót megtévesztjük a gyors CPU-nak köszönhetően és elhítetjük vele, hogy valóban egyszerre futtat számos programot, de valójában nem másról van szó, mint a multitasking-rendszer ütemező komponenséről, amely a végrehajtandó feladatokat a legoptimálisabb módon juttatja el a központi egységhez, így ezeket egymás után nagyon gyorsan végrehajtva valóban a párhuzamosság érzetét kapjuk. Asztali operációs rendszerek esetében már nem igazán találkozhatunk olyannal, amely ezt a rendszert ne tartalmazná, azonban a Windows Phone 7 első verziójában ezt még nem találjuk – legalábbis a fejlesztők által használható módon nem. A következőkben választ fogunk rá kapni, hogy miért is volt ez így.

### A feladatütemezés elméletben

Ez a rész talán nem illeszkedik szorosan a könyv témájához, azonban véleményem szerint nagyon érdekes téma, és nagyon sokat segít a Mangóban lévő megoldás megértésében – persze ahhoz, hogy ezt a kedves Olvasó is át tudja érezni, szintén csillogó szemekkel kell, hogy tekintsen a számítógépes algoritmusokra és különböző matematikai problémákra.

Ahogy azt már említettem, asztali operációs rendszerek esetében ma már fel sem merül, hogy a hatalmas számítási kapacitás mellett egy operációs rendszer megengedje magának azt a luxust, hogy egyszerre csak egy programot szolgál ki. Azonban az utóbbi évek során az informatika átesett egy újabb – és egyben sokadik – metamorfózisán, és az addigi nézetek új irányba tolódtak el. Okos telefonok, táblagépek, beágyazott rendszerek jelennek meg sorra és mindegyik valamelyik népszerű operációs rendszer mobilizált változatát használja fedélzeti szoftverként. Gyorsan hozzászoktunk, és lassan elképzelni sem tudjuk ezek nélkül az életünket (főleg amióta a szó minden értelmében közösségi lények vagyunk). Azt várjuk el ezektől a kütyűktől, hogy ugyanazokat a feladatokat lássák el, amiket a számítógépünk – azonban a laptop és a telefon között is van néhány hatalmas különbség, csak jobban bele kell gondolni.

Elsőre ez a különbség nem tűnik olyan nagynak! Az én laptopomban négy processzormag van 2.4 GHz-es órajelen üzemeltetve, míg a telefonomban két processzormag 1.2 GHz-es teljesítménnyel. Ez matematikailag pont a fele lenne, de a kettő a valóságban összehasonlíthatatlan. Miért? A számítógémem processzora rengeteg segédprocesszort, közvetlenül elérhető és másodlagos tárat, valamint ehhez tartozó magas áteresztőképességű vezetőrendszert tartalmaz, megfelelő energiaellátás és hűtés mellett. Ezek a segédprocesszorok villámsebességgel végeznek el számos összetett aritmetikai műveletet, és a számítási sorok végén az eredményeket nagy szélességű buszokra továbbítják. Ezzel szemben a telefonom a számítógémem drabális hardvereit egyetlen chipbe tömöríti, ezt úgy hívjuk, hogy SoC (*System on a Chip*). Egyetlen apró chip feladata a CPU, a GPU (grafikus számítási egység) és egyéb vezérlők szerepeinek felvállalása minimális energiaellátás és hűtés hiánya mellett! Amikor telefonunkon egy játékkal játszunk, annak a számítási feladatai mind ugyanahhoz a chiphez futnak be, és annak kell biztosítania a másodpercenkénti 30 képkockát (telefonokon ez a szabványos képkockaszám másodpercenként) ahhoz, hogy egy játék élvezhető legyen. Nem beszélve arról, hogy az egymás mellett futó alkalmazások állapotait meg kell tartani, ezért az operatív memóriánkat is a lehető legjobban be kell osztanunk.

A feladatunk az, hogy van X darab folyamatunk (itt ne Windows vagy egyéb operációs rendszer folyamatra gondoljunk, a processzor szemszögéből nézzük a problémát, elemi szintű folyamatokról beszélünk), és ezeket úgy kellene ütemeznünk a processzor felé, hogy mindegyik folyamat dolgozhasson valamennyit, és így a felhasználó azt az érzetet kapja, hogy egyszerre, egymás mellett futnak az alkalmazásai. A legegyszerűbb megoldás úgy néz ki, hogy azt mondjuk, egyik folyamatot sem éheztetjük ki, mindegyik kap lehetőséget felszólalni, azonban csak bizonyos ideig. Egységesen mindegyik kap például 100 ms processzoridőt, ezzel gazdálkodhat egy menetben. Jön először A folyamat, dolgozik 100 milliszekundumig, utána őt követi B, utána jön C, majd ha C végzett, akkor megint A folyamat kerül előre a sorban. Ahhoz, hogy észrevegyük, mi ezzel a probléma, nézzünk egy példát!

A folyamatnak összesen 350 ms időre lesz szüksége ahhoz, hogy elvégezzen minden számítást. Ez esetben négyszer fog magának lefoglalni egy 100 ms hosszú intervallumot, de az utolsó iterációban 50 ms alatt befejezi a feladatát és terminál, így 50 ms üresen marad. Ebből látszik, hogy 50 ms processzoridő el lett pazarolva. A fenti algoritmust *round-robin ütemezési algoritmus*nak hívják, és nagy előnye, hogy könnyen implementálható. Azonban vannak hátrányai. Ha túl rövid egy időszelvény, akkor a túl sok váltás lassuláshoz fog vezetni, ha pedig túl hosszú egy intervallum, akkor alacsonyabb időigényű folyamatok el fogják pazarolni a nagyobbak elől a számítási időt. És az utolsó mondatból következik még egy probléma! Ha van egy nagyobb prioritású folyamatunk, akkor ő ki lesz éheztetve, és ez alaposan ellentmond az eredeti feltevésünknek, miszerint semelyik folyamatot nem éheztetjük ki.

Ezek a problémák hivatott segíteni a *prioritásos ütemezés* algoritmusát. Itt minden folyamat kap egy prioritási értéket, ezek alapján csökkenő sorrendbe állítjuk őket, és a legmagasabb prioritású megkezdheti a futását. Azonban, hogy az elején megelőzzünk súlyos problémákat – mint például a végtelen ideig tartó futás – illetve optimalizáljunk is, bevezetjük itt is az időszeleteket. A legmagasabb

prioritású futhat bizonyos ideig, azonban ha lejárt az idő, akkor a következő legmagasabb prioritású kap processzoridőt. A végtelen futás elkerülésének érdekében pedig minden iteráció végén csökkentjük a prioritási értéket, így az eddig legmagasabb prioritású „kevésbé lesz fontos” a következő körben.

A mai modern operációs rendszerek is ilyen alapelvekre épülnek, és azt gondolom, a fenti példák és számolások után senki számára sem kétséges, hogy egy mobiltelefon esetében ezeknek az optimális implementálása komoly kihívást jelent. A helyzet messze nem olyan „egyszerű”, mint az operációs rendszerek nagy testvérei esetében: meg kell találni az arany középutat az energiaigény, hardver kihasználása és teljesítménye között.

## ***A multitasking megvalósítása WP 7.5 platformon***

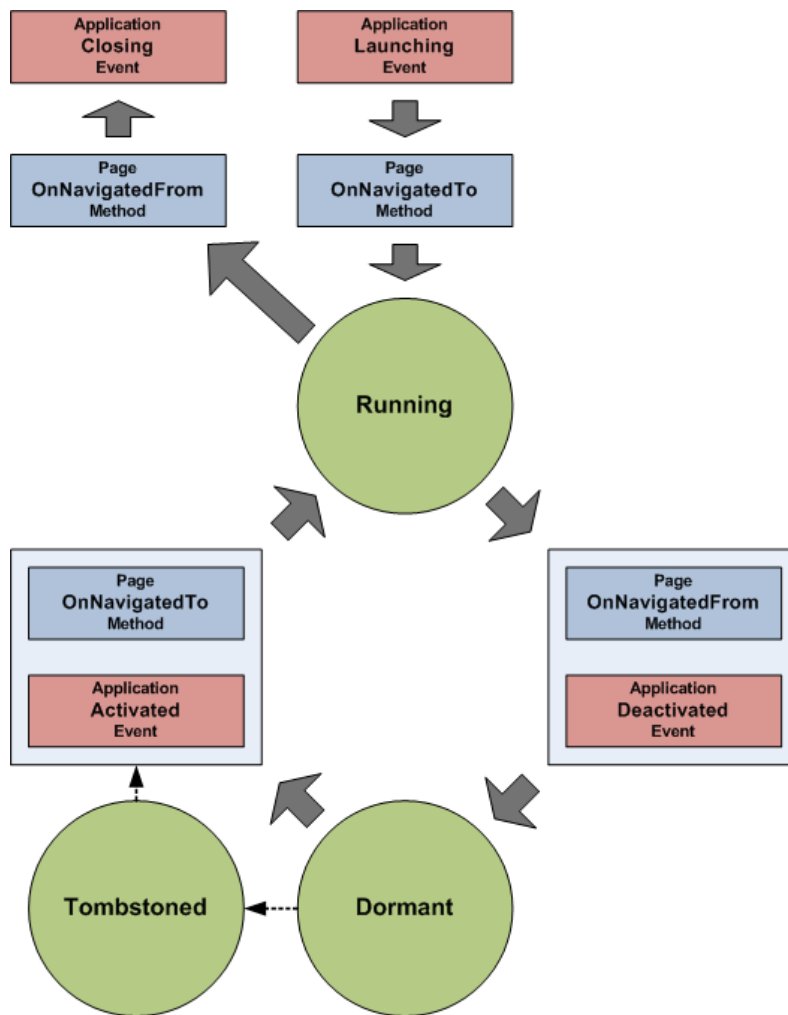
Az eredeti Windows Phone 7 operációs rendszer nem tartalmazta a multitasking lehetőségét (ellenben több szálú alkalmazásokat írhattunk), ez a Mango frissítéssel jött. Azonban gondoljunk csak bele jobban, mit is jelent egy mobiltelefon esetében az, hogy több programot futtat! Nincs tálca, ahol egy szempillantás alatt válthatunk az alkalmazások között, nincsenek ablakok, ahol egymás takarásában futnak a programok külön rajzterületeken. Akkor mit nyerhetünk több program futtatásával?

A választ két részben kapjuk meg. Először is nézzük az eredeti Windows Phone 7 operációs rendszert! Ha bezártunk egy alkalmazást, vagy az alkalmazás valamilyen okból passzív státuszba került (alkalmazásfelületek között navigálunk pl. egy fénykép készítésekor, vagy túl sokáig nem használtuk a telefont, és ő elment pihenni, stb.), akkor valahova el kellett menteni az aktuális állapotot, majd amikor visszatért a felhasználó az alkalmazáshoz, akkor vissza kellett tölteni az előzőleg elmentett adatokat. A válasz másik része pedig még sokkal szembetűnőbb! Böngészünk, és közben szeretnénk zenét hallgatni. A telefonnal érkezett, Microsoft által készített alkalmazások esetében ez eddig is működött!

Mi is azt szeretnénk, hogy a saját alkalmazásunkban lejátszott zene akkor is tovább folytatódjon, ha éppen nem a saját alkalmazásunk aktív vagy a telefon aludni tért. Igen, erre mostantól fogva lehetőségünk van! Nem kell egyszerre látnunk az alkalmazásokat egy kis képernyőn, nem kell ablakban futniuk, ettől még ott lehetnek a háttérben, és ugyanúgy szükségük lehet a processzorra. A Microsoft az eredeti Windows Phone 7 operációs rendszerben nem akarta a telefonok korlátozott hardver erőforrása, energiaigénye és az alkalmazás életciklus-modellje miatt bevezetni a multitaskingot. Ezt azonban előbb-utóbb kénytelen volt implementálni, hiszen olyan képességről beszélünk, amelyet nem lehet kikerülni.

Az 5-1 ábrán látható, hogyan is épül fel egy alkalmazás életciklusa. Az egész egyetlen nagy állapotgéppel írható le, amelynek középpontjában az az állapot áll, amikor az alkalmazás fut. Azonban mi van olyankor, ha nem fut éppen? Több minden is történhet. Megmaradt az előbbiek során ismerttetett állapotmentés mechanizmus is, de ez tovább bővült azzal, hogy a Windows Phone ütemező rendszere eldöntheti, mit tesz az alkalmazásunkkal, és az például egy navigációs művelet során ott maradhat a memóriában.

Az állapotok mentésének a későbbiekben több módját is megnézzük, és láthatjuk majd, hogyan válthatunk gyorsan az alkalmazások között – a multitaskingnak köszönhetően.



5-1. ábra: A Windows Phone 7.5 alkalmazások állapotai

Nézzük meg, hogy az egyes állapotok és események mit jelölnek! A későbbiek során még számos alkalommal hasznosnak bizonyulhat ez a tudás.

- **Launching esemény:** új alkalmazáspéldányt indítottunk, azaz kiválasztjuk a Start felületen vagy az alkalmazáslistából az alkalmazás lapkáját, esetleg egy értesítéshez tartozó üzenet. Ha az alkalmazás logikája nem igényel mást, akkor itt ne folytassunk munkafolyamatot, ez valóban egy teljesen új példány legyen – persze ez alól vannak kivételek. Ez a kódból kezelhető állapotok közé tartozik. Továbbá fontos, hogy itt erőforrásigényes műveletet ne végezzünk, az alkalmazást engedjük betöltődni minél gyorsabban, nagyobb adategységek betöltését mögöttes szálon hajtsuk végre! Vannak olyan alkalmazások, ahol a státusz nem fontos (például egy közösségi oldal kliense, ahova az új híreket az alkalmazás következő indításakor újra le fogjuk tölteni). Azonban gondoljunk például egy képszerkesztő programra, ahonnan véletlen a vissza gombhoz érve kiléptünk. Amikor visszatérünk az alkalmazásba, szeretnénk az előző munkafolyamatot folytatni, így vigyáznunk kell arra, hogy az állapotot folyamatosan mentjük az adatvesztés elkerülése végett. Továbbá vigyázzunk, hogy nagy adatmennyiségeket ne töltsünk be és mentünk ki egyszerre, ezt a feladatot végezzük el egy mögöttes szálon!
- **Running:** az alkalmazás egészen addig „futtatás alatt” állapotban fog maradni, amíg valamilyen felhasználói interakció hatására nem navigálunk valamerre, vagy éppen a felhasználó interakció hiányában a telefon nem tér aludni.
- **OnNavigatedFrom esemény:** akkor játszódik le, ha a felhasználó elnavigál az adott alkalmazásfelületről, vagy az alkalmazás deaktivált állapotba került. Itt el kell mentenünk az



aktuális oldal állapotát, és ha a felhasználó visszatér ide, akkor helyre kell állítanunk. Érdemes még figyelni arra is, hogy ha visszafelé szeretnének navigálni az adott oldalról, akkor nem kell állapotot mentenünk, mivel a következő alkalommal, mikor ide visszatér a felhasználó, újra létrejön a felület. A megvalósítás módja természetesen nagyban függ az alkalmazás logikájától.

- **Deactivated esemény:** ez az állapot akkor jelentkezik, ha a felhasználó átvált egy másik alkalmazásra, vagy megnyomja a Start gombot, illetve ha hosszú ideig nem volt felhasználói interakció, és a telefon zárolódik. Fontos megemlíteni, hogy ez az állapot következik be akkor is, ha valamilyen Chooser segítségével az alkalmazás az operációs rendszer beépített eszközeit szeretné használni, például a kamerát, kép kiválasztását, e-mail üzenet írását, és így tovább. Ennek az eseménynek a hatására mindenképpen el kell mentenünk az alkalmazás állapotát! Ennek módját a későbbiek során részletesen megnézzük.
- **Dormant:** miután a *Deactivated* állapot bekövetkezett, az alkalmazás folyamata és a hozzá tartozó szálak leállításra kerülnek, a rendszer még megpróbálja az alkalmazáshoz tartozó adatokat a memóriában tartani, hogy így azok később gyorsan visszanyerhetők legyenek. A folyamat teljesen automatikus, azonban legyünk vele óvatosak, mivel ha az alkalmazásunk után egy erőforrásigényesebb alkalmazást indít a felhasználó, akkor a mi alkalmazásunk valóban lezárásra kerül! Az esemény hatására le kell mentenünk az állapotát, az eddig elfoglalt memóriaterület pedig felszabadításra kerül.
- **Tombstoned:** a Windows Phone 7 rendelkezik egy **Dictionary<T>** típusú gyűjteménnyel, amelyen keresztül objektumokat menthetünk el. Ennek működését még részletezni fogjuk, azonban a mechanizmus ebben az állapotban nyeri el igazán az értelmét. Az alkalmazás leállításra kerül, az adatokat nem őrzi meg automatikusan a rendszer, de a fent említett gyűjteményt elmenti, így onnan vissza tudjuk majd az állapotot tölteni. Vigyázzunk vele, mert ha a telefont pl. újraindítják, akkor a gyűjtemény elveszik, így ebben az esetben más utat kell választanunk!
- **Activated esemény:** ha *Dormant* vagy *Tombstoned* állapotból tér vissza az alkalmazásunk, akkor az *Activated* esemény fog lejátszódni. Itt tudjuk leellenőrizni, hogy hova volt az állapot elmentve, és ettől függően tudjuk visszaállítani.
- **OnNavigatedTo esemény:** akkor következik be, amikor a felhasználó egy új alkalmazásfelületre navigál, vagy amikor az adott lap az alkalmazás indulása után először kerül betöltésre. Itt szükséges leellenőrizni azt, hogy egy új példányra van-e szükségünk, vagy egy lementett állapotot szeretnénk helyreállítani!
- **Closing esemény:** az alkalmazás lezárása előtti utolsónak lezajló esemény. Nagyon fontos vigyázni arra, hogy itt maximum 10 másodpercig tartózkodhat az alkalmazás, ha ezt túllépi, a folyamatban lévő műveletek befejezése nélkül fog leállni. Itt tehát nem célszerű erőforrásigényes műveleteket használnunk!

## Állapotok kezelése

Az előző részekben alaposan végignéztük, hogy mi minden történik a háttérben, amelyből mi semmit sem látunk. Megnéztük, milyen állapotok között ingázik egy alkalmazás az életciklusa során, és milyen teendőink vannak az egyes esetekben. Egy dolgot azonban még nem láttunk: Hogyan kell ezeket gyakorlatban megvalósítani?

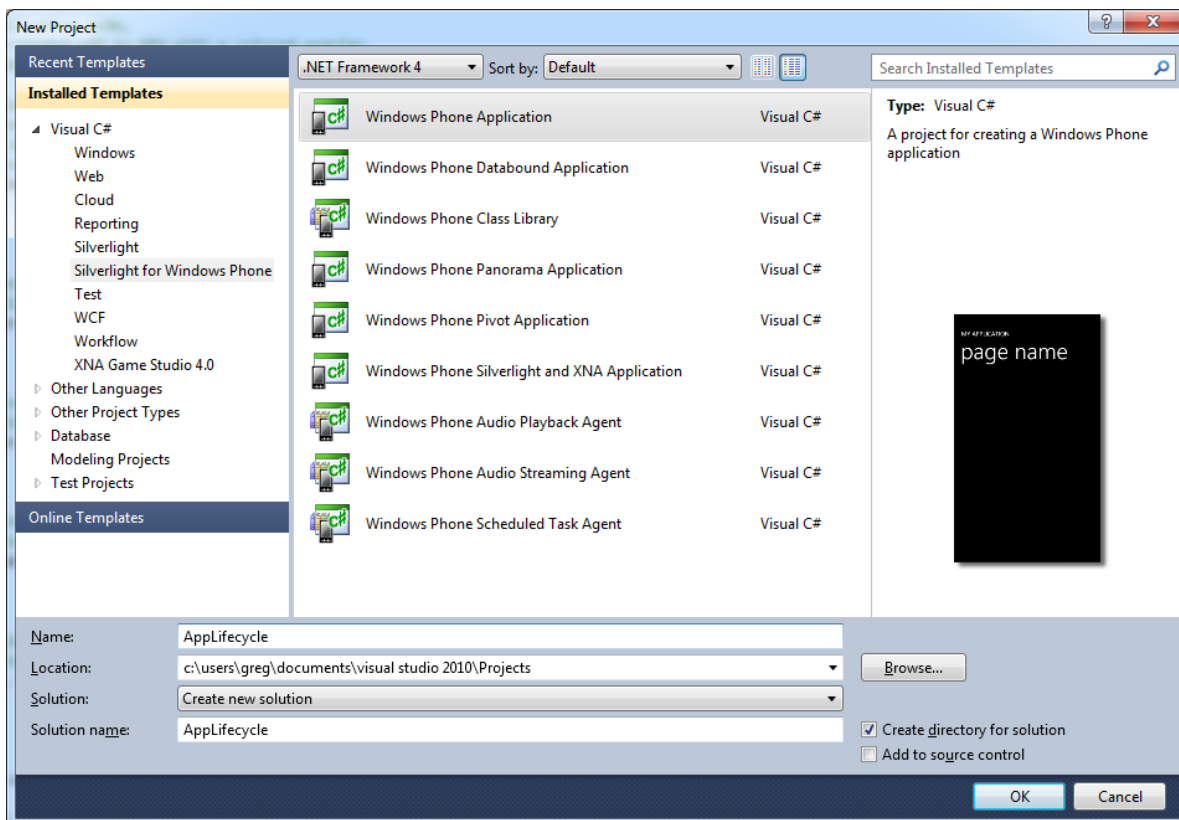
### Életciklus-események kezelése

Az 5-1. ábrán láthattuk, mely eseményekkel kell foglalkoznunk, és az ábrát követő leírásban szerepelt szövegesen is, hogy az egyes eseményeknél mi a tennivalónk. Az áttekinthetőség kedvéért az alábbi táblázatban összefoglalom az eseményeket és a hozzájuk tartozó műveleteket:

Esemény	Végrehajtandó művelet
Launching	Minimális kód, semmi erőforrásigényes műveletet ne végezzünk!
Activated	Állítsuk helyre az állapotot, ha lehet FAS (Fast Application Switching) segítségével!
Deactivated	Mentsük el az állapotot! Ha lezáródott az alkalmazás, akkor biztosítsuk a perzisztens tárolást!
Closing	Utólagos műveleteket végezzünk, vigyázzunk a 10 másodperces időhatárra!
OnNavigatedFrom	Ha nem visszafelé navigálunk, akkor mentsük el az adott felhasználó felülethez tartozó állapotot!
OnNavigatedTo	Amennyiben nem egy új példányra van szükségünk, a tárból állítsuk helyre az alkalmazás állapotát!

Tanács: Egy Windows Phone alkalmazás ne tartalmazzon opciót a kilépésre, ezt a feladatot hagyjuk meg a telefon vissza gombjának!

Próbáljuk ki mindezt a gyakorlatban! Indítsuk el a Visual Studio 2010-et, és hozzunk létre egy új projektet a File ⇒ New ⇒ Project parancs segítségével! A megjelenő dialóguson a C# nyelvű Silverlight for Windows Phone szekcióból válasszuk ki az egyszerű Windows Phone Application sablont, és az alkalmazásnak adjuk az **AppLifecycle** nevet (5-2 ábra)! A fejezet során végig ezt a projektet fogjuk használni.



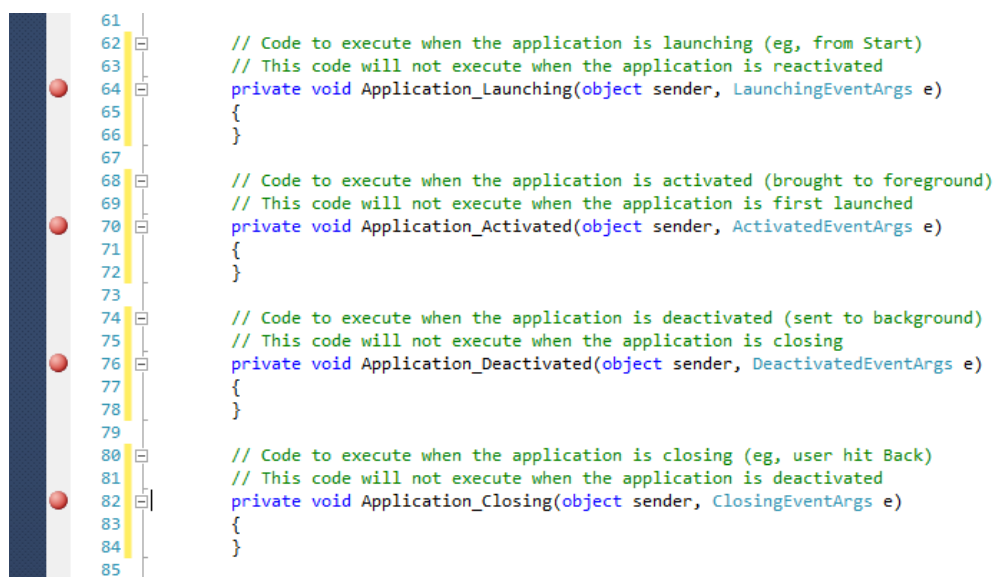
5-2. ábra: A fejezethez szükséges projekt létrehozása

Ha megnyitjuk az **App.xaml.cs** fájlt, akkor láthatjuk, hogy a fenti táblázatból négy különböző eseményvezérlő már előre el lett készítve. Pillanatnyilag még nem tudjuk őket semmilyen hasznos dologra felhasználni, azonban azért, hogy lássuk azok végrehajtási sorrendjét, helyezzünk el minden eseményvezérlőhöz egy töréspontot, ahogyan az 5-3. ábrán is látható! A megjelenő felületen válasszuk a

Windows Phone 7.1 beállítást és hagyjuk jóvá! Ha ezzel megvagyunk, akkor hajtsuk végre a következő lépéseket, és figyeljük meg, hogy az egyes esetekben hol áll meg a kód végrehajtása!

- Az F5 segítségével Debug üzemmódban futtassuk az alkalmazást! Ennek hatására elindul az emulátor.
- A kód végrehajtása megáll az **Application\_Launching** eseménykezelőnél. Ez a működés annak tudható be, hogy az alkalmazásnak ez volt az első elindítása.
- Ezután az emulátoron nyomjuk le a Start gombot! Ennek hatására úgy tűnik, mintha az alkalmazásunk kilépett volna.
- A Visual Studio most az **Application\_Deactivated** eseménykezelőn elhelyezett töréspont hatására megáll. Az alkalmazás állapota megváltozott, és ha valamilyen logika már megvalósításra került volna, akkor itt kellene a állapotot mentenünk.
- Két eseményünkkel még nem találkoztunk, ezért F5 segítségével újra indítsuk el az alkalmazást! Az újra meg fog állni a **Launching** eseménynél. Léptessük tovább!
- Most az emulátoron nyomjuk le a Back (vissza) gombot, és figyeljük meg, hogy mi történik!
- Megállt a végrehajtás az **Application\_Closing** eseménykezelőnél. Ez azt jelenti, hogy innentől fogva 10 másodpercünk van az alkalmazás állapotának elmentésére, utána végérvényesen kilépünk!

Az **Application\_Activated** eseménykezelő egyszer sem került meghívásra. Ezt az okozta, hogy az alkalmazást egyszer sem futtattuk le az emulátoron található példány újrafuttatásával. Ezt hiába is próbáltuk volna meg, a Visual Studio hibakereső elengedi az alkalmazáspéldányunkat, miután megnyomtuk a Back vagy Start gombok valamelyikét.



5-3. ábra: töréspontok elhelyezése az App.xaml.cs fájlban

Egyelőre nem sok minden történt, mindössze láttuk, hogy melyik esemény mikor játszódik le. A következő részben alkalmazásunkat úgy alakítjuk át, hogy az elmentse a legutóbbi állapotot és a következő indításkor visszaolvassa azt.

## Az alkalmazás állapotának mentése és helyreállítása

Ha az alkalmazásunk adatorientált, és a felhasználó által tett módosításokat el kell tárolnia, akkor a memóriában található objektumokat perzisztens tárolóba kell elmentenünk. Erre többféle lehetőségünk van (sőt, van ahol egy lehetőségen belül is több irányba ágazik el az út). Először is válasszuk szét a műveletet az alapján, hogy az alkalmazás véglegesen leállt-e, vagy csak szüneteltetésre került valamilyen formában! Tudjuk, hogy a szüneteltetésnek két módja is van. Ha *Dormant* állapotba került a telefon, azaz

minden hozzá tartozó adat a memóriában maradt, akkor rendkívül gyorsan folytatható a munkafolyamat, manuálisan semmit sem kell helyreállítanunk. Ekkor kizárólag azt kell megvizsgálnunk, hogy használható-e a Fast Application Switching mechanizmus. Ezt az **Application\_Activated** eseményvezérlőn belül tesszük meg:

```
private void Application_Activated(object sender, ActivatedEventArgs e)
{
    if ( e.IsApplicationInstancePreserved )
    {
        // dormant állapotban voltunk, nem kell semmit sem tennünk manuálisan
    }
    else
    {
        // sajnos nem dormant volt, vissza kell hoznunk az adatokat
    }
}
```

Ha nem *Dormant* állapotban volt az alkalmazásunk, akkor nekünk kell gondoskodnunk az adatok mentéséről. Erre egyik lehetőségünk a **Tombstoning** állapotban használható **PhoneApplicationService**, amely tartalmaz egy **Dictionary<T>** típusú generikus gyűjteményt, és az ebben eltárolt objektumokat a telefon automatikusan elmenti, amikor a *Deactivated* esemény lejátszódik. Ennek nagy előnye, hogy szintén gyorsan visszanyerhetjük az adatokat, hátránya viszont, hogy ez nem teljesen perzisztens megoldás. Ha a telefont újraindítjuk, vagy egy frissítés miatt az alkalmazásunk véglegesen leáll, akkor az adatai elvesznek. Erre jelenthet megoldást az adatok szerializálása az *Isolated Storage*-ba. Miután közvetlen fájlrendszer hozzáférésünk nincs, de mégis a telefon tárhelyét szeretnénk használni adatok mentésére, így ő az egyik megoldás. Számos szerializációs módszert használhatunk (bináris, JSON, XML, **DataContract**, stb.), azonban ezek közül egyik sem tud sebességben versenyezni az előző megoldásokkal. Másik lehetőség lenne a Mangóban megjelent SQL CE adatbázismotor használata – ezt részletesebben az adatkezeléssel foglalkozó fejezet tárgyalja.

Alakítsuk át az alkalmazásunkat úgy, hogy az kilistázza a telefon színtémáit, és ha a felhasználó kiválaszt egyet, akkor azt láthatóvá teszi!

Sajnos, ténylegesen a telefon témáját nem tudjuk kódból változtatni (legalábbis nem legálisan), talán majd egy következő Windows Phone verzióban.

Hozzunk létre egy új osztályt, amely a színek nevét és hexadecimális kódját tárolja! Ahhoz, hogy az osztályból létrehozott példányokat menteni tudjuk (*Isolated Storage* vagy a telefon beépített **PhoneApplicationService** megoldása segítségével), annak sorosíthatónak kell lennie, mégpedig a **DataContractSerializer** segítségével. Az ehhez szükséges objektumok Windows Phone esetében is implementálásra kerültek, semmi más nem kell tennünk, mint felvenni a **System.Runtime.Serialization** assembly-t a könyvtár referenciák közé!

```
[DataContract]
public class PhoneTheme
{
    [DataMember]
    public string Name { get; set; }

    [DataMember]
    public string HexValue { get; set; }
}
```

Szükségünk lesz egy egyszerű felhasználói felületre is, amely mindösszesen egy listát fog tartalmazni a **ListBox** vezérlővel megvalósítva. Miután saját entitásokkal dolgozunk, a vezérlő alapesetben nem lenne alkalmas azok megjelenítésére, így egy sablon segítségével átalakítjuk. Ennek hatására a háttér a tárolt szín lesz, és minden listaelem közepén szövegesen is megjelenik majd az entitás neve:

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
  <ListBox x:Name="accentList">
    <ListBox.Resources>
      <DataTemplate x:Key="AccentItemTemplate">
        <Grid Width="400" Height="100" Margin="28,0,0,20"
          Background="{Binding HexValue}">
          <Border BorderBrush="White" BorderThickness="2"/>
          <TextBlock HorizontalAlignment="Center"
            Text="{Binding Name}" VerticalAlignment="Center"
            FontSize="24" />
        </Grid>
      </DataTemplate>
    </ListBox.Resources>
    <ListBox.ItemTemplate>
      <StaticResource ResourceKey="AccentItemTemplate"/>
    </ListBox.ItemTemplate>
  </ListBox>
</Grid>
```

Most már van olyan entitás osztályunk, amely képes az elemeket tárolni, és van hozzá egy felhasználói felületünk, amely megjeleníti azokat. Hozzuk létre a tényleges listánkat is! Egyszerű esettel van dolgunk, elegendő lesz egy **List<T>** generikus listapéldány, amelyet helyben inicializálunk is az új elemek megadásával. Nyissuk meg a **MainPage.xaml.cs** fájlt, és hozzuk létre a következő adattagot:

```
private List<PhoneTheme> themes = new List<PhoneTheme>
{
    new PhoneTheme { Name = "Blue", HexValue = "#FF1BA1E2" },
    new PhoneTheme { Name = "Brown", HexValue = "#FFA05000" },
    new PhoneTheme { Name = "Green", HexValue = "#FF339933" },
    new PhoneTheme { Name = "Lime", HexValue = "#FFA2C139" },
    new PhoneTheme { Name = "Magenta", HexValue = "#FFD80073" },
    new PhoneTheme { Name = "Mango", HexValue = "#FFF09609" },
    new PhoneTheme { Name = "Pink", HexValue = "#FFE671B8" },
    new PhoneTheme { Name = "Purple", HexValue = "#FFA200FF" },
    new PhoneTheme { Name = "Red", HexValue = "#FFE51400" },
    new PhoneTheme { Name = "Teal", HexValue = "#FF00ABA9" },
};
```

Mielőtt elsiklanánk felette, szükségünk lesz egy metódusra, amely a szövegesen megadott színek kódokat úgy alakítja át, hogy a Silverlight azt megjeleníthesse. De mit is tud a Silverlight megjeleníteni? Hiszen XAML kódban is névvel vagy hexadecimális kóddal adjuk meg a színeket, itt miért nem jó ez? A XAML-ben megadott kódot az objektum képes interpretálni, és a szükséges változtatásokat elvégezni automatikusan, azonban erre nincs lehetőségünk mögöttes kódból. Itt a **Brush** alaposztály valamelyik leszármazottját kell segítségül hívnunk, amely a mi esetünkben a **SolidColorBrush** lesz, ez egyszínű kitöltést tesz lehetővé. Ennek az objektumnak a konstruktora egy **Color** típusú objektumpéldányt vár, így a szövegesen megadott színek kódunkat ilyen típusra kell átalakítanunk. Ennek az elvégzése rendkívül egyszerű:

```
private SolidColorBrush GetColorFromHex( string color )
{
    return new SolidColorBrush(
        Color.FromArgb(
            Convert.ToByte( color.Substring( 1, 2 ), 16 ),
            Convert.ToByte( color.Substring( 3, 2 ), 16 ),
```

```
        Convert.ToByte( color.Substring( 5, 2 ), 16 ),  
        Convert.ToByte( color.Substring( 7, 2 ), 16 )  
    );  
}
```

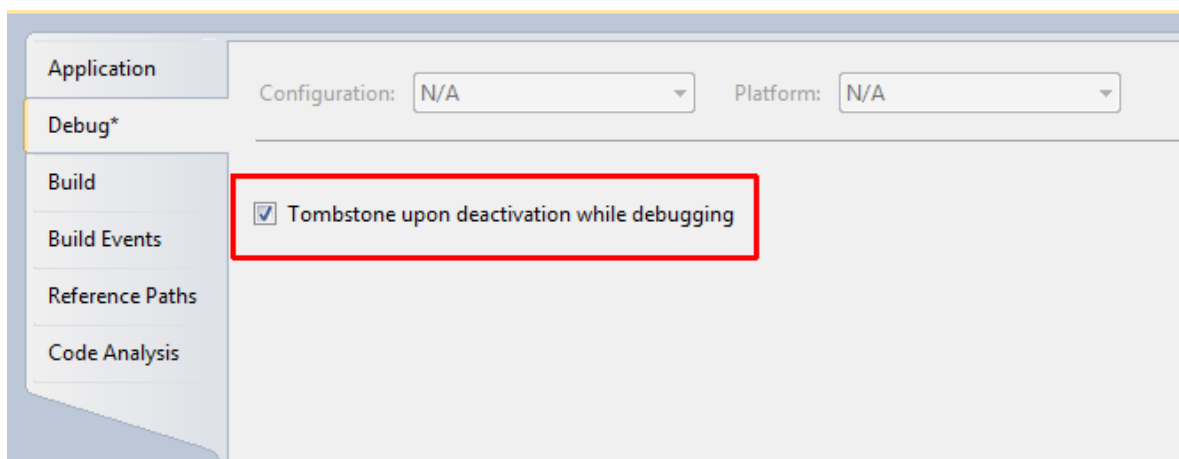
Kattintsunk tervezői nézetben kettőt a telefon felületén, és a Visual Studio létrehozza a **Loaded** eseménykezelőt. Ebben első dolgunk az, hogy az előbb létrehozott listát átadjuk a felületen elhelyezett **ListBox** vezérlőnek. Ezt könnyedén megtehetjük, mivel a **ListBox** **ItemsSource** tulajdonságának a feltöltése után a Silverlight adatkötő motor automatikusan kivezeti a listaelemeket a felületre, ahol a vezérlő a nemrég létrehozott sablon segítségével meg is jeleníti azokat. Itt viszont lesz még egy feladatunk! Ha el volt mentve az állapot, akkor azt itt helyre kell állítanunk! Először nézzük ezt meg a telefon beépített eszközével megvalósítva, amelyet a **PhoneApplicationService** osztályon keresztül érhetünk el! Ennek az osztálynak van egy **State** nevezetű gyűjteménye, amelyen megvizsgáljuk, hogy tartalmazza-e a „Theme” nevű kulcsot. Ha igen, akkor nem először használjuk az alkalmazást, és korábban mentettünk már állapotot. Nincs is más dolgunk, mint kiolvasni a kulcshoz tartozó értéket és a felületen átállítani az adott színre a háttérrel:

```
void MainPage_Loaded(object sender, RoutedEventArgs e)  
{  
    accentList.ItemsSource = themes;  
  
    if ( PhoneApplicationService.Current.State.ContainsKey( "Theme" ) )  
        LayoutRoot.Background = GetColorFromHex(  
((PhoneTheme)PhoneApplicationService.Current.State[ "Theme" ]).HexValue );  
}
```

Valahol az előbb betöltött objektumot el is kellett mentenünk, erre a legmegfelelőbb hely a listában kiválasztott elem változtatását jelző esemény. Ha a tervezői nézetben duplán kattintunk a **ListBox** vezérlőre, akkor a Visual Studio létrehozza a **SelectionChanged** eseménykezelőt, ezen belül dolgozunk tovább. Le kell kérdeznünk az aktuálisan kiválasztott elemet. Mivel az adatkötő motor felhasználásával a tényleges objektumpéldányokat adtuk át a **ListBox**-nak, így lekérdezhetjük magát az elemet, de ugyanúgy megfelelne az index is, mivel a forráslistát az osztály továbbra is tárolja:

```
private void accentList_SelectionChanged(object sender, SelectionChangedEventArgs e)  
{  
    PhoneTheme theme = accentList.SelectedItem as PhoneTheme;  
    LayoutRoot.Background = GetColorFromHex( theme.HexValue );  
  
    PhoneApplicationService.Current.State[ "Theme" ] = theme;  
}
```

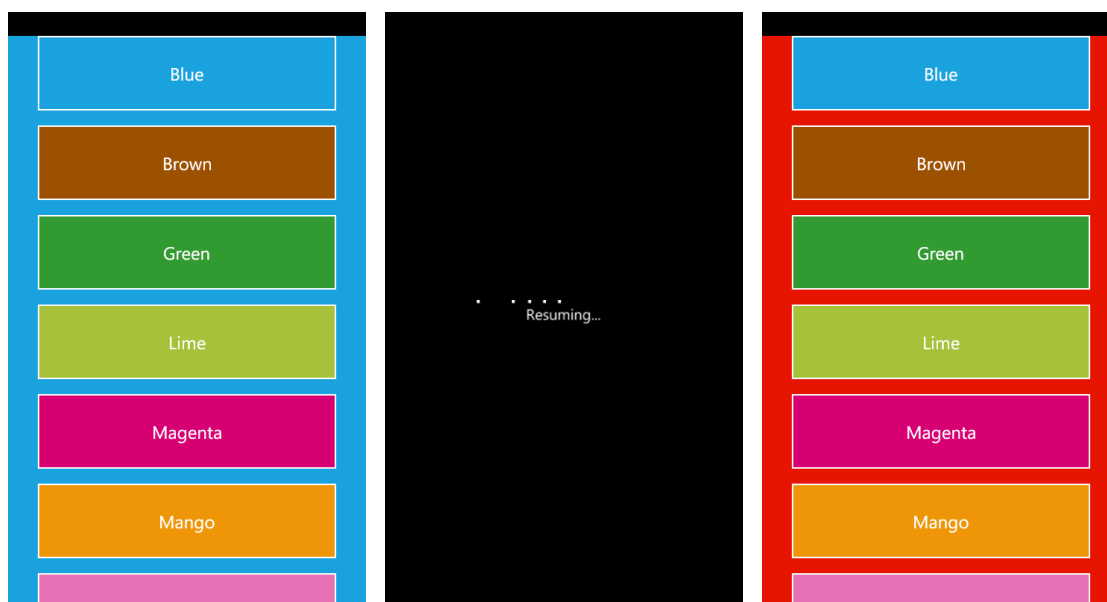
Mielőtt futtatnánk az alkalmazást, tennünk kell a beállítások között is egy változtatást. A Mangóban a multitasking bevezetésével az emulátor is megváltozott. Ez alapértelmezésben nem engedélyezi a **Tombstoning** állapot tesztelését. Ahhoz, hogy ezt engedélyezzük, jobb egérgombbal kattintsunk a projekt nevére, és válasszuk a Properties opciót! Ezen belül válasszuk a Debug fület, és az egyetlen itt található „Tombstone upon deactivation while debugging” opciót jelöljük be! Ezek után futtathatjuk az alkalmazást.



**5-4. ábra: Alvó állapot engedélyezése Visual Studióban**

A tesztelést a következő lépésekben végezzük el:

- Indítsuk el Debug módban az emulátort és benne az alkalmazást!
- Válasszunk ki egy színt a listából, majd a Start gomb segítségével lépünk ki!
- Itt be tudjuk tölteni az Internet Explorert vagy egy másik, az emulátorra telepített alkalmazást.
- Lépünk ki a vissza gombbal, és térjünk vissza a saját alkalmazásunkhoz!
- Az állapota megfelelően helyre lesz állítva.
- (Ha nem a **Resuming** állapot következik be, akkor zárjuk be az emulátort, és fordítsuk újra a projektet, majd futtassuk újra!



**5-5. ábra: Az alkalmazás futásának pillanatfelvételei**

A fenti módszer nem menti el az adatokat végérvényesen. Ha valóban perzisztálni szeretnénk azokat, akkor az Isolated Storage-ba kell mentenünk. Mivel általánosítható megoldást szeretnénk mutatni és nem csak az alkalmazásra specifikus, így az adatsorosítást választottam. Az entitásosztályt fel kell készíteni **DataContractSerializer**rel való együttműködésre, most ezt ki is próbáljuk. Szükségünk van egy alkalmazásszintű statikus példányra a **PhoneTheme** osztályból, a memóriában ide fogjuk menteni a felületen végzett változtatásokat, és ide fogjuk majd visszatölteni azokat. Ezenkívül két metódusra is szükségünk van még, az egyik feladata az adatok mentése, a másiké pedig azok visszatöltése.



```
public static PhoneTheme Theme { get; set; }

private void SaveData()
{
    using ( IsolatedStorageFile file = IsolatedStorageFile.GetUserStoreForApplication() )
    {
        using ( IsolatedStorageFileStream stream = file.CreateFile( "theme.xml" ) )
        {
            // új DataContractSerializer példány, a megfelelő típussal ellátva
            DataContractSerializer serializer =
                new DataContractSerializer( Theme.GetType() );
            // az objektum sorosítása a megnyitott adatfolyamra
            serializer.WriteObject( stream, Theme );
        }
    }
}

private void LoadData()
{
    using ( IsolatedStorageFile file = IsolatedStorageFile.GetUserStoreForApplication() )
    {
        // vizsgáljuk meg létezik-e már a fájl
        if ( !file.FileExists( "theme.xml" ) )
            Theme = new PhoneTheme { Name = "Blue", HexValue = "#FF1BA1E2" };
        else
            using ( IsolatedStorageFileStream stream = file.OpenFile( "theme.xml",
                FileMode.Open, FileAccess.Read ) )
            {
                // létezett a fájl, így megfelelő típussal visszaolvassuk az adatokat.
                DataContractSerializer serializer = new DataContractSerializer(
                    typeof( PhoneTheme ) );
                Theme = (PhoneTheme)serializer.ReadObject( stream );
            }
    }
}
```

A beolvasás esetére megvizsgáltuk, hogy létezik-e már a fájl, így a **LoadData()** metódus is nyugodtan használható. Nincs is más dolgunk, mint az alkalmazás indításakor és leállításakor meghívni az oda tartozó metódusokat (nem szükséges megkülönböztetni, hogy milyen típusú indításról és leállításról van szó).

```
private void Application_Launching(object sender, LaunchingEventArgs e)
{
    LoadData();
}

private void Application_Activated(object sender, ActivatedEventArgs e)
{
    if ( e.IsApplicationInstancePreserved )
    {
    }
    else
    {
        LoadData();
    }
}

private void Application_Deactivated(object sender, DeactivatedEventArgs e)
{
    SaveData();
}
```



```
private void Application_Closing(object sender, ClosingEventArgs e)
{
    SaveData();
}
```

A felhasználó felület mögöttes kódja az alábbiak szerint módosul, felhasználva az alkalmazásszinten elhelyezett statikus objektumpéldányt:

```
void MainPage_Loaded(object sender, RoutedEventArgs e)
{
    accentList.ItemsSource = themes;

    LayoutRoot.Background = GetColorFromHex( App.Theme.HexValue );
}

private void accentList_SelectionChanged(object sender, SelectionChangedEventArgs e)
{
    PhoneTheme theme = accentList.SelectedItem as PhoneTheme;
    LayoutRoot.Background = GetColorFromHex( theme.HexValue );

    App.Theme = theme;
}
```

## Folyamatok a háttérben

A fejezet elején említettem, hogy a Windows Phone 7 nem engedte az alkalmazásoknak, hogy amikor nem ők voltak előtérben, a háttérben műveleteket végezzenek. Részletesen leírtam, hogyan működik a multitasking, és ennek hiányában miért nem tehette az operációs rendszer ezeket a funkciókat elérhetővé.

### Elméleti áttekintés

Alapesetben egy alkalmazásra úgy tekintünk, hogy annak az előtérben kell futnia. Azonban vannak olyan esetek, amikor szeretnénk, hogy olyankor is történjen valami, amikor a felhasználó éppen más alkalmazást futtat, vagy egyáltalán nem is használja a telefonját. Tipikus eseteket is felsorolhatunk:

- Van egy zenelejátszó alkalmazásunk. A felhasználó valószínűleg nem fogja folyamatosan a kezében a telefonját, és nem használja aktívan, lehet, hogy közben egy könyvet olvas, vagy éppen a munkáját végzi. Ilyenkor a telefon alvó állapotba helyezné a lejátszónkat, és a zene lejátszása megszakadna. Szeretnénk, ha a zene lejátszása folytatódna a háttérben.
- Főzéshez készítünk egy időmérő alkalmazást, amelynek az a feladata, hogy bizonyos idő letelte után riasszon bennünket, így biztosan nem felejtjük ott tovább az ételt, és az nem fog odaégni. Főzés közben biztosan nem használjuk a telefonunkat, így ebben az esetben is kellemetlen lenne, ha alvó állapotba kerülne az alkalmazás. Megjegyzésként megemlíteném, akár azt is megoldhatjuk, hogy a telefon ne tudjon aludni, amíg manuálisan ki nem lépünk az alkalmazásból. Ezzel azonban az akkumulátort nagyon hamar lemeríthetjük, így használata nem javasolt.
- Időjárás-figyelő alkalmazást készítünk, amelynek extra funkciója, hogy vihar közeledése esetén időben riaszt bennünket. A funkció riasztáshoz kapcsolódó részére pontosan azok az elvek érvényesek, amit az előző listapontban említettem, viszont itt még valamivel számolnunk kell. Az időjárás-adatokat nem elegendő akkor lekérnünk egy webes szolgáltatástól, amikor a felhasználó az alkalmazásunkat használja, mert így nem biztos, hogy mindig a legfrissebb információkkal fogunk rendelkezni! Szinkronizálnunk kell a háttérben, ez pedig időről időre adatok letöltését jelenti – a felhasználó beavatkozása nélkül is.

A fenti példákat szándékosan választottam, még találkozni fogunk velük az alfejezetek során, ahol részletesen bemutatom ezeket az eseteket.

A Windows Phone Mango egy energiatakarékos módját választotta a háttérműveletek kezelésének. Előre pontosan meghatározott feladatkör szerint dolgozhat alkalmazásunk a háttérben, és akkor is csak rövid ideig. Ha túl alacsony a telefon akkumulátorának a töltöttségi szintje, vagy ha a felhasználó akarja, akkor a háttérben élő folyamatok futtatási joga megvonható az alkalmazástól. Természetesen akkor is kikapcsolásra fognak kerülni, amikor az alkalmazásunk előtérben fut, így „aktív” állapotban más módot kell találnunk az egyébként háttérben futó feladatok ellátására. Ezeknek a folyamatoknak a neve *Background Agents*, és ez csak gyűjtőnév, a valós feladatok során ezek több speciális változatát vehetjük használatba. Ezeket az ágenseket **Task** típusok gyűjtik magukba, melyek közül kettőt rögtön érdemes megemlíteni:

- **PeriodicTask**: periodikusan (például minden 30. percben) meghívódik, és valamilyen gyorsan végrehajtható feladatot lát el.
- **ResourceIntensiveTask**: nagyobb számítási feladatok ellátására szolgál (pl. adatok tömörítése/kitömörítése), legfeljebb 10 percig futhat.

Az ágensek létrehozása során többször találkozunk olyan esettel, amikor külön projektben (és ezért külön .dll állományban) helyezkedik el az ágens feladatát leíró kód, amely az a **WPAppManifest.xml** fájlban keresztül kapcsolódik az alkalmazáshoz. Álljon itt egy példa ennek az állománynak a szerkezetére:

```
<ExtendedTask Name="BackgroundTask">
<BackgroundServiceAgent Specifier="ScheduledTaskAgent"
Name="LocationTaskAgent" Source="LocationTaskAgent"
Type="LocationTaskAgent.ScheduledAgent" />
</ExtendedTask>
```

### Zenelejátszás integrálása saját alkalmazásokba

Az összes népszerű mobil operációs rendszerre számos zene- és videólejátszó szoftver létezik. Sajnálatosan Windows Phone-ra eddig egyáltalán nem volt lehetőségünk saját lejátszók készítésére, de még a beépített Zune lejátszóval sem tudtuk integrálni alkalmazásainkat. A Windows Phone Mango megjelenésével szerencsére ez gyökeresen megváltozott! A multitasking bevezetésével a Microsoft létrehozta a háttérben futó folyamatokat. A zene lejátszásáért felelős ágens mélyen az operációs rendszerben található Zune komponensbe van integrálva, így használhatjuk a hangerővezérlés hardveres gombjait, vagy a telefon zárolást feloldó képernyőjén található zenelejátszás kezelőfelületet is. Amikor alkalmazásunk zenét játszik le, az adott audió bekerül a Zune *Media Queue*-ba, így ugyanabban a bánásmódban részesül, mint bármelyik zenénk a médiagyűjteményben. A Mango megjelenésével nemcsak egy ilyen ágens, hanem rögtön kettőt is kapunk. Az **AudioPlayerAgent** egy listát vár zenéinkből, melyek helyben tároltak vagy akár egy távoli címen elérhetőek is lehetnek. Itt vigyáznunk kell arra, hogy a telefon számára fogyasztható audio formátumokkal kódolt zenei állományokra hivatkozzunk. Az **AudioStreamingAgent** esetében mi írhatjuk meg azt a komponenst, amely a zenei információt képes dekódolni, és így olyan forrásadattal dolgozhatunk, amilyennel csak szeretnénk. A következő mintaalkalmazásban az **AudioPlayerAgent**-et használjuk fel, így hasznos, ha most az elején elő is készítünk néhány MP3 zeneszámot a kedvenceink közül!

Adjunk hozzá meglévő alkalmazásunkhoz egy új lapot! A projekt gyorsmenüjéből válasszuk ki az Add ⇒ New Item ⇒ Windows Phone Portrait Page parancsot, és nevezzük el a fájlt **AudioPlayerPage.xaml**-nek! A felületet az alábbi kódok szerint alakítsuk ki, az csak egy lejátszás, egy előre- és egy hátratekerés gombot tartalmaz:

```
<StackPanel x:Name="TitlePanel" Grid.Row="0" Margin="12,17,0,28">
    <TextBlock x:Name="ApplicationTitle" Text="BACKGROUND AUDIO PLAYER"
        Style="{StaticResource PhoneTextNormalStyle}"/>
    <TextBlock x:Name="PageTitle" Text="play a song" Margin="9,-7,0,0"
        Style="{StaticResource PhoneTextTitle1Style}"/>
</StackPanel>

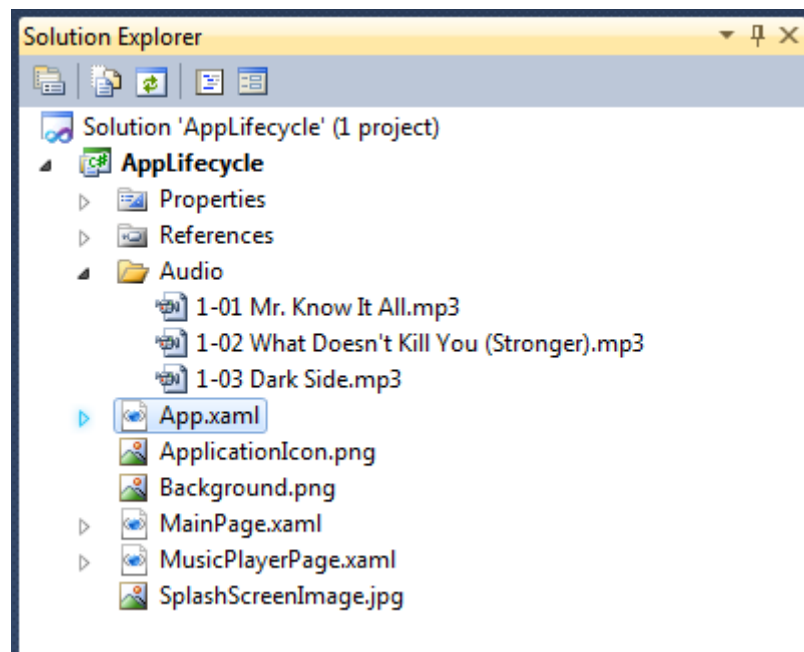
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
```

```

<StackPanel Orientation="Horizontal" Width="420" Margin="18,40,18,0"
    VerticalAlignment="Top">
    <Button Content="prev" x:Name="prevButton" Height="140" Width="140"
        Click="prevButton_Click"/>
    <Button Content="play" x:Name="playButton" Height="140" Width="140"
        Click="playButton_Click"/>
    <Button Content="next" x:Name="nextButton" Height="140" Width="140"
        Click="nextButton_Click"/>
</StackPanel>
<TextBlock x:Name="txtCurrentTrack" Height="75" HorizontalAlignment="Left"
    Margin="12,193,0,0" VerticalAlignment="Top"
    Width="438" TextWrapping="Wrap" />
</Grid>

```

Hozzunk létre egy új mappát és másoljuk bele az MP3 fájlokat, amiket szeretnénk majd lejátszani! Jelöljük ki a zenefájlokat, jobb kattintással rajtuk a Properties parancsot választva állítsuk be a Copy To Output Directory tulajdonságot a Copy if newer opcióra! Az én projekt struktúráám az 5-6. ábrán látható, ehhez hasonló eredményt kell kapnunk.



5-6. ábra: A projekt struktúrája

Következő tennivalónk a zenefájlok átmásolása az Isolated Storage-ba.

Fontos megemlíteni, hogy az **AudioPlayerAgent** csak Isolated Storage-ban és távoli elérési úton elhelyezett audio fájlokat tud lejátszani, a Zune gyűjteményünkhöz nincs hozzáférése, így Zune-on keresztül a telefonra másolt zenéket ne is próbáljuk meg elérni!

Az **App.xaml.cs** fájlban helyezzük el az alábbi metódust, és ne felejtsük azt az **App** osztály konstruktorából meghívni! Természetesen a zeneszámokat tartalmazó fájlok listáját mindenkinek a saját fájljaira kell szabnia.

```
using System.IO.IsolatedStorage;
using System.Windows.Resources;

private void CopyToIsolatedStorage()
{
    using (IsolatedStorageFile storage =
        IsolatedStorageFile.GetUserStoreForApplication())
    {
        string[] files = new string[] { "1-01 Mr. Know It All.mp3",
                                         "1-02 What Doesn't Kill You (Stronger).mp3",
                                         "1-03 Dark Side.mp3" };

        foreach (var _fileName in files)
        {
            string _filePath = "Audio/" + _fileName;
            if (!storage.FileExists(_filePath))
            {
                StreamResourceInfo resource = Application.GetResourceStream(
                    new Uri(_filePath, UriKind.Relative));

                using (IsolatedStorageFileStream file =
                    storage.CreateFile(_fileName))
                {
                    int chunkSize = 4096;
                    byte[] bytes = new byte[chunkSize];
                    int byteCount;

                    while ((byteCount = resource.Stream.Read(bytes, 0, chunkSize))
                        > 0)
                    {
                        file.Write(bytes, 0, byteCount);
                    }
                }
            }
        }
    }
}
```

Van már tehát lejátszható zeneszámunk. Még felhasználói felület is van hozzá, azonban még nem működik az alkalmazásunk, több dolog is hiányzik ehhez! Először is, ha futtatjuk emulátorban a projektet, akkor nem a megfelelő felület jelenik meg. Ehhez be kell állítani, hogy melyik lap (**Page** objektum) legyen az alkalmazás indításakor az alapértelmezett. Ezt úgy tudjuk megtenni, hogy a projekten belül található **Properties** mappában az ott lévő **WPAppManifest.xml** fájlban átírjuk a szükséges hivatkozást a következőre:

```
<Tasks>
  <DefaultTask Name = "_default" NavigationPage="MusicPlayerPage.xaml"/>
</Tasks>
```

Ezek után, ha futtatjuk, akkor már a megfelelő felületet fogjuk látni. Azonban még mindig nem működik! Van három gombunk, az ezekhez tartozó funkcionalitást meg kell írunk! A **BackgroundAudioPlayer** egy *singleton* osztály, így azt példányosítás nélkül a megfelelő névtér feloldása után használhatjuk is. Ennek az osztálynak a műveleteit tudjuk arra felhasználni, hogy a zeneszámok között váltogassunk, megállítsuk vagy elindítsuk az aktuális zeneszámot, esetleg lekérdezzük a zenelejátszó státuszát.

A következőkben a feladatunk az, hogy ezeket a műveleteket fel is használjuk. Figyelnünk kell a zenelejátszás státuszára is, ennek függvényében változtatjuk a lejátszás/szüneteltetés gomb tartalmát, illetve az aktuális zeneszámról megjelenített információt. Mivel a zene lejátszása az alkalmazásból való kilépéskor is folytatódik, így vigyáznunk kell arra is, hogy az alkalmazás újbóli előtérbe kerülése esetén a gombok megfelelő állapotban jelenjenek meg a felületen.

```

using System.Windows.Navigation;
using Microsoft.Phone.BackgroundAudio;

private void prevButton_Click(object sender, RoutedEventArgs e)
{
    BackgroundAudioPlayer.Instance.SkipPrevious();
}

private void playButton_Click(object sender, RoutedEventArgs e)
{
    if (PlayState.Playing == BackgroundAudioPlayer.Instance.PlayerState)
    {
        BackgroundAudioPlayer.Instance.Pause();
    }
    else
    {
        BackgroundAudioPlayer.Instance.Play();
    }
}

private void nextButton_Click(object sender, RoutedEventArgs e)
{
    BackgroundAudioPlayer.Instance.SkipNext();
}

// ezt a konstruktorban helyezzük el!
BackgroundAudioPlayer.Instance.PlayStateChanged += new EventHandler(Instance_PlayStateChanged);

void Instance_PlayStateChanged(object sender, EventArgs e)
{
    switch (BackgroundAudioPlayer.Instance.PlayerState)
    {
        case PlayState.Playing:
            playButton.Content = "pause";
            break;

        case PlayState.Paused:
        case PlayState.Stopped:
            playButton.Content = "play";
            break;
    }

    if (null != BackgroundAudioPlayer.Instance.Track)
    {
        txtCurrentTrack.Text = BackgroundAudioPlayer.Instance.Track.Title + " by " +
            BackgroundAudioPlayer.Instance.Track.Artist;
    }
}

protected override void OnNavigatedTo(NavigationEventArgs e)
{
    if (PlayState.Playing == BackgroundAudioPlayer.Instance.PlayerState)
    {
        playButton.Content = "pause";
        txtCurrentTrack.Text = BackgroundAudioPlayer.Instance.Track.Title + " by " +
            BackgroundAudioPlayer.Instance.Track.Artist;
    }
    else
    {
        playButton.Content = "play";
        txtCurrentTrack.Text = "";
    }
}

```

Ha fordítjuk és futtatjuk a projektet, azt tapasztaljuk, hogy a zenék lejátszása még mindig nem működik. Ez nem is csoda, hiszen egy nagyon fontos komponens még hiányzik! Ez nem más, mint maga a zenét lejátszó ágens. Pótoljuk a hiányosságot, és a Solution Explorerben adjuk hozzá egy új projektet jelenlegi megoldásunkhoz! Válasszuk ki a projekt gyorsmenüjéből az Add ⇒ New Project parancsot, majd projekt típusaként a Windows Phone Audio Playback Agent-et válasszuk! Adjuk neki az **AudioPlayer** nevet! A Visual Studio létrehozza a projektet és abban egy **AudioPlayer.cs** fájlt, amelyben dolgozni fogunk.

Mielőtt továbbhaladnánk, adjuk hozzá az új projekt referenciáját a Windows Phone projektünkhöz! Szükségünk lesz egy listára, amely a zenefájlokra hivatkozik, ebben a listában **AudioTrack** típusú objektumpéldányokat sorolunk fel. Az **AudioTrack** konstruktora egymás után a következő paramétereket fogadja: zenefájl, dal címe, előadó, lemez címe, lemez képe. Szükségünk lesz még egy számlálóra is, amely azt tárolja, hogy éppen melyik zeneszámot játszunk le! Ha annak értéke egyenlő a lista méretével, akkor túl vagyunk az utolsó zeneszámon, és a nulladik sorszámúra kell ugranunk. Ha visszafelé lépünk a zeneszámok között, akkor pedig a logikát megfordítva, 0 után a lista utolsó elemére ugrunk. Ezek után nincs semmi más dolgunk, mint a szükséges helyeken meghívni a lejátszásért felelős függvényt, illetve váltani listában a felhasználó igénye szerint:

```
static int currentTrackNumber = 0;

private static List<AudioTrack> _playList = new List<AudioTrack>
{
    new AudioTrack(new Uri("1-01 Mr. Know It All.mp3", UriKind.Relative),
        "Mr. Know It All",
        "Kelly Clarkson",
        "Stronger",
        null),

    new AudioTrack(new Uri("1-02 What Doesn't Kill You (Stronger).mp3", UriKind.Relative),
        "Stronger",
        "Kelly Clarkson",
        "Stronger",
        null),

    new AudioTrack(new Uri("1-03 Dark Side.mp3", UriKind.Relative),
        "Dark Side",
        "Kelly Clarkson",
        "Stronger",
        null),

    new AudioTrack(new Uri("http://traffic.libsyn.com/wpradio/WPRadio_29.mp3",
UriKind.Absolute),
        "Episode 29",
        "Windows Phone Radio",
        "Windows Phone Radio Podcast",
        null)
};

private void PlayNextTrack(BackgroundAudioPlayer player)
{
    if (++currentTrackNumber >= _playList.Count)
    {
        currentTrackNumber = 0;
    }

    PlayTrack(player);
}

private void PlayPreviousTrack(BackgroundAudioPlayer player)
{
    if (--currentTrackNumber < 0)
    {
        currentTrackNumber = _playList.Count - 1;
    }
}
```

```

    }

    PlayTrack(player);
}

private void PlayTrack(BackgroundAudioPlayer player)
{
    player.Track = _playList[currentTrackNumber];
}

protected override void OnPlayStateChanged(BackgroundAudioPlayer player, AudioTrack track,
PlayState playState)
{
    switch (playState)
    {
        case PlayState.TrackReady:
            player.Play();
            break;

        case PlayState.TrackEnded:
            PlayNextTrack(player);
            break;
    }

    NotifyComplete();
}

protected override void OnUserAction(BackgroundAudioPlayer player, AudioTrack track, UserAction
action, object param)
{
    switch (action)
    {
        case UserAction.Play:
            PlayTrack(player);
            break;

        case UserAction.Pause:
            player.Pause();
            break;

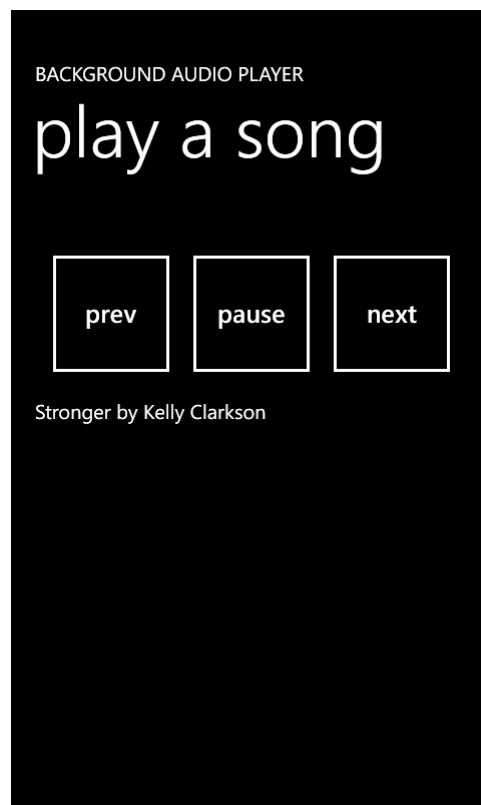
        case UserAction.SkipPrevious:
            PlayPreviousTrack(player);
            break;

        case UserAction.SkipNext:
            PlayNextTrack(player);
            break;
    }

    NotifyComplete();
}

```

Teszteljük alkalmazásunkat emulátorban, vagy akár éles eszközön, hogyha rendelkezésünkre áll! Az 5-7. ábra az alkalmazást mutatja be futás közben.



5-7. ábra: Az Audio Player alkalmazás futás közben

### Figyelmeztetések és riasztások kezelése

A Windows Phone Mango lehetővé teszi, hogy saját riasztásokat és emlékeztetőket készítsünk. Példaként egy saját alkalmazásomat mutatom be, amely főzés közben segít nekem. A neve **Cook Timer** és megtalálható a Windows Phone Marketben. Három különböző időzítőt tudunk beállítani, melyek egymástól független mérik az idő előrehaladását, és amikor elkészült az adott étel, figyelmeztetnek bennünket. Természetesen ehhez nem kell folyamatosan használnunk az alkalmazást, az nyugodtan deaktivált állapotba kerülhet, a figyelmeztetések akkor is működni fognak! Ezeket a **ScheduledActionService** objektum felügyeli, melyen keresztül egy új emlékeztetőt be tudunk ütemezni, illetve ha a felhasználó változtatást végzett a háttérben, esetleg le szeretné állítani a riasztást, akkor ki tudjuk azt keresni és törölni tudjuk a sorból. Új időzítés hozzáadásra a következő kódrészlet mutat példát:

```
private void MakeAlarm()
{
    alarm = new Alarm( Name );
    alarm.Content = "Time is up!\n" + Name + " is/are done now!";
    alarm.BeginTime = DateTime.Now.AddMinutes(Math.Ceiling( beginTime.Value ));
    alarm.ExpirationTime = DateTime.Now.AddMinutes(Math.Ceiling(beginTime.Value ))
        .AddSeconds( 10 );
    alarm.RecurrenceType = RecurrenceInterval.None;
    alarm.Sound = new Uri( "Sounds/House Fire Alarm.mp3", UriKind.Relative );

    ScheduledActionService.Add( alarm );
}
```

A fenti példában látható, hogy a riasztásnak egy nevet kell adnunk, definiálni kell a megjelenítendő tartalmát, és meg kell adni a riasztás időpontját, illetve azt, hogy újra emlékeztessen-e. Saját hangokat is rendelhetünk hozzá, azonban ekkor figyeljünk arra, hogy az adott MP3 az alkalmazás mellé legyen

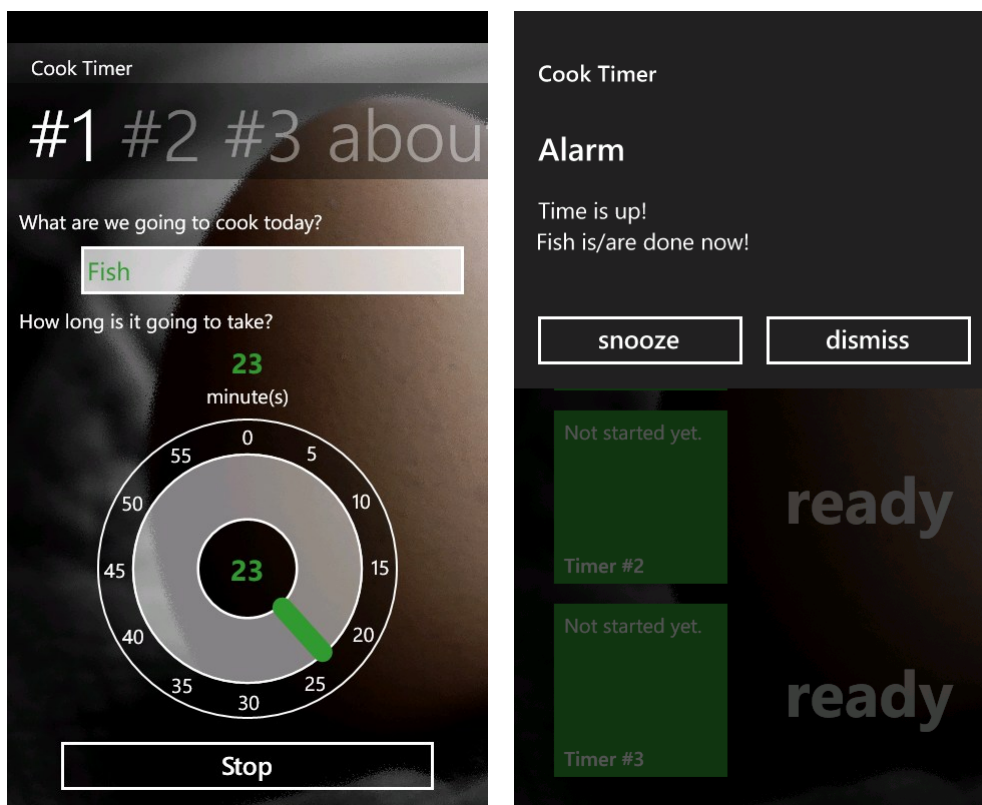


csomagolva, ne Zune média könyvtárból akarjuk elérni, mert ahhoz itt sincs hozzáférésünk! Ha időközben szeretnénk leállítani a riasztást és törölni a sorból, akkor arra a következő kódrészlet mutat példát:

```
void CleanupOldReminder( AlarmItem item )
{
    var reminder = ScheduledActionService.Find( item.Title );

    if ( reminder != null && reminder.IsScheduled == false )
        ScheduledActionService.Remove( reminder.Name );
}
```

Az 5-8. ábra az alkalmazást mutatja be működés közben.



5-8. ábra: a Cook Timer alkalmazás működés közben

## Adatok letöltése a háttérben

Elképzeltető, hogy az alkalmazásunk időről időre adatokat szeretne szinkronizálni valamilyen szerver oldali komponenssel, esetleg nagyobb állományok letöltése vált szükségessé. A Windows Phone Mango bevezette a Background File Transfer objektumokat, melyek HTTP és HTTPS protokollokon képesek adatokat forgalmazni. Sajnos az FTP protokoll használata jelenleg nem támogatott.

Mielőtt részletesen megnéznénk, hogyan is történik letöltés, nézzük meg azokat a korlátozásokat, amelyekkel számolnunk kell!

### Méretkorlátok:

Adatforgalmazás típusa	Maximális méret
Fájl feltöltése	5 MB
Fájl letöltése mobil csatornán (3G, HSDPA, stb)	20 MB
Fájl letöltése WIFI hálózaton (akkumulátorról)	100 MB

### Egyéb korlátozások:

Korlátozás típusa	Maximális érték
Az adatforgalmazó sorban lévő elemek száma	5 (a teljesített letöltéseket manuálisan kell törölnünk kódból)
Egyidejű adatforgalmazás adott telefonról (több alkalmazás által)	2
Maximálisan beütemezhető letöltések (várakozási sorba kerülnek)	500
HTTP fejlécek száma kérésenként	15
HTTP fejlécek maximális mérete	16 KB külön-külön

Fontos azt is tudni, hogy a letöltések szüneteltetésre kerülnek, ha 3G hálózat esetében 50 Kbps és WIFI esetében 100 Kbps alá csökken a hálózat áteresztő képessége.

Két objektumtípus segít bennünket ezeknek a feladatoknak a megvalósításában. Az egyik a **BackgroundTransferRequest**, amely leírja magát a kérést egyetlen távoli állományra mutatva. A másik a **BackgroundTransferService**, ez egy olyan sorként képzelhető el, amely egyesével végrehajtja az átadott letöltés-kéréseket.

A **BackgroundTransferRequest** objektumnak meg kell adnunk, hogy honnan kell a letöltést végeznie, és a HTTP protokoll melyik igéjét szeretnénk hozzá használni. Miután fájlrendszerbe fognak letöltődni a fájlok, így egy olyan helyi elérési útra is szükségünk lesz, ahova ezeket a Windows Phone bemásolhatja. Azt is megadhatjuk, hogy milyen beállítások mellett szeretnénk az adatforgalmazást engedélyezni. Ilyen beállítás például az, hogy csak a celluláris hálózaton, esetleg Wifi-n keresztül kívánjuk ezt lehetővé tenni, esetleg várjunk töltő csatlakoztatására, hanghívás megzavarhatja-e a letöltést (ha nem 3G hálózaton vagyunk, hanem mondjuk GPRS vagy EDGE módban működünk, akkor ez előfordulhat), és így tovább. Ha a beállításokkal végeztünk, akkor a kivételkezelést figyelembe véve megkezdhetjük a letöltést a **BackgroundTransferService** objektum segítségével. A következő kódrészlet a Silverlight 4.0 fejlesztőknek című könyv PDF változatát tölti le a háttérben:

```
// Silverlight 4.0 fejlesztőknek könyv pdf-ben
string transferFileName = "http://goo.gl/9dkcY";
Uri transferUri = new Uri( Uri.EscapeUriString( transferFileName ), UriKind.RelativeOrAbsolute );
BackgroundTransferRequest request = new BackgroundTransferRequest( transferUri );
request.Method = "GET";

//megadjuk, hogy hova töltsse le a fájlt
string downloadFile = transferFileName.Substring(transferFileName.LastIndexOf( "/" ) + 1 );
Uri downloadUri = new Uri( "shared/transfers/" + downloadFile, UriKind.RelativeOrAbsolute );
request.DownloadLocation = downloadUri;

request.TransferPreferences = TransferPreferences.AllowCellularAndBattery;

try
{
    // a háttérben beütemezzük a letöltést
    BackgroundTransferService.Add(request);
} catch(Exception ex) {
    // ha valami probléma volt
}
```

## Összefoglalás

A fejezetben láthattuk, hogy az első Windows Phone 7 verzióhoz képest – amely alig egy évvel ezelőtt jelent meg – mennyit fejlődött az operációs rendszer Mango kódnevű legújabb kiadása. Az új multitasking környezet számos új dolgot tesz lehetővé a fejlesztőknek, amelyre eddig csak nagyon nehézkesen vagy esetleg egyáltalán nem találhattunk megoldást. Az elméleti bevezetést követően gyakorlatban is láthattuk azokat a lépéseket, melyek az alkalmazást segítik beilleszkedni a Windows Phone életciklus modelljébe, és arról gondoskodnak, hogy semmilyen esetben se veszthessenek el a felhasználó munkafolyamataihoz tartozó adatok. Láthattuk, hogy nem végezhetünk akármit szabadon a háttérben, minden feladatnak speciálisan a hozzá illeszkedő feladat kategóriába kell kerülnie, ahol a megfelelő objektumok végrehajtják az általunk megvalósított feladatok logikáját.

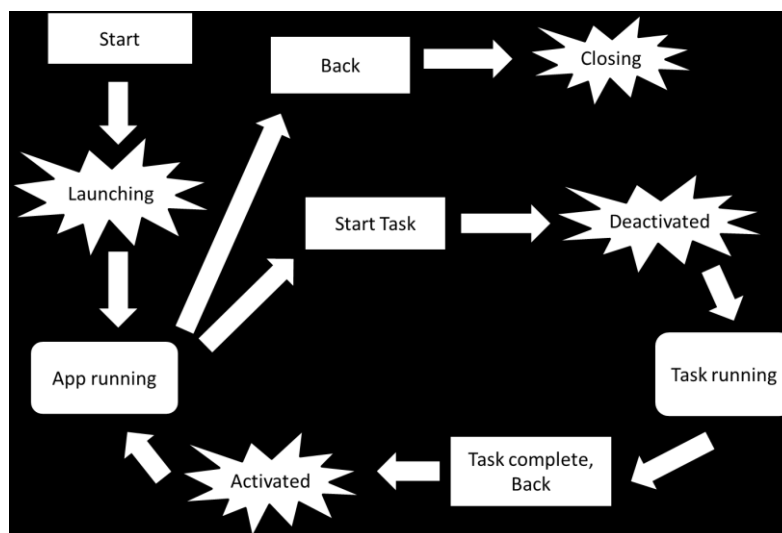


## 6. Alapvető telefonos funkciók használata

A Windows Phone 7 platform megalkotása során az egyik legfontosabb szempont a biztonság, a rendszer sebezhetetlensége volt. Emiatt a platform natív, a CLR réteg alatt futó biztonsági mechanizmusokkal rendelkezik, de ennek köszönhető az is, hogy az összes alkalmazás úgynevezett **sandbox**-ban, vagyis a többi programtól elzárt „homokozóban” fut. Ez a saját tér tartalmazza az alkalmazás izolált tárhelyét (Isolated Storage) és a program kontextusát, oly módon, hogy ahhoz külső alkalmazások ne tudjanak hozzáférni, sőt annak létezéséről ne is tudhassanak. Ennek következménye az, hogy a programok nem egy felhasználó kontextusában futnak, valamint a teljes háttértárra vonatkozó fájlrendszer fogalma sem értelmezhető az alkalmazások szemszögéből.

Az alkalmazások tehát egymás elől rejtve futnak és léteznek, viszont előfordulnak olyan általános feladatok, amelyekre számos programnak szüksége lehet. Ilyen feladat egy telefonhívás indítása, e-mail vagy SMS küldése, fényképek készítése, de ide tartozik a médiatár és a kapcsolattár elérése is. Ezeket a funkciókat a platform részeként szállított alkalmazások valósítják meg. Azért, hogy ezekhez fejlesztőként mi is hozzáférhessünk, bevezették az úgynevezett taszkok (Task) rendszerét. A taszkok a platform főbb funkcióihoz hozzáférést biztosító programok, amelyek ugyan szintén sandboxban futnak, de bármely alkalmazás által elérhetőek az SDK-n keresztül. A taszkok a **Microsoft.Phone.Tasks** névtérben találhatóak, indításuk a példányosítást és helyes paraméterezést követő **Show()** függvényhívással történik.

Egy taszk indításakor alkalmazásunk a háttérbe kerül, deaktiválódik, a futtatott feladat pedig aktívvá válik. Más szavakkal: kontextusváltás történik, amelynek működését bővebben az 5. fejezet taglalja. A folyamat lefutását, az érintett állapotokat és az ezek közti eseményeket a 6-1 ábra mutatja.



6-1 ábra: Események és állapotváltozások taszk indításakor

A taszkok két fő csoportra bonthatók: a valamilyen adatot visszatérési értéként szolgáltató **Chooserekre** és az egyszerű programindítást elvégző **Launcherekre**. Egy telefonhívás indítása például egy **Launcher**, hiszen nem várunk semmi adatot, a lényeg csupán a feladat elvégzése. Ezzel szemben egy fénykép készítése során értelmetlen lenne, ha nem kapnánk vissza az elkészült képet, tehát ezt a funkciót egy **Chooser** valósítja meg.

## 6. Alapvető telefonos funkciók használata

A Mango verzióban elérhető **Launcherek** az alábbiak:

Launcher neve	Megvalósított funkció
EmailComposeTask	E-mail üzenet írása és küldése
MediaPlayerLauncher	Megadott média (video / audio) lejátszása
PhoneCallTask	Telefonhívás indítása
SmsComposeTask	SMS írása
WebBrowserTask	Böngésző indítása
SearchTask	Webes keresés
MarketplaceDetailTask	Alkalmazás részleteinek mutatása a Marketplace kliensben
MarketplaceHubTask	Marketplace kliens indítása
MarketplaceReviewTask	Vélemény írása alkalmazásunkról a Marketplace-re
MarketplaceSearchTask	Keresés indítása a Marketplace kliensben

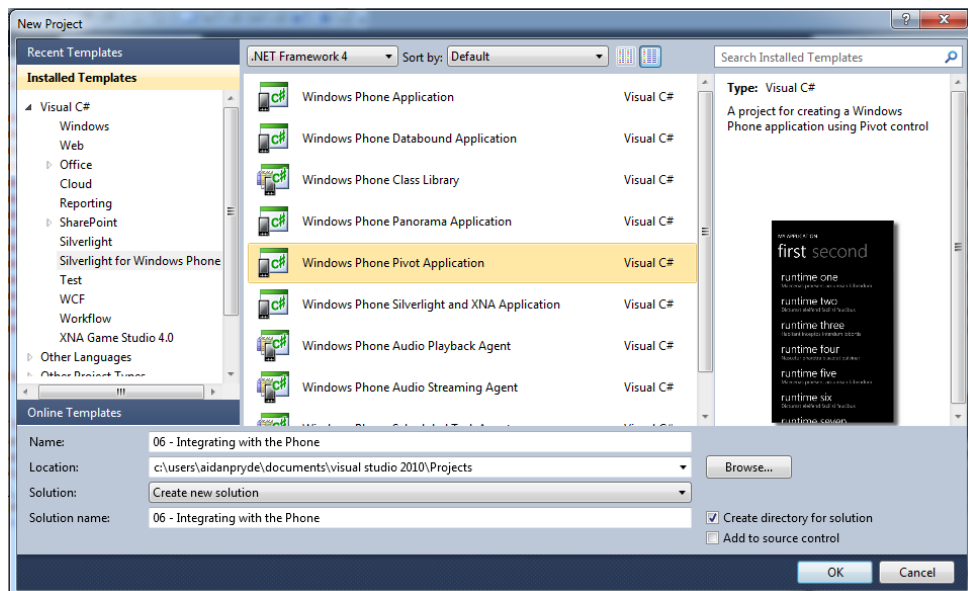
A **Chooserek** listája:

Choose neve	Megvalósított funkció
SavePhoneNumberTask	Telefonszám mentése a kapcsolatok tárába
SaveContactTask	Új partner felvétele a kapcsolatlistára
SaveEmailAddressTask	E-mail cím mentése a kapcsolatok tárába
EmailAddressChooserTask	E-mail cím kiválasztása kapcsolatlistánkról
AddressChooserTask	Cím kiválasztása kapcsolatlistánkról
CameraCaptureTask	Kép készítése alkalmazásunk számára
PhoneNumberChooserTask	Telefonszám kiválasztása kapcsolatlistánkról
PhotoChooserTask	Fénykép kiválasztása a fényképtárból
SaveRingtoneTask	Csengőhang mentése
GameInviteTask	Több résztvevős játékokban a partnerek meghívását teszi lehetővé

Több olyan taszk is létezik, amely feladatát egy-egy beépített vezérlő is képes ellátni. Amennyiben viszont nincsenek egyedi igényeink a funkcióval kapcsolatban, akkor nemcsak egyszerűbb a taszk használata, de a platform többi részével konzisztens megoldást is biztosít. Ez a felhasználó számára sokkal egységesebb rendszerképet és könnyebb kezelhetőséget garantál.

## A példaalkalmazás létrehozása

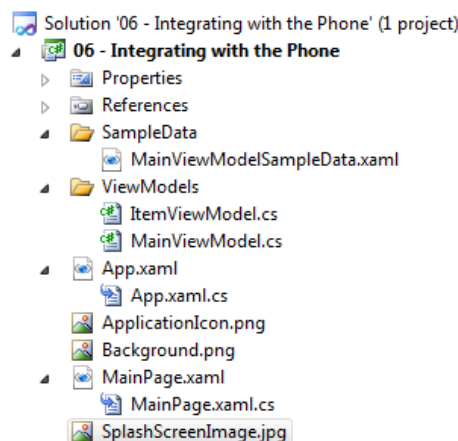
A fejezet első részében a felsorolt taszkokkal fogunk részletesen megismerkedni. Ehhez hozzunk létre egy új projektet **06 - Integrating with the Phone** névvel! Mivel a **Chooserek** és **Launcherek** két jól szeparálható részre osztják a feladatot, érdemes ezek mentén külön lapokra bontani a megjelenítési felületet is. Erre elegáns módszer a 4. fejezetben bemutatott **Pivot** vezérlő, amit legegyszerűbben a Windows Phone Pivot Application projektsablon kiválasztásával (6-2 ábra) helyezhetünk el alkalmazásunkban.



**6-2 ábra: Windows Phone Pivot Application sablon kiválasztása a projekt létrehozásakor**

A következő dialógusban mindenképpen válasszuk a Windows Phone OS 7.1-es verziót, mivel a taszkok egy része csupán a Mango frissítéssel vált elérhetővé!

A sablonból generált projekt felépítését (6-3 ábra) érdemes tüzetesebben átvizsgálni, mivel példát mutat arra, hogyan tudjuk az MVVM modellnek megfelelően szervezni kódunkat.



**6-3 ábra: A sablon által generált projekt**

Ez a megközelítés a lehető legjobban elválasztja a különböző felelősségeket (adatszolgáltatás, logika, megjelenítés), és kihasználja a Silverlight által biztosított adatkötési mechanizmust. A **ViewModels** mappa fogja össze azokat az osztályokat, amelyek az egyes oldalak, megjelenítési felületek által elvárt formában tárolják az egyéb adatforrásból származó információkat. Ilyen adatforrás például a **SampleData MainViewModelSampleData.xaml** állománya, ami XAML formátumban rögzít néhány tesztbejegyzést a tervezésidejű megjelenítéshez (a Blend és a Visual Studio design felülete is ezt használja). Ahhoz, hogy ezt a forrást a tervező eszközök meg is jelenítsék, csak az alábbi attribútumokat kell hozzáadni a PhoneApplicationPage deklarációjához (természetesen a sablon alapján ezt a VS már megtette helyettünk):

```
xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
mc:Ignorable="d" d:DesignWidth="480" d:DesignHeight="768"
d:DataContext="{d:DesignData SampleData/MainViewModelSampleData.xaml}"
```

Futtatáskor azonban nem ezeket az adatokat fogjuk látni, ugyanis a **MainViewModel** osztály rendelkezik egy **LoadData** metódussal, ami a valós adatok előállításáért felelős. A **Model** osztály példányát az **App.xaml.cs** állítja elő, és publikálja egy statikus mező formájában:

```
private static MainViewModel viewModel = null;
//...
public static MainViewModel ViewModel
{
    get
    {
        // ViewModel létrehozásának elhalasztása addig, amíg szükségünk nem lesz rá
        if (viewModel == null)
            viewModel = new MainViewModel();

        return viewModel;
    }
}
```

A megjelenítésért a **MainPage.xaml** és a hozzá tartozó kódfájl felelős. Mivel a Pivot sablont választottuk, az oldal központi eleme ez a vezérlő. Használatához hivatkoznunk kell a **Microsoft.Phone.Controls** névtérre, de ezt a Visual Studio már megtette helyettünk, sőt felvett két lapot a **Pivot**ba, és azokon egy-egy **ListBox**ot helyezett el. A listák adatkötés használatával tárják elének a ViewModelekben definiált publikus tulajdonságokat.

```
xmlns:controls="clr-namespace:Microsoft.Phone.Controls;assembly=Microsoft.Phone.Controls"
...
<Grid x:Name="LayoutRoot" Background="Transparent">
    <!--Pivot Control-->
    <controls:Pivot Title="MY APPLICATION">
        <!--Pivot item one-->
        <controls:PivotItem Header="first">
            <!--Double line list with text wrapping-->
            <ListBox x:Name="FirstListBox" Margin="0,0,-12,0"
                ItemsSource="{Binding Items}">
                <ListBox.ItemTemplate>
                    <DataTemplate>
                        <StackPanel Margin="0,0,17" Width="432" Height="78">
                            <TextBlock Text="{Binding LineOne}" TextWrapping="Wrap"
                                Style="{StaticResource PhoneTextExtraLargeStyle}"/>
                            <TextBlock Text="{Binding LineTwo}" TextWrapping="Wrap"
                                Margin="12,-6,12,0"
                                Style="{StaticResource PhoneTextSubtleStyle}"/>
                        </StackPanel>
                    </DataTemplate>
                </ListBox.ItemTemplate>
            </ListBox>
        </controls:PivotItem>

        <!--Pivot item two-->
        <controls:PivotItem Header="second">
            <!--Triple line list no text wrapping-->
            <ListBox x:Name="SecondListBox" Margin="0,0,-12,0"
                ItemsSource="{Binding Items}">
                <ListBox.ItemTemplate>
                    <DataTemplate>
                        <StackPanel Margin="0,0,17">
```



```

        <TextBlock Text="{Binding LineOne}"
            TextWrapping="NoWrap" Margin="12,0,0,0"
            Style="{StaticResource PhoneTextExtraLargeStyle}"/>
        <TextBlock Text="{Binding LineThree}"
            TextWrapping="NoWrap" Margin="12,-6,0,0"
            Style="{StaticResource PhoneTextSubtleStyle}"/>
    </StackPanel>
</DataTemplate>
</ListBox.ItemTemplate>
</ListBox>
</controls:PivotItem>
</controls:Pivot>
</Grid>

```

A megjelenítés és az adatok megismerése után már csak a két oldal közötti kapocs kialakítása maradt hátra. Az oldal kontextusát a **MainPage** konstruktora állítja a megfelelő ViewModelre:

```

// Constructor
public MainPage()
{
    InitializeComponent();

    // Adatkötés
    DataContext = App.ViewModel;
    this.Loaded += new RoutedEventHandler(MainPage_Loaded);
}

// Adatok betöltése, amint elkészült az oldal.
private void MainPage_Loaded(object sender, RoutedEventArgs e)
{
    if (!App.ViewModel.IsDataLoaded)
    {
        App.ViewModel.LoadData();
    }
}

```

A generált projekt megvizsgálásával egy kis betekintést nyerhettünk az MVVM architektúris minta használatába, viszont ha komolyan érdeklődünk a téma iránt, akkor a *Silverlight 4: A technológia, és ami mögötte van* című könyv 10. fejezete tökéletes kiindulópontot jelent a tanuláshoz.

Alakítsuk át a projektet a taszkok teszteléséhez szükséges formába! A Pivot két lapot tartalmazzon, egyet a launcherek és egyet a chooserek számára. Minden taszk indítására vegyünk fel külön gombot, ezek pozicionálását pedig bízzuk egy **StackPanel**re! A **StackPanel**t viszont mindenképpen egy **ScrollViewer**en belül definiáljuk, hiszen ezzel biztosítható az, hogy a panel görgethetővé válik, amennyiben a tartalom túlnyúlik a rendelkezésre álló helyen!

```

<controls:Pivot Title="Integrating with the Phone">
    <controls:PivotItem Header="launchers">
        <ScrollViewer>
            <StackPanel>
                <Button Content="PhoneCallTask" Name="btnPhoneCallTask"
                    Click="btnPhoneCallTask_Click"/>
                <Button Content="SmsComposeTask" Name="btnSmsComposeTask"
                    Click="btnSmsComposeTask_Click"/>
                <Button Content="WebBrowserTask" Name="btnWebBrowserTask"
                    Click="btnWebBrowserTask_Click" />
                <Button Content="EmailComposeTask" Name="btnEmailLauncher"
                    Click="btnEmailLauncher_Click" />
                <Button Content="MediaPlayerLauncher" Name="btnMediaLauncher"
                    Click="btnMediaLauncher_Click" />
                <Button Content="SearchTask" Name="btnSearchTask"

```

```
Click="btnSearchTask_Click"/>
</StackPanel>
</ScrollView>
</controls:PivotItem>

<controls:PivotItem Header="choosers">
  <ScrollView>
    <StackPanel>
      <Button Content="SavePhoneNumberTask" Name="btnSavePhone"
        Click="btnSavePhone_Click" />
      <Button Content="PhoneNumberChooserTask" Name="btnPhoneNumberChooserTask"
        Click="btnPhoneNumberChooserTask_Click" />
      <Button Content="CameraCaptureTask" Name="btnCameraCaptureTask"
        Click="btnCameraCaptureTask_Click" />
      <Button Content="PhotoChooserTask" Name="btnPhotoChooserTask"
        Click="btnPhotoChooserTask_Click" />
      <Image Name="imgResult" Width="400" Height="400" />
    </StackPanel>
  </ScrollView>
</controls:PivotItem>
</controls:Pivot>
```

A launcherek és chooserek listájáról hiányoznak a Marketplace-szel kapcsolatos taszkok, mivel ezeket a 12. fejezet részletesen taglalja. Ezenfelül fölösleges különbséget tenni a telefonszámot vagy e-mail címet szolgáltató és módosító feladatok között, hiszen működésük teljesen azonos, csupán az érintett tulajdonság különbözik. A chooserek lapja tartalmazzon egy **Image** vezérlőt is, hiszen a visszatérési értékek tartalmazhatnak szöveget és képet is! A szöveges válaszok megjelenítésére tökéletes egy felugró ablak (**MessageBox**), de vizuális elemeket ez sajnos nem tartalmazhat.

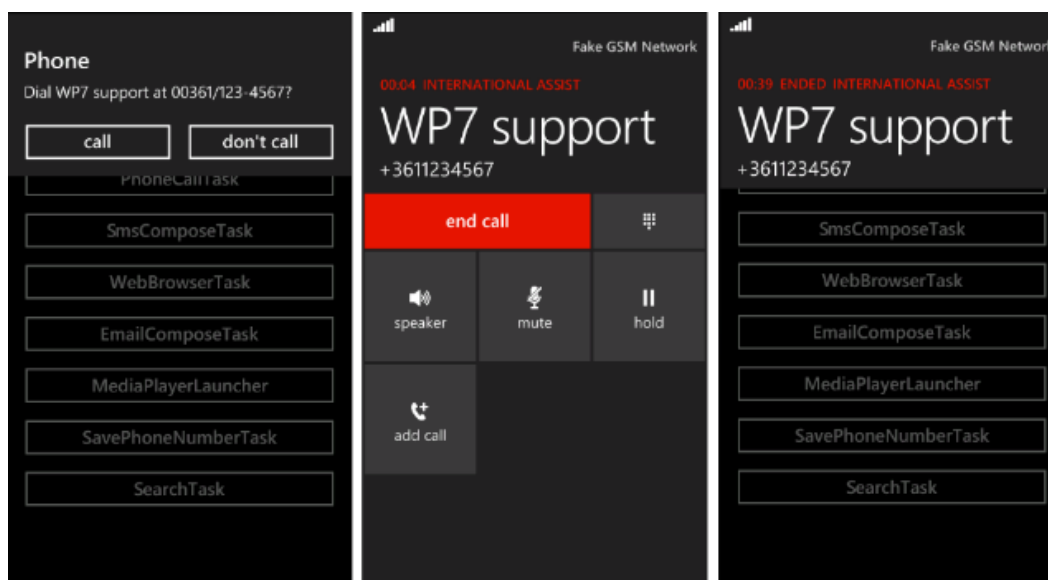
## Launcherek

### *PhoneCallTask*

Telefonhívás indításakor kötelező megadni a hívandó telefonszámot. Emellett egy nevet is megjeleníthetünk, de ennek hiánya sem okoz problémát:

```
private void btnPhoneCallTask_Click(object sender, RoutedEventArgs e)
{
    PhoneCallTask phoneCallTask = new PhoneCallTask();
    // a tagolás nem kötelező, de így is megadhatjuk a számot
    phoneCallTask.PhoneNumber = "00361/123-4567";
    // ezt nem kötelező megadni
    phoneCallTask.DisplayName = "WP7 support";
    phoneCallTask.Show();
}
```

A **Show()** metódus elindítja a taszkot, ami először lehetőséget ad a felhasználónak a hívás elindítására vagy elutasítására. A hívás lebonyolítása a megszokott felületen történik, a kihangosítási, némítási lehetőségekkel, majd a hívást összegző üzenetet láthatjuk (6-4 ábra).



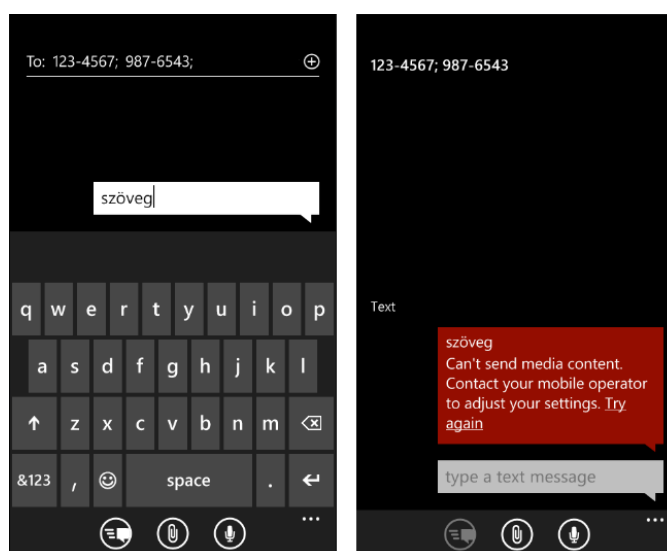
6-4 ábra: Telefonhívás lebonyolítása alkalmazásunkból

## SmsComposeTask

Sms küldésekor lehetőségünk van megadni a címzettek listáját és az üzenet szövegét is, de mindkét érték elhagyható:

```
private void btnSmsComposeTask_Click(object sender, RoutedEventArgs e)
{
    SmsComposeTask smsComposeTask = new SmsComposeTask();
    smsComposeTask.To = "123-4567; 987-6543";
    smsComposeTask.Body = "szöveg";
    smsComposeTask.Show();
}
```

A feladat indításakor a megadott értékek megjelennek az sms-ek küldését végző alkalmazásban. A címzetteket a + gombra kattintva kiegészíthetjük, ahogyan az üzenet szövegét is átírhatjuk. Az **ApplicationBar**on található gombokkal médiaelem csatolására és üzenet diktálására is lehetőségünk van. Természetesen a küldés az emulátoron meg fog hiúsulni, amit az üzenetváltásban jelez számunkra a rendszer(6-5 ábra).

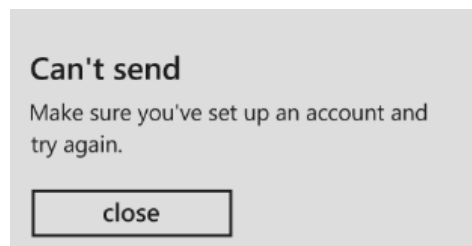


6-5 ábra: Sms küldése SmsComposeTask használatával

### EmailComposeTask

Levelek küldéséhez szükségünk van egy levelező szerverre is. Szerencsére ez nem jelent külön feladatot, hiszen a telefon konfigurálása során valószínűleg minden felhasználó beállít egy felhasználói fiókot, amelyet a szolgáltatások igénybevétele során használ. A levelek küldése előtt a taszk megkérdezi a felhasználót, hogy melyik fiókot szeretné használni a feladathoz, ezt követően jelenik csak meg a szerkesztő felület. Mivel az emulátoron jelenleg nincs mód arra, hogy ilyen fiókokat állítsunk be, a levélküldés sajnos nem lesz tesztelhető, csak készülékre telepítve (6-6 ábra).

Két megoldás létezik arra, hogy ezt a funkciót kipróbáljuk. Az egyik az, hogy telefonkészülékre telepítjük az alkalmazásunkat, a másik pedig az, hogy keresünk egy a Microsoft által nem támogatott emulátort, ami alkalmas ennek a szolgáltatásnak a kipróbálására. Mi a készüléken való kipróbálást ajánljuk, ugyanis az a megbízhatóbb, de bizonyos esetekben jó tudni arról, hogy létezik más megoldás is. (Ezeket az emulátorokat többnyire ingyenesen tölthetjük le.)



6-6 ábra: A levélküldés sajnos nem elérhető az emulátorról

```
private void btnEmailLauncher_Click(object sender, RoutedEventArgs e)
{
    EmailComposeTask emailTask = new EmailComposeTask();
    // több címzett megadása
    emailTask.To = "alpha@test.com; bravo@test.com";
    // másolat küldése
    emailTask.Cc = "charlie@test.com";
    // rejtett másolat
    emailTask.Bcc = "delta@test.com";
    emailTask.Subject = "tárgy";
    emailTask.Body = "üzenet szövege";
    // task indítása
    emailTask.Show();
}
```

Az üzenet minden adata a kódból is beállítható, de lehetőség van arra is, hogy ezek nélkül, pusztán a **Show()** függvényhívással egy üres e-mail alkalmazást nyissunk.

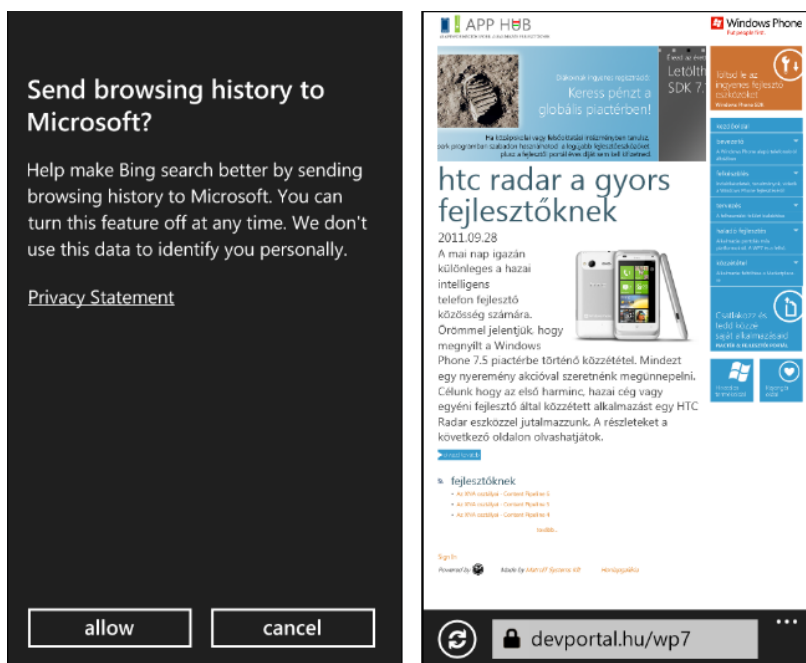
### WebBrowserTask

Gyakori igény alkalmazásokban teljes weboldalak megjelenítése. Erre tökéletes megoldás a WebBrowser vezérlő, viszont ez önmagában csak egy megjelenítő és értelmező felület, a beépített Internet Explorer 9 előnyeit, az alsó URL sávot, a kedvencek és előzmények rendszerezését és még számos funkciót elveszítünk a használatával. Ha nincsenek olyan egyedi igényeink, mint például saját elemek felvétele az **ApplicationBar**-ba, akkor érdemes lehet a böngésző programot elindító **WebBrowserTask**-ot használnunk:

```
private void btnWebBrowserTask_Click(object sender, RoutedEventArgs e)
{
    WebBrowserTask webBrowserTask = new WebBrowserTask();
    // elavult, már ne használjuk!
    //webBrowserTask.URL = "xy.com";
}
```

```
webBrowserTask.Uri = new Uri("http://www.devportal.hu/wp7", UriKind.Absolute);
webBrowserTask.Show();
}
```

A böngésző első indításakor engedélyt kér a böngészés adatainak felhasználására a Bing kereső továbbfejlesztésének érdekében. Választásunkat követően az oldal betöltődik az Internet Explorer 9 mobilos változatába. A böngésző a Mango frissítésnek hála teljes HTML 5 támogatással rendelkezik, így a generált oldal teljesen megegyezik a számítógépeken megszokottal. Jó példa erre az Orchard keretrendszerben elkészített HTML 5-öt használó <https://devportal.hu/wp7> oldal, ami a 6-7 ábrán látható.



6-7 ábra: A <https://devportal.hu/wp7> oldal egy WebBrowserTaskban megjelenítve

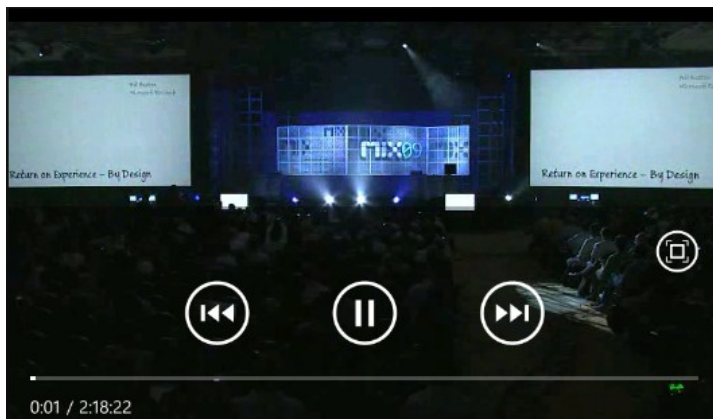
## MediaPlayerLauncher

A platform két rendkívül egyszerűen használható lehetőséget biztosít médiaelemek lejátszására. Az egyik a Toolboxban megtalálható **MediaElement** vezérlő, ami teljes testreszabhatóságot biztosít számunkra, a másik pedig a pár kódsorral paramétrezhető **MediaPlayerLauncher** taszk (6-8 ábra).

A taszk használatakor a példányosítást követően meg kell adnunk a lejátszandó video vagy audio elem címét, ami lehet lokális vagy távoli, internetről származó is. Ennek megfelelően ki kell választanunk, hogy relatív vagy abszolút címezsről van szó. A példában egy internetes videót választottam, így most abszolút címezést alkalmazunk. Ha a telefonon található tartalmat szeretnénk lejátszani, akkor meg kell adnunk, hogy az alkalmazással telepített vagy az Isolated Storage-be mentett médiát címezünk meg, ezt pedig a **MediaPlayerLauncher Location** tulajdonságán keresztül tehetjük meg.

```
private void btnMediaLauncher_Click(object sender, RoutedEventArgs e)
{
    MediaPlayerLauncher mpl = new MediaPlayerLauncher();
    mpl.Media = new Uri(@"http://mschannel9.vo.msecnd.net/o9/mix/09/wmv/key01.wmv",
        UriKind.Absolute);
    //mpl.Location = MediaLocationType.Data;
    mpl.Controls = MediaPlaybackControls.All;
    mpl.Show();
}
```

A **Controls** tulajdonságon keresztül kiválaszthatjuk, hogy milyen előre definiált vezérlők jelenjenek meg a médialejátszóban. Ez a mező a **MediaPlayerControl** enumeráció (**All**, **FastForward**, **None**, **Pause**, **Rewind**, **Skip**, **Stop**) egy értékét veheti fel. A 6-8 ábrán az összes elem együttes használata látható.

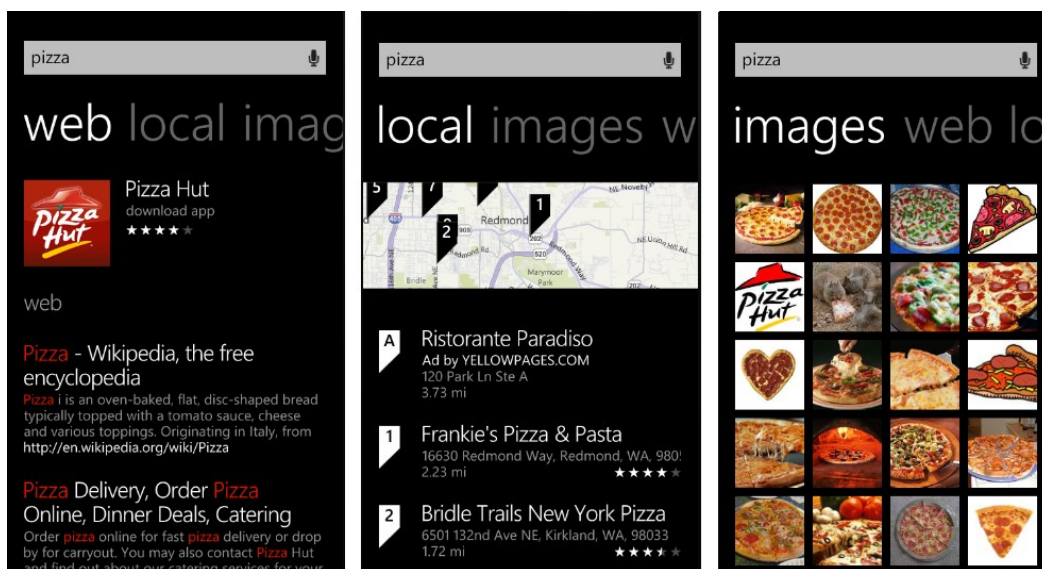


**6-8 ábra: MediaPlayerLauncher taszk egy videó lejátszása közben, minden beépített vezérlővel**

### SearchTask

Kereső funkciót megvalósító taszk. Segítségével a megadott kulcsszó alapján kereshetünk a weben, akár képek között, de ha engedélyeztük a Location Servicest, akkor a pozíciónk körüli kiemelt helyek között is. A Mango újdonsága az ún. App Connect, aminek hála a keresés a Marketplace alkalmazásait is megvizsgálja, és a többi találat előtt ajánlja az illeszkedő alkalmazásokat. A keresőszót a taszkon belül módosíthatjuk, a beviteli mező jobb szélén található mikrofon gombra kattintva akár diktálva is. A találatok egy Pivot vezérlőn jelennek meg, aminek lapjait a 6-9 ábra mutatja.

```
private void btnSearchTask_Click(object sender, RoutedEventArgs e)
{
    SearchTask searchTask = new SearchTask();
    searchTask.SearchQuery = "pizza";
    searchTask.Show();
}
```



**6-9 ábra: A SearchTask találatainak listája**

## Chooserek

A chooserek közé tartoznak azok a taszkok, amelyek adatot szolgáltatnak alkalmazásunk számára, valamint azok is, amelyekkel a műveletek kimenetéről, esetleges meghiúsulásokról kell értesítenünk a felhasználót. Egy telefonhívás indítása launcher, mivel egy fellépő hibáról azonnal értesül a felhasználó, de egy telefonszám mentését indítványozó **SavePhoneNumberTask** kimenete közvetlenül nem érzékelhető, hibája pedig adatvesztéshez vezethet. Emiatt az ilyen taszkok visszatérési értékkel rendelkeznek, ami tartalmazhat egy kivételt (Exception), illetve a futás kimenetelét (sikeres, meghiúsult), valamint bármilyen a taszk jellegéből adódó adatot (pl. képet).

A taszkok természetesen külön szálon futnak, így a kommunikáció egy eseménykezelő (**Completed**) implementálásával történik. A példa során ebben a metódusban egy párbeszédablakban fogjuk megjeleníteni a szöveges visszatérési értékeket, a képeket pedig a gombok alatt található **Image** vezérlőben. Egy valós alkalmazás során ez a megközelítés nem lenne működőképes, mivel a **MessageBox** megjelenítése, azaz a **Completed** eseménykezelő közvetlenül az **Application\_Activated** után fut le, mégis képes „megfogni” programunk aktiválódását. Ez azt jelenti, hogy amíg nem zártuk be a **MessageBox**ot az OK gombbal, addig a rendszer úgy érzékeli, hogy az állapotváltás eseménykezelője fut. Ennek az a következménye, hogy ha túl sokáig hagyjuk fent a párbeszédablakot, akkor a rendszer – az alkalmazás eseményeinek 10 másodperces időkorlátja miatt – leállítja programunk futását. Az állapotátmenetek működésével az 5. fejezet foglalkozik részletesen.

### SavePhoneNumberTask

Az első megvizsgált chooser esetében nem valamilyen szolgáltatott adatra, hanem a művelet eredményességére vagyunk kíváncsiak. Ennek megfelelően a taszk felparaméterezése után definiálnunk kell egy eseménykezelő függvényt, amely akkor fut le, amikor a taszk futása már befejeződött, és újra a mi alkalmazásunk kerül előtérbe.

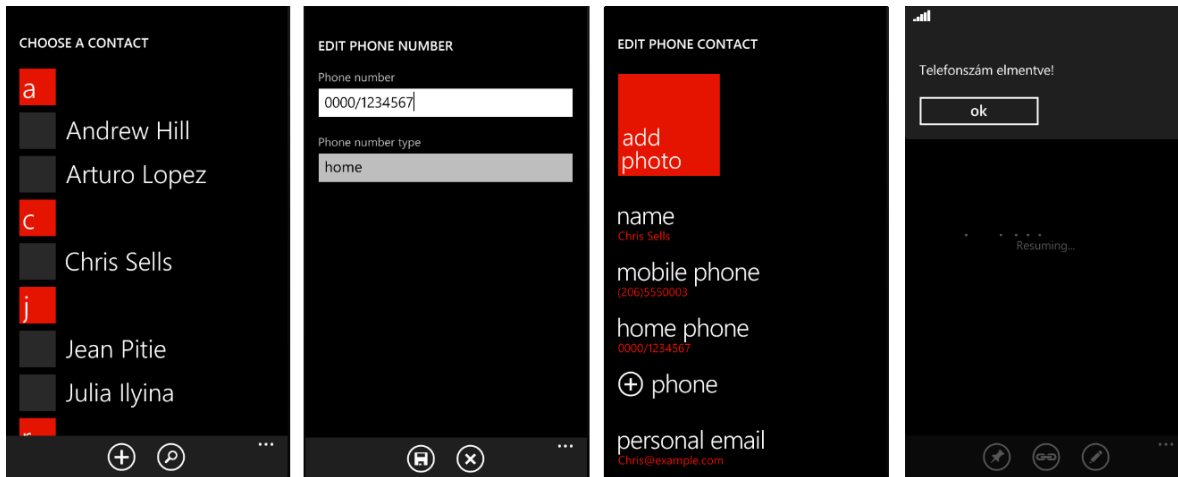
A **SavePhoneNumberTask** a **Contacts** alkalmazást indítja el, amelyben lehetőséget biztosít arra, hogy valamely ismerősünk adatait egy új telefonszámmal egészítsük ki. A visszatérési érték **TaskResult** mezője a **TaskResult** felsorolás egyik lehetséges értékével jelzi a futás kimenetelét, amit nekünk kell a felhasználó felé továbbítanunk egy üzenetdoboz megjelenítésével:

```
private void btnSavePhone_Click(object sender, RoutedEventArgs e)
{
    SavePhoneNumberTask spn = new SavePhoneNumberTask();
    spn.PhoneNumber = "0000/1234567";
    spn.Completed += (s, result) =>
    {
        switch (result.TaskResult)
        {
            case TaskResult.OK:
                MessageBox.Show("Telefonszám elmentve!");
                break;
            case TaskResult.Cancel:
                MessageBox.Show("Mentés megszakítva!");
                break;
            case TaskResult.None:
                MessageBox.Show("A telefonszám mentése nem sikerült!");
                break;
        }
    };
    spn.Show();
}
```

A **Completed** eseménykezelőjét megadhatjuk külön eljárásként is, de mivel egyszerű, rövid kódról van szó, amit egészen biztos, hogy nem fogunk máshol használni, érdemes lambda kifejezéssel megadott anonim metódust alkalmazni.



A **Contacts** alkalmazás indulásakor néhány előre definiált partner közül választhatunk, amelyek a tesztelések megkönnyítésére kerültek be az emulátorba. Ha itt kiválasztunk egy partnert, akkor a telefonszám mentésére lehetőséget biztosító oldal következik. A módosítások mentése után a kapcsolat adatait részletező oldalra kerülünk, ahonnan a Back gomb megnyomásával térhetünk vissza a taszkot indító alkalmazásba. Ekkor fut le az általunk definiált eseménykezelő, ami megvizsgálja és az általunk definiált eljárás alapján jelzi a visszatérési értéket. A folyamat a 6-10 ábrán tekinthető meg.



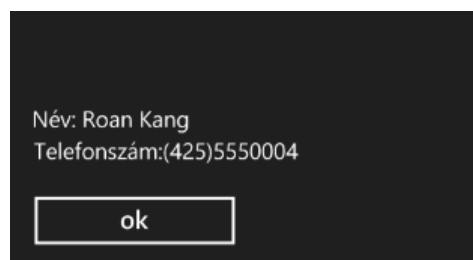
6-10 ábra: Egy telefonszám mentésének lépései és a visszatérési érték megjelenítése

### PhoneNumberChooserTask

A **PhoneNumberChooserTask** szintén a **Contacts** alkalmazást nyitja meg, viszont ez a taszk már adatot is szolgáltat a hívó alkalmazás számára. A **Completed** eseménykezelő második paramétere egy **PhoneNumberResult** példány, amin keresztül elérhetjük a kiválasztott partner nevét és telefonszámát (6-11 ábra):

```
private void btnPhoneNumberChooserTask_Click(object sender, RoutedEventArgs e)
{
    PhoneNumberChooserTask phoneNumberChooser = new PhoneNumberChooserTask();
    phoneNumberChooser.Completed += (s, result) =>
    {
        if (result.Error == null && result.TaskResult == TaskResult.OK)
        {
            string name = result.DisplayName;
            string number = result.PhoneNumber;

            MessageBox.Show(string.Format("Név: {0}\nTelefonszám:{1}",
                name, number));
        }
    };
    phoneNumberChooser.Show();
}
```



6-11 ábra: A kiválasztott partner neve és telefonszáma

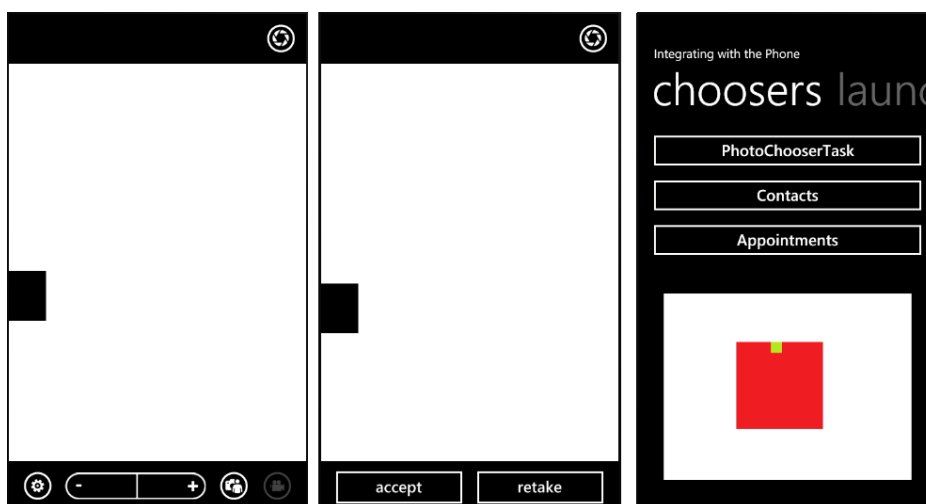


## CameraCaptureTask

A kamera használatának legegyszerűbb módja egy **CameraCaptureTask** indítása. A Mango frissítéssel megjelent a **PhotoCamera** osztály is az eszközkészletben, erről a fejezet következő részében lesz szó.

```
private void btnCameraCaptureTask_Click(object sender, RoutedEventArgs e)
{
    CameraCaptureTask cameraTask = new CameraCaptureTask();
    cameraTask.Completed += (s, result) =>
    {
        if (result.TaskResult == TaskResult.OK)
        {
            // System.Windows.Media.Imaging.BitmapImage
            BitmapImage bmp = new BitmapImage();
            bmp.SetSource(result.ChosenPhoto);
            imgResult.Source = bmp;
        }
    };
    cameraTask.Show();
}
```

Az emulátor szerencsére egy kamera működését is szimulálni tudja, ha nem is naplementés tájképekkel, de a célra tökéletesen megfelelő mozgó fekete kockával szembesülhetünk a taszk indításakor. A feladat egy **Stream**mel tér vissza, amit nem lehet közvetlenül képforrásként használni, ezért be kell vezetnünk egy **BitmapImage** változót is. Ez a **SetSource** metódusával már fel tudja dolgozni a streamet, így a kapott képet meg tudjuk jeleníteni (6-12 ábra).



6-12 ábra: Kép készítése **CameraCaptureTask** használatával

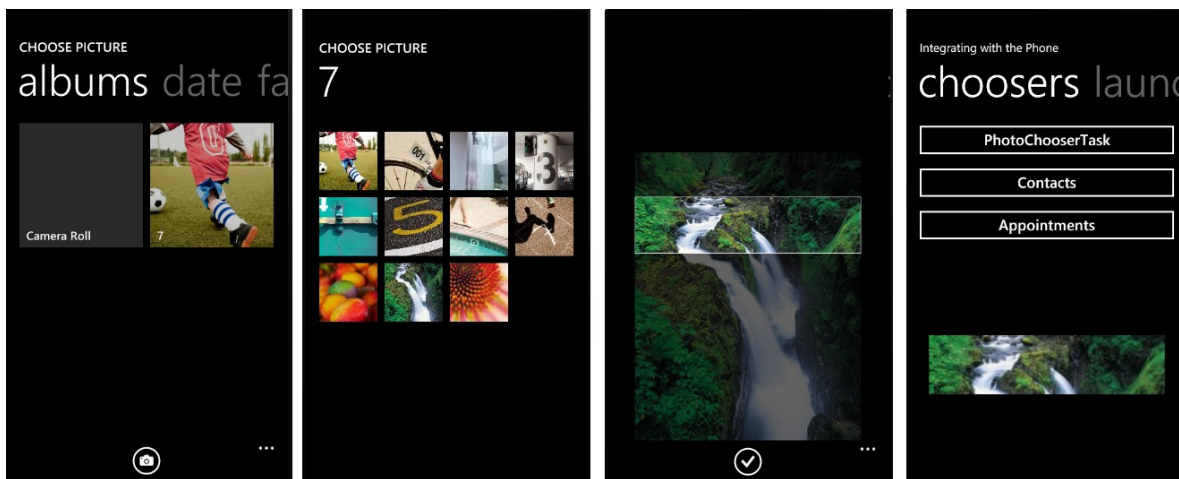
## PhotoChooserTask

A **PhotoChooserTask** egy kép kiválasztását teszi lehetővé a telefon médiatárából. Az osztály rendelkezik egy **ShowCamera** tulajdonsággal, amit ha engedélyezünk, akkor kikerül a különböző képtárak listája alá egy **CameraCaptureTask**ot indító gomb is.

Meghatározhatjuk az általunk elvárt kép dimenzióit is a **PixelHeight** és **PixelWidth** tulajdonságok értékének beállításával. Ha ezeket üresen hagyjuk, akkor a teljes kiválasztott képet visszkapjuk, abban az esetben viszont, ha a dimenziók rögzítése mellett döntünk, akkor lehetőségünk lesz a képen kijelölni a kívánt területet.

```
private void btnPhotoChooserTask_Click(object sender, RoutedEventArgs e)
{
    PhotoChooserTask photoChooserTask = new PhotoChooserTask();
    // lehetőséget adhatunk a kamera taszk indítására is egy gomb engedélyezésével
    photoChooserTask.ShowCamera = true;
    // dimenziók beállítása
    photoChooserTask.PixelHeight = 50;
    photoChooserTask.PixelWidth = 200;
    photoChooserTask.Completed += (s, result) =>
    {
        if (result.TaskResult == TaskResult.OK && result.OriginalFileName != null)
        {
            MessageBox.Show(result.OriginalFileName);
            BitmapImage bmp = new BitmapImage();
            bmp.SetSource(result.ChosenPhoto);
            imgResult.Source = bmp;
        }
    };
    photoChooserTask.Show();
}
```

A példakódban egy 50 pixel magas és 200 pixel széles képet szeretnénk beolvasni, így a taszk lehetőséget kínál egy ekkora terület kijelölésére, és csupán az ezen belül található képrészt adja majd vissza alkalmazásunk számára (6-13 ábra).



6-13 ábra: Meghatározott méretű kép kiválasztása és megjelenítése

## Adatok megosztása alkalmazások között

A taszkok rendszere mellett a Mango frissítéssel bekerült egy új lehetőség az alkalmazások közti adatmegosztásra. Az új fejlesztői készletben megjelent ugyanis a **PhoneDataSharingContext** osztály, ami egy kijelölt tárterületen képes lekérdezéseket futtatni, a találatokat pedig generikus listán szolgáltatja a hívó alkalmazásnak. Jelenleg két implementációja, a **Contacts** és az **Appointments** osztály érhető el.

Ezek bemutatására vegyünk fel egy új lapot a korábban használt pivotra **datasharing** névvel! Az új lapon definiáljunk egy-egy gombot, valamint egy új képet a két feladathoz:

```
<controls:PivotItem Header="datasharing">
    <ScrollView>
        <StackPanel>
            <Button Content="Contacts" Name="btnContacts"
                Click="btnContacts_Click" />
            <Button Content="Appointments" Name="btnAppointments"
                Click="btnAppointments_Click" />
            <Image Name="imgDataSharing" Width="400" Height="400" />
        
```

```

</StackPanel>
</ScrollView>
</controls:PivotItem>

```

## Contacts

A **Contacts** osztály a telefonon tárolt összes kapcsolatunkhoz hozzáférést biztosít alkalmazásunk számára. Segítségével név, e-mail cím, telefonszám vagy minden mező alapján kereshetünk, a találatoknak pedig minden információjához hozzáférhetünk.

A partnerlista egyik nagyszerű tulajdonsága az, hogy a különböző helyekről származó adatokat összefésüli, és egy közös tárban teszi hozzáférhetővé. Ez azt jelenti, hogy aki a SIM kártyánkra mentett ismerősünk, de szerepel például a facebookos vagy hotmailes ismerőseink listáján is, annak összes adatát egyetlen helyről, transzparens módon érhetjük el. Tehát annak ellenére, hogy több információforrásunk is van, ezek uniójaként egy felhasználót látunk a partnerlistánkon (néhány esetben a felhasználók összekapcsolását nekünk kell elvégeznünk, hiszen a név nem egyedi azonosító egy személy esetében). Ennek természetes velejárója az, hogy egy felhasználó tartalmazhat több telefonszámot, e-mail címet, de akár olyan ellentmondásos helyzetek is kialakulhatnak, amikor egy partnerünk több születéssel, párkapcsolattal (ennek megítélése nyilván felhasználófüggő) rendelkezik.

A Contacts nem túl meglepő módon a **Contact** osztály példányainak listáját szolgáltatja. Az ezen keresztül elérhető tulajdonságok az alábbi táblázatban találhatók:

Tulajdonság	Leírás
<b>Accounts</b>	Azon források listája, amelyeken partnerünk a felhasználó (pl. Live, facebook, stb.)
<b>Addresses</b>	Címek listája
<b>Birthdays</b>	Partnerünk összes születésnapjának listája
<b>Children</b>	Ismerős gyermekei
<b>Companies</b>	Munkahelyek listája
<b>CompleteName</b>	A partner teljes neve. Típusosan elérhető a név összes része ( <b>Nickname</b> , <b>FirstName</b> , <b>MiddleName</b> , <b>LastName</b> ).
<b>DisplayName</b>	Megjelenített név
<b>EmailAddresses</b>	E-mail címek listája
<b>IsPinnedToStart</b>	Boolean érték, megmutatja, hogy kitűztük-e a felhasználó hivatkozását a kezdőlapra.
<b>Notes</b>	A partnerhez rögzített jegyzeteink listája
<b>PhoneNumbers</b>	A partner telefonszámai. Tartalmazzák a számot és a telefonszám típusát is (otthoni, céges, mobil, stb.)
<b>SignificantOthers</b>	A partner párkapcsolatai
<b>Websites</b>	Weblapok listája
<b>GetPicture()</b>	Ugyan nem tulajdonság, hanem metódus, de fontos megjegyezni, hogy a felhasználó képének lekérésére is van lehetőségünk.

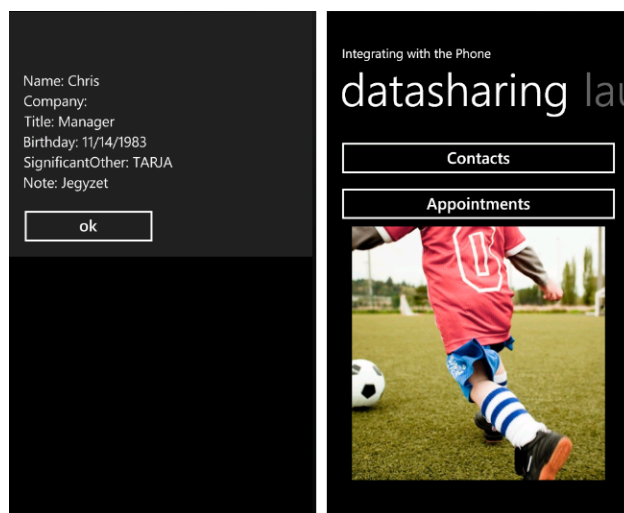
Az emulátoron szereplő kapcsolatlista jó kiindulási alap a teszteléshez, de a biztonság kedvéért egészítsük ki például Chris Sells adatait azokkal az információkkal, amiket le szeretnénk kérdezni!

A módosításokhoz használhatjuk akár a **SavePhoneNumberTaskot** is, mivel a telefonszám módosítását követően a **Contacts** alkalmazásban maradunk addig, amíg a Back gombbal vissza nem navigálunk az alkalmazásunkba. Itt válasszunk egy profilképet, és töltsük ki a **JobTitle**, **Birthday**, **SignificantOther** és **OfficeLocation** tulajdonságokat!

A módosításokat követően írjuk meg a **Contacts** gomb katt eseménykezelőjét!

```
private void btnContacts_Click(object sender, RoutedEventArgs e)
{
    Contacts contacts = new Contacts();
    contacts.SearchCompleted += (s, result) =>
    {
        var contact = result.Results.First();
        // felhasználó képének lekérése
        BitmapImage bmp = new BitmapImage();
        bmp.SetSource(contact.GetPicture());
        imgDataSharing.Source = bmp;
        // egyéb adatok
        string name = contact.CompleteName.FirstName;
        string company = contact.Companies.First().CompanyName;
        string title = contact.Companies.First().JobTitle;
        string birthday = contact.Birthdays.First().ToShortDateString();
        string significantOther = contact.SignificantOthers.First();
        string note = contact.Notes.First();
        // megjelenítés
        string formatStr = "Name: {0}\nCompany: {1}\nTitle: {2}\n" +
            "Birthday: {3}\nSignificantOther: {4}\nNote: {5}";
        MessageBox.Show(string.Format(formatStr,
            name, company, title, birthday,
            significantOther, note));
    };
    // csak a nevek között keresünk
    contacts.SearchAsync("c", FilterKind.DisplayName, null);
}
```

A fenti kódban minden listának csak az első elemét kértük le, de természetesen a keresés ettől függetlenül felhasználók listáját adja vissza, ahol a tulajdonságok többsége szintén egy **IEnumerable** példány. A függvény futásának eredménye a 6-14 ábrán látható.



**6-14 ábra: Partner adatainak lekérdezése PhoneDataSharingContext használatával**

### Appointments

A naptárbejegyzések lekérdezése a kapcsolatok kezeléséhez hasonlít, viszont mivel ezek az adatok mindenképpen felhasználói fiókhoz kötődnek (Live, Hotmail, Facebook, stb.), sajnos az emulátoron mindig egy üres listát fogunk visszakapni.

A bejegyzést megtestesítő **Appointment** osztály főbb tulajdonságai a következő táblázatban láthatók:

Tulajdonság	Leírás
Account	A naptárbejegyzést tároló felhasználói fiók
Attendees	Résztevő felhasználók ( <b>Attendee</b> lista)
Details	A bejegyzés szöveges leírása
StartTime	Kezdés ideje
EndTime	Befejezés ideje
IsAllDayEvent	Boolean, egész napos eseményről van-e szó?
IsPrivate	Mindenki által látható vagy privát esemény?
Location	A találkozó helye (szöveggént)
Organizer	A szervező személy ( <b>Attendee</b> példány)
Status	Részvételi állapotunk
Subject	A bejegyzés tárgya röviden

Alkalmazása tehát nagyon hasonlít a partneradatok lekérdezéséhez. A **SearchAsync()** hívása új szálon elindítja a keresést, a visszatérési értéket pedig a **SearchCompleted** eseménykezelőben használhatjuk fel:

```
private void btnAppointments_Click(object sender, RoutedEventArgs e)
{
    Appointments apps = new Appointments();
    apps.SearchCompleted += (s, result) =>
    {
        var events = result.Results;
        if (events != null && events.Count() > 0)
        {
            var myEvent = events.OrderBy(c => c.StartTime).First();
            string formatStr = @"A következő esemény {0} napon kezdődik,
                                tárgya: {1}, résztvevői pedig:\n{2}";
            StringBuilder builder = new StringBuilder();
            foreach (var item in myEvent.Attendees)
            {
                builder.AppendFormat("{0}\n", item.DisplayName);
            }
            MessageBox.Show(string.Format(formatStr,
                myEvent.StartTime.ToShortDateString(),
                myEvent.Subject, builder.ToString()));
        }
    };
    apps.SearchAsync(DateTime.Now, DateTime.Now.AddMonths(6), null);
}
```

## Kamera használata taszk nélkül

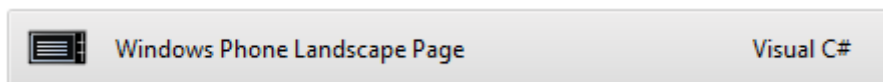
A **CameraCaptureTask** használatán túl lehetőségünk van közvetlenebb módon is kommunikálnunk a készülék kameraival. Az operációs rendszer Mango verziója ugyanis elérhetővé teszi számunkra a fényképek készítését lehetővé tevő **PhotoCamera** és a videorögzítést megkönnyítő **VideoCaptureDevice** osztályokat. Ezek működése nagyrészt hasonlít egymáshoz, a különbség főként az adatok tárolásakor mutatkozik meg. Mindkét esetben az Isolated Storage-be vagy a Media Library-ba menthetünk, de videó rögzítésekor a folyamatos adatfolyam miatt a **FileSink** segédosztályt is használnunk kell.

A következő példa során valós idejű képet fogunk mutatni a kamera képéről, és lehetőséget adunk fénykép készítésére akár a fő, hátlapon lévő kamerát, akár az előlapon lévő használva.

Mindenekelőtt a **WMAppManifest.xml** állományban jelezzük, hogy alkalmazásunk fogja használni a készülék kameráit. Ehhez egészítsük ki a **Capabilities** szekciót a két kamera azonosítójával (a hátlapon lévő kamera bejegyzése valószínűleg már a generált állományban is megtalálható):

```
<Capabilities>
  <Capability Name="ID_HW_FRONTCAMERA"/>
  <Capability Name="ID_CAP_ISV_CAMERA"/>
  ...
</Capabilities>
```

Vegyünk fel egy **Windows Phone Landscape Page**-et projektünkbe **Camera** névvel (6-15 ábra)! Ez a sablon egy olyan oldalt generál, amely csak Landscape módban jelenik meg, de természetesen ez két tulajdonság módosításával felüldefiniálható.



**6-15 ábra: Windows Phone Landscape Page sablon**

Mivel az oldalt a továbbiakban fektetett módban fogjuk használni, érdemes az emulátort is ehhez igazítani. Ehhez az emulátor jobb oldali menüjében találjuk a jobbra és balra forgató gombokat (6-16 ábra).



**6-16 ábra: Emulátor forgatása**

Adjunk hozzá egy gombot az eddig szerkesztett MainPage oldalhoz, ami az új **Camera.xaml** oldalunkra fog navigálni!

```
<Button Content="Camera" Name="btnCamera" Click="btnCamera_Click" />

// kódfájl:
private void btnCamera_Click(object sender, RoutedEventArgs e)
{
    NavigationService.Navigate(new Uri("/Camera.xaml", UriKind.Relative));
}
```

Mielőtt nekiállnánk a kódfájlban létrehozni minden objektumot és a funkcionalitást implementálni, gondoljuk végig, hogy mire lesz szükségünk! Mivel két kamera képét egyszerre nem tudjuk kezelni, lehetővé kell tennünk a felhasználó számára, hogy váltani tudjon azok között. Erre tökéletesen megfelel egy Slider, ami két értéket vehet fel. Azért is szerencsés ez a választás, mert az operációs rendszer legtöbb beállításánál is így vannak megvalósítva a kapcsolók, és az sohasem baj, ha a platformmal konzisztensek maradunk. A kapcsoló állásától függ a kirajzolt kép és az adatmentés forrása. Egy gombbal pedig

lehetőséget kell adnunk a fénykép készítésére is, viszont a feldolgozás idejére ezt inaktívvá is kell tennünk.

Látszik, hogy itt az előző taszkindításoknál bonyolultabb dolgunk lesz, például adatkötéseket is meg kell valósítanunk. Ennek érdekében a feladatot a fejezet elején leírt modell szerint egészítsük ki, azaz hozzunk létre egy **ViewModel**-t az oldal számára! A **ViewModels** mappához adjunk hozzá egy új osztályt **CameraViewModel** névvel, ez pedig valósítsa meg az **INotifyPropertyChanged** interfészt:

```
public class CameraViewModel : INotifyPropertyChanged
{
    public event PropertyChangedEventHandler PropertyChanged;
    private void NotifyPropertyChanged(String propertyName)
    {
        PropertyChangedEventHandler handler = PropertyChanged;
        if (null != handler)
        {
            handler(this, new PropertyChangedEventArgs(propertyName));
        }
    }
}
```

Ez az osztály tároljon minden tulajdonságot és logikát, amit a **Camera.xaml**-nek a megjelenítés során alkalmaznia kell! Vegyük is fel ezeket az alábbiak szerint:

```
public PhotoCamera Camera { get; set; }

private VideoBrush cameraBrush;
/// <summary>
/// Az élő képet megjelenítő VideoBrush
/// </summary>
public VideoBrush CameraBrush
{
    get { return cameraBrush; }
    set
    {
        cameraBrush = value;
        NotifyPropertyChanged("CameraBrush");
    }
}

private bool isCameraChooserEnabled;
/// <summary>
/// True, ha mindkét kamera elérhető, azaz a választó
/// slide-nak aktívnak kell lennie
/// </summary>
public bool IsCameraChooserEnabled
{
    get { return isCameraChooserEnabled; }
    set
    {
        isCameraChooserEnabled = value;
        Deployment.Current.Dispatcher.BeginInvoke(() =>
        {
            NotifyPropertyChanged("IsCameraChooserEnabled");
        });
    }
}

private bool isFrontCamera;
/// <summary>
/// True, ha az előlapi kamera aktív, egyébként false
/// </summary>
public bool IsFrontCamera
{

```

```
get { return isFrontCamera; }
set
{
    isCameraChooserEnabled = value;
    Deployment.Current.Dispatcher.BeginInvoke(() =>
    {
        NotifyPropertyChanged("IsFrontCamera");
    });
}
}

private bool isPhotoEnabled;
/// <summary>
/// True, ha készen állunk fénykép készítésére,
/// azaz a fotó gomb aktív
/// </summary>
public bool IsPhotoEnabled
{
    get { return isPhotoEnabled; }
    set
    {
        isPhotoEnabled = value;
        Deployment.Current.Dispatcher.BeginInvoke(() =>
        {
            NotifyPropertyChanged("IsPhotoEnabled");
        });
    }
}

/// <summary>
/// Fénykép készítése
/// </summary>
public void TakePhoto()
{
    // nem készíthetünk új képet, amíg ezt nem dolgoztuk fel
    IsPhotoEnabled = false;
    Camera.CaptureImage();
}
```

A kódban kiemelem azokat a tulajdonságokat, amelyeket kötnünk kell egy vezérlőhöz a megjelenítő oldalon. Ezeknél a **set** metódust úgy kell módosítani, hogy az értékadást követően egy **NotifyPropertyChanged** hívással jelezzék a mögöttes érték változását. Emiatt nem lehet a rövidített **public string PropertyName { get; set; }** formátumot használni, a teljes kifejtésre van szükség. Szerencsére ez a „prop” + TAB + TAB beírásával legenerálható, és így a gépelés nagy részét megúszhatjuk. Mivel a kamerával kapcsolatos feladatok nem az UI szálon futnak, szükségünk lesz a Dispatcher használatára, hogy a változásokat a felhasználói felületen is megjeleníthessük.

Kezdjük el leprogramozni a vezérlő logikáját! A két kamerát egyszerre nem tudjuk kezelni, így valahol meg kell oldanunk a kiválasztott eszköz eseményeinek figyelését, míg a másik kamerát ki kell kapcsolnunk. Magától értetődő választás erre az **IsFrontCamera** tulajdonság set metódusa, mivel a kamera kiválasztásakor közvetlenül ez az eljárás fog futni. Itt szabaduljunk meg egy **Dispose** hívással a korábbi kamerától, majd példányosítsunk egy megfelelő típusú **PhotoCamera**-t (**Primary, FrontFacing**)! Definiáljuk az új objektum **CaptureImageAvailable** eseménykezelőjét, ami a kép elkészültekor fog lefutni. Emellett gondoskodjunk az **IsCameraChooserEnabled** tulajdonság kezeléséről is, mivel az objektum inicializálása viszonylag sok időt vesz igénybe, és ha még közben kamerát váltunk, akkor a **Dispose()** hívás a félkész objektumon hibát fog eredményezni.

```
public bool IsFrontCamera
{
    get { return isFrontCamera; }
    set
```



```

{
    isFrontCamera = value;
    // előzőleg beállított kamera törlése, ha volt
    if (Camera != null)
    {
        Camera.Dispose();
    }

    if (isFrontCamera)
    { // előlapi kamera
        Camera = new PhotoCamera(CameraType.FrontFacing);
    }
    else
    { // hátlapi kamera
        Camera = new PhotoCamera(CameraType.Primary);
    }

    // Init lezárása előtt ne lehessen kamerát váltani (Dispose)
    IsCameraChooserEnabled = false;

    Camera.Initialized += (s, e) =>
    {
        if (PhotoCamera.IsCameraTypeSupported(CameraType.FrontFacing)
            && PhotoCamera.IsCameraTypeSupported(CameraType.Primary))
        {
            // kameraváltás újbóli engedélyezése
            IsCameraChooserEnabled = true;
        }
    };

    Camera.CaptureImageAvailable += (s, e) =>
    {
        string name = DateTime.Now.ToShortTimeString() + ".jpg";
        // TODO: mentés megírása

        // újra fotózhatunk
        IsPhotoEnabled = true;

        Deployment.Current.Dispatcher.BeginInvoke(() =>
        {
            // visszajelzés küldése (UI szál)
            MessageBox.Show(name + " mentve!");
        });
    };

    NotifyPropertyChanged("IsFrontCamera");
}
}

```

A kép már elkészül, azt viszont még csak a telefon pozíciójából tudjuk kikövetkeztetni, hogy mit fotóztunk. Ennek kiküszöbölésére csak annyi a teendőnk, hogy a korábban deklarált ecsetet összekötjük a kamerával, még ugyanebben a set metódusban, például a **NotifyPropertyChanged("IsFrontCamera");** hívása előtt:

```

// ...
// VideoBrush beállítása a kiválasztott kamerára
cameraBrush.SetSource(Camera);
NotifyPropertyChanged("IsFrontCamera");
}

```

A tulajdonságok halmaza és a logika már kész, már csak inicializálni kell az értékeket. Vegyünk fel erre egy új függvényt, **LoadData** névvel:

```
public void LoadData()
{
    CameraBrush = new VideoBrush();
    IsPhotoEnabled = true;

    if (PhotoCamera.IsCameraTypeSupported(CameraType.FrontFacing)
        && PhotoCamera.IsCameraTypeSupported(CameraType.Primary))
    {
        // mindkét kamera elérhető --> választhassunk
        IsCameraChooserEnabled = true;
        IsFrontCamera = true;
    }
    else
    {
        // csak az egyik kamera érhető el (vagy egyik sem)
        IsCameraChooserEnabled = false;
        if (PhotoCamera.IsCameraTypeSupported(CameraType.FrontFacing))
        {
            IsFrontCamera = true;
        }
        else if (PhotoCamera.IsCameraTypeSupported(CameraType.Primary))
        {
            IsFrontCamera = false;
        }
        else
        {
            MessageBox.Show("Nincs elérhető kamera!");
        }
    }
}
```

Ennek az eljárásnak az a feladata, hogy kiderítse, hozzáférünk-e egyáltalán a kamerákhoz, és ennek megfelelően inicializálja a tulajdonságokat.

Nem lenne szép megoldás, ha ezt a **ViewModel**-t egy **Camera** objektum példányosítaná, ehelyett vegyünk fel egy statikus tulajdonságot az **App** osztályban, amin keresztül a **View** eléri a hozzá tartozó **ViewModel**-t:

```
private static CameraViewModel cameraVM;
public static CameraViewModel CameraVM
{
    get
    {
        if (cameraVM == null)
        {
            cameraVM = new CameraViewModel();
        }
        return cameraVM;
    }
}
```

A **Camera.xaml** kódfájljában pedig módosítsuk úgy a konstruktort, hogy az kérje el az **App** osztálytól a statikus **ViewModel**-t, majd ezt állítsa be **DataContext**-nek! Amint az oldal betöltődött, inicializáljuk az értékeket a **LoadData** meghívásával:

```
CameraViewModel ViewModel;

public Camera()
{
    InitializeComponent();

    ViewModel = App.CameraVM;
    DataContext = ViewModel;
}
```

```

this.Loaded += (s, e) =>
{
    ViewModel.LoadData();
};
}

```

Ezzel be is fejeztük a logika megírását és a **ViewModel** „behuzalozását”, már csupán a megjelenítő felületet kell létrehoznunk. A fektetett megjelenítés miatt a bal oldalra kerülhet az élőképet mutató **Canvas**, jobbra mellé pedig a vezérlő gombok:

```

<Grid x:Name="LayoutRoot" Background="Transparent">
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="600" />
        <ColumnDefinition Width="*" />
    </Grid.ColumnDefinitions>

    <!--ContentPanel - place additional content here-->
    <Canvas Name="ContentCanvas" Width="600" Height="440"
        Background="{Binding CameraBrush}">
    </Canvas>

    <StackPanel Grid.Column="1">
        <Button Name="btnPhoto" Content="Kép"
            Click="btnPhoto_Click"
            IsEnabled="{Binding IsPhotoEnabled}"/>
        <Slider Value="{Binding IsFrontCamera, Mode=TwoWay}"
            IsEnabled="{Binding IsCameraChooserEnabled}"
            Minimum="0" Maximum="1"/>
    </StackPanel>
</Grid>

```

A **CameraBrush** háttérnek való bekötésével festjük fel a pillanatnyi képet a **Canvas** felületére, a gombhoz pedig bekötjük az **IsPhotoEnabled** tulajdonságot, így az csak akkor lesz aktív, amikor nincs folyamatban lévő képfeldolgozás. A **Slider** lesz az a vezérlő, amellyel a kamerák között válthatunk, amennyiben az **IsCameraChooserEnabled** tulajdonság **true** értéket tartalmaz. Fontos, hogy az adatkötés kétirányú legyen, így a vezérlő értékének módosításával a mögöttes tulajdonság értéke is változik!

Egyetlen feladatunk maradt hátra, a képkészítést indító gomb eseménykezelőjének megírása. Mivel a **ViewModel** megvalósítja a műveletet, itt csupán egy függvényhívásra van szükségünk:

```

private void btnPhoto_Click(object sender, RoutedEventArgs e)
{
    ViewModel.TakePhoto();
}

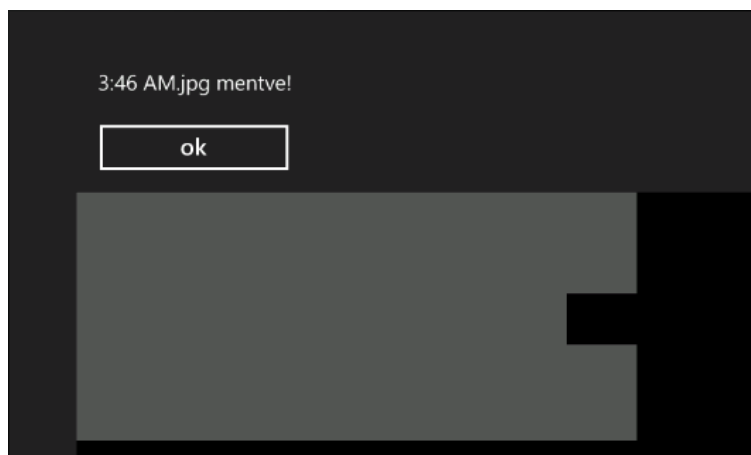
```

Ha sehol nem hibáztunk, akkor a projekt telepítése után a 6-17 ábrán látható elrendezést fogjuk látni. A kiválasztott kamerát az alapján tudjuk azonosítani, hogy a mozgóképet jelképező fekete téglalap melyik irányban mozog. Az előlapon lévő kamera kiválasztása esetén ez a mozgás az óramutató járásával ellentétes lesz, a hátlapon lévő fő kamera esetében pedig azzal megegyező.



**6-17 ábra: A kamerák képét mutató alkalmazás felülete**

A slider értékének megváltoztatását követően a téglalap irányt vált, amiből tudhatjuk, hogy a kamera váltása megtörtént. Kép készítésekor a gomb egy időre inaktívvá válik, majd a feldolgozás végén egy üzenetablakban kapunk értesítést a feladat befejeződéséről (6-18 ábra).



**6-18 ábra: Értesítés a feldolgozás befejeződéséről**

Már csak annyi maradt hátra, hogy a kapott képpel kezdjünk is valamit, például tároljuk el a készülék médiatárában. Ehhez egy **MediaLibrary** példányra lesz szükségünk, amelyen keresztül hozzáférhetünk a telefonon tárolt zenékhez és képekhez. Az osztály a **Microsoft.Xna.Framework** szerelvény része, ezért hivatkozzunk erre a dll-re projektünkben! A tárolás szintén a **ViewModel** dolga, így a **MediaLibrary** típusú **library** objektumot a **CameraViewModel** osztályban deklaráljuk:

```
// Microsoft.Xna.Framework dll!  
MediaLibrary library = new MediaLibrary();
```

A mentés akkor történjen, amikor a kép már elkészült, de a felhasználó számára még nem küldtük el az értesítést erről! Így hibakezelésre is lehetőségünk van – de ettől most eltekintünk. A mentés helye a **CaptureImageAvailable** eseménykezelő, ahol korábban már egy **// TODO: mentés megírása** sort hagytunk, ezt cseréljük most ki az alábbiaknak megfelelően:

```

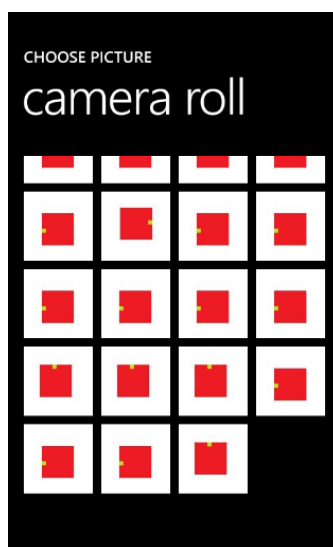
Camera.CaptureImageAvailable += (s, e) =>
{
    string name = DateTime.Now.ToShortTimeString() + ".jpg";
    // mentés a médiatárba
    library.SavePictureToCameraRoll(name, e.ImageStream);

    // újra fotózhatunk
    IsPhotoEnabled = true;

    Deployment.Current.Dispatcher.BeginInvoke(() =>
    {
        // visszajelzés küldése (UI szál)
        MessageBox.Show(name + " mentve!");
    });
};

```

A funkció működését a fejezet korábbi részeiben bemutatott **PhotoChooserTask** segítségével ellenőrizhetjük. Az én emulátoromon például a 6-19 ábra művészeti remekművei készültek.



**6-19 ábra: A kamerák által készített képek**

Az emulátor ezeknél a képeknél is különbséget tesz a két eszköz között, az előlapon lévő kamera képét ugyanis egy bal oldalra mutató sárga pötty, míg a hátlapon lévőét egy felfelé mutató jelképezi.

## Összefoglalás

A fejezet során rengeteg olyan lehetőséggel ismerkedtünk meg, amelyek egyszerű hozzáférést biztosítanak a telefon valamely funkciójához. Végigvettük az erre kínált taszkokat, megkülönböztetve a visszatérési értéket is szolgáltató choosereket és a csupán feladatindítást kínáló launchereket. Ezt követően megnéztük a partner- és naptárbejegyzési adatok elérésére megoldást kínáló egyéb lehetőségeket is a **PhoneDataSharingContext** használatával. A fejezet végén pedig egy példaalkalmazás írásán keresztül láthattuk, hogy hogyan tudunk a taszkoknál közvetlenebb módon hozzáférni a kamerákhoz és azok képét a médiatárban rögzíteni.



# 7. További telefonos funkciók használata

Eddig jobbra a telefon szoftveres képességeivel játszottunk, most azonban egy kicsit mélyebbre hatolunk. Ebben a fejezetben azt vizsgáljuk, hogyan tudjuk kihasználni a Windows Phone 7.5 operációs rendszert futtató telefonokba alapfelszereltségként beépített szenzorokat, vagyis érzékelőket.

## Az érintőképernyő kezelése

Elsőként azzal az érzékelővel foglalkozunk, amely ma már elengedhetetlen része okostelefonjainknak, és amelyet a felhasználó lassan már anélkül használ, hogy egyáltalán észrevenné azt. Ez az érintőképernyő – kétségkívül a legfontosabb szenzor. Bár ma teljesen természetesnek számít, hogy ott van a telefonban, néhány évvel ezelőtt még ritkán építették bele.

A Silverlight for Windows Phone több szinten teszi elérhetővé a fejlesztő számára az érintőképernyő eseményeit. Attól függően, hogy milyen bonyolultan szeretnénk hozzáférni, figyelhetünk az egyszerű eseményekre, mint a **Tap** (érintés) és társai, lemehetünk egy szinttel mélyebbre a **Manipulation** eseményekhez, vagy a legmélyebb szintre, a **Touch.FrameReported**ig, ahol a nyers adatokkal dolgozhatunk. Minél mélyebb szinten programozunk, annál pontosabb, bonyolultabb algoritmusokat építhetünk az eseményekre, de annál több munkánk is lesz vele.

## Alapvető érintési események

Ezeknek az eseménynek a kezelői egy **GestureEventArgs** típusú paraméterrel rendelkeznek, mely – azonfelül, hogy ismerik a **RoutedEventArgs** leszármazottakra jellemző **Handled**, illetve **OriginalSource** tulajdonságokat – tulajdonképpen csak annyi hasznos információval szolgál, hogy lekérdezhető belőle az esemény bekövetkezésének helye. Tegyük hozzá: elég gyakran még ez az információ sem érdekes.

Az esemény kiváltásának helyét a **GetPosition** metódussal kérdezhetjük le, amely egy **UIElement** típusú paramétert vár; ehhez a vizuális elemhez képest adja vissza az érintés pozícióját. Ebbe a kategóriába három esemény tartozik:

7-1 táblázat: Érintési események

Esemény	Leírás
<b>Tap</b>	A <b>Tap</b> esemény akkor következik be, amikor a felhasználó megérinti egy elem felületét – tulajdonképpen ez a kattintás megfelelője az érintőképernyőn, azzal a különbséggel, hogy <b>Tap</b> esetén nem lehet megkülönböztetni bal/jobbs egérgombot és egyéb egérre jellemző tulajdonságokat, és ennek az érintés miatt értelme sincsen.
<b>DoubleTap</b>	Szintén a <b>UIElement</b> ből származó vezérlőkön és vizuális elemeken értelmezett a <b>DoubleTap</b> esemény. Akkor következik be, amikor a felhasználó rövid időn belül kétszer megérinti egy vizuális elem felületét.
<b>Hold</b>	A harmadik érintési esemény, amit a <b>UIElement</b> osztályon definiáltak, a <b>Hold</b> . Ez akkor következik be, amikor a felhasználó megérint egy vizuális elemet, de ujját nem emeli fel, hanem kb. egy másodpercig rajta hagyja az elemen.

Lássuk, hogyan lehet felhasználni ezeket az eseményeket!

## 7. További telefonos funkciók használata

Készítsünk egy Windows Phone Application sablonra épülő alkalmazást, és helyezzünk el benne –konkrétan a ContentPanel nevű Gridben – egy négyzetet!

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <Rectangle x:Name="rect" Width="100" Height="100"
        Fill="{StaticResource PhoneAccentBrush}" />
</Grid>
```

A megoldandó feladat annyi, hogy tapintásra a négyzet forduljon el középpontja körül 10 fokkal jobbra! Ehhez arra lesz szükség, hogy elhelyezzünk a négyzetben egy transzformációt, amit állíthatunk.

Forgatásra a **RotateTransform** a legmegfelelőbb, de alkalmazhatunk **CompositeTransform**ot, vagy ha több különálló transzformációra lesz szükségünk, **TransformGroup**ot is.

A transzformációt a **UIElement RenderTransform** tulajdonságán keresztül tudjuk beállítani. Az alábbi kód szemlélteti, hogy hogyan tudunk ilyen transzformációt létrehozni XAML-ben:

```
<Rectangle x:Name="rect" Width="100" Height="100"
    Fill="{StaticResource PhoneAccentBrush}">
    <Rectangle.RenderTransform>
        <RotateTransform Angle="0"/>
    </Rectangle.RenderTransform>
</Rectangle>
```

A forgatás alapesetben a vizuális elem befoglaló négyzetének bal felső sarka körül történik. Ezt úgy tudjuk átállítani az elem középpontjára, hogy átállítjuk a **Rectangle RenderTransformOrigin** tulajdonságát. Itt 0 és 1 között relatívan adhatjuk meg, hogy az egyes tengelyeken hol helyezkedjen el a középpont.

```
<Rectangle x:Name="rect" Width="100" Height="100"
    Fill="{StaticResource PhoneAccentBrush}" RenderTransformOrigin=".5,.5">
    ...
</Rectangle>
```

Ezután már csak fel kell iratkozni a **Tap** eseményre – ezt a szokásos módon tehetjük meg:

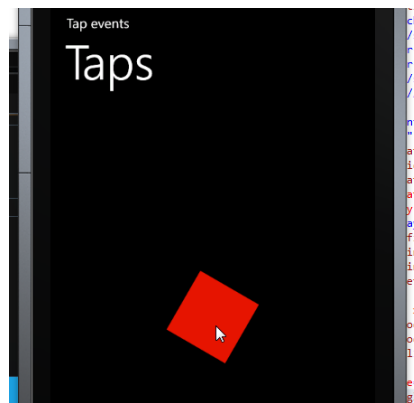
```
<Rectangle x:Name="rect" Width="100" Height="100"
    Fill="{StaticResource PhoneAccentBrush}"
    Tap="rect_Tap" RenderTransformOrigin=".5,.5">
    ...
</Rectangle>
```

A mögöttes kódban pedig csak módosítanunk kell az előbb létrehozott transzformáció **Angle** tulajdonságát. Ez fokokban adja meg, hogy mennyivel kell elfordulnia az elemnek a **RenderTransformOrigin**ben megadott pont körül. Természetesen a **RenderTransform** tulajdonság nem **RotateTransform** típusú – elvégre 5-6 másik transzformáció-típust is tudnia kell fogadni –, így át kell alakítanunk, hogy az **Angle**-t lássuk:

```
private void rect_Tap(object sender, GestureEventArgs e)
{
    ((RotateTransform)rect.RenderTransform).Angle += 10;
}
```

Teszteljük le az alkalmazást! A négyzet nyomkodás hatására forog (7-1 ábra).





7-1 ábra: A Tap esemény hatása

Ha egy transzformációt gyakran akarunk a mögöttes kódban használni, egyszerűbb, ha adunk neki egy nevet az **x:Name** tulajdonságon keresztül. Így megkímélhetjük magunkat a sok típuskonverziótól.

Oldjuk meg azt is, hogy a négyzet nyomva tartása (**Hold**) esetén a négyzet nagyobbra nőjön, két gyors egymás utáni tapintás (**DoubleTap**) pedig alaphelyzetbe állítsa!

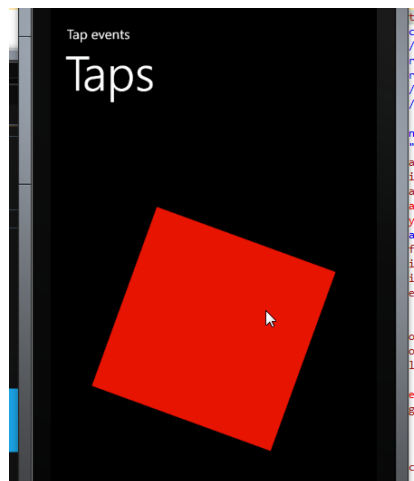
Iratkozzunk fel a Hold eseményre!

```
<Rectangle x:Name="rect" Width="100" Height="100"
    Fill="{StaticResource PhoneAccentBrush}"
    Tap="rect_Tap" Hold="rect_Hold"
    RenderTransformOrigin=".5,.5">
    ...
</Rectangle>
```

Az eseménykezelőben növeljük meg a négyzet méretét!

```
private void rect_Hold(object sender, GestureEventArgs e)
{
    rect.Width += 20;
    rect.Height += 20;
}
```

Tesztelésnél látható, hogy a Hold esemény hatására a négyzet 20-20 képpontonként nő. (Emellett persze továbbra is működik a Tap, tehát ha 1 mp-en belül felengedjük az ujjunkat, akkor elfordul a négyzet.)



7-2 ábra – A Hold (és a Tap) esemény hatása

A **DoubleTap** megoldása is az előbb leírtakat követi. Először is fel kell iratkozni az eseményre:

```
<Rectangle x:Name="rect" Width="100" Height="100"
    Fill="{StaticResource PhoneAccentBrush}"
    Tap="rect_Tap" DoubleTap="rect_DoubleTap" Hold="rect_Hold"
    RenderTransformOrigin=".5,.5">
    <Rectangle.RenderTransform>
    <RotateTransform Angle="0"/>
    </Rectangle.RenderTransform>
</Rectangle>
```

Ezután a mögöttes kódban vissza kell állítani alaphelyzetbe az előzőleg megváltoztatott három tulajdonságot:

```
private void rect_DoubleTap(object sender, GestureEventArgs e)
{
    rect.Width = 100;
    rect.Height = 100;
    ((RotateTransform)rect.RenderTransform).Angle = 0;
}
```

### Multitouch manipulációk

Az előző események egyetlen ujj érintését jelzik vissza felénk. A Windows Phone-nal felszerelt telefonokban alapkövetelmény a legalább négy érintési pontot kezelni képes kapacitív kijelző, így elviekben semmi nem gátolja a felhasználót abban, hogy előre megfontolt szándékkal egyszerre több ujjával is megérintse a kijelzőt.

Az események második szintje a többpontos (*multitouch*) érintések kezelését is lehetővé tevő úgynevezett **Manipulation** események csoportja. Itt is három eseményt találunk:

7-2 táblázat: Manipulation események

Esemény	Leírás
<b>ManipulationStarted</b>	A <b>ManipulationStarted</b> esemény akkor következik be, amikor a felhasználó megérinti a kijelzőt. Ehhez nem kell több ujját használnia – már egy érintéstől is kiváltásra kerül az esemény. Ennek egyik paramétere egy <b>ManipulationStartedEventArgs</b> típusú objektum. Ennek <b>ManipulationContainer</b> tulajdonsága megadja, hogy mely objektumhoz viszonyítva történik a manipuláció (ez maga a manipulált objektum), illetve a <b>ManipulationOrigin</b> tulajdonság, ami a manipulálás indításakor a <b>ManipulationContainer</b> bal felső sarkához viszonyított pozíciót adja meg – tehát hogy hol nyomták meg az elemet, amit megnyomtak.  Hasznos lehet még tudni, hogy szintén a <b>ManipulationStartedEventArgs</b> része a <b>Complete</b> metódus. Ha ezt meghívjuk, az operációs rendszer lezártnak tekinti a manipulációt, tehát ha a felhasználó továbbra is a kijelzőn tartja ujját, nem a <b>ManipulationDelta</b> , hanem a <b>ManipulationCompleted</b> esemény fut majd le.
	Készítsük be a telefont, ha többérintéses eseményeket szeretnénk tesztelni – az emulátor nem támogatja a több érintést.

Esemény	Leírás
<b>ManipulationDelta</b>	<p>A manipuláció időtartama alatt, vagyis amíg legalább egy ujj érinti a kijelzőt, a <b>ManipulationDelta</b> eseményt kaphatjuk el. Ennek <b>ManipulationDeltaEventArgs</b> paramétere szintén tartalmaz egy <b>ManipulationContainer</b> és egy <b>ManipulationOrigin</b> tulajdonságot, illetve <b>Complete</b> metódust, amelyek a korábban ismertetettel egyező működésűek.</p> <p>Ezenkívül két talán legfontosabb tulajdonsága a <b>CumulativeManipulation</b> és a <b>DeltaManipulation</b>. Típusuk megegyezik: <b>ManipulationDelta</b>, amely a több ujj által kifejtett manipulációt összegzi. Ennek <b>Scale</b> tulajdonságából megkapjuk azt, hogy a több ujj arányában mennyit távolodott egymástól vagy közeledett egymáshoz, <b>Translation</b> tulajdonságából pedig, hogy mennyit tolódott el az ujjak által kijelölt középpont. A különbség a <b>CumulativeManipulation</b> és a <b>DeltaManipulation</b> között, hogy előbbi a manipuláció kezdete óta összegyűlt eltéréseket mutatja, míg utóbbi csak annak változását, vagyis a legutóbbi <b>ManipulationDelta</b> esemény óta történt elmozdulást.</p> <p>Említésre méltó ezenfelül a <b>ManipulationDeltaEventArgs Velocities</b> tulajdonsága, melyből az ujjak egymástól való távolodásának (<b>ExpansionVelocity</b>) és az eltolásnak a (<b>LinearVelocity</b>) sebességét kaphatjuk meg.</p>
<b>ManipulationCompleted</b>	<p>Amikor az utolsó ujj is elhagyja az érintőképernyőt, a <b>ManipulationCompleted</b> eseményt jelzi a rendszer. Ennek <b>ManipulationCompletedEventArgs</b> típusú paramétere egy <b>TotalManipulation</b> nevű, <b>ManipulationDelta</b> típusú tulajdonságon keresztül adja tudtunkra, hogy a manipuláció kezdete óta mekkora eltolás és átméretezés történt, <b>FinalVelocities</b> tulajdonságával pedig azt mutatja, hogy az utolsó ujj felemelésekor milyen sebességű volt az átméretezés és eltolás.</p>

A következőkben arra fogjuk felhasználni ezt a három eseményt, hogy egy „Drag” műveletet hajtsunk végre, vagyis lehetővé tegyünk, hogy a felhasználó ujjmozdulatok segítségével húzzon arrébb egy objektumot a felületen. Emellett pedig a felhasználó két ujjának közelítésével vagy távolításával képes lesz növelni és csökkenteni a négyzet méretét.

A Windows Phone Application sablon alapján készített alkalmazás **MainPage** lapjának **ContentPanel**jébe ezúttal egy **Bordert** helyezzünk el, magát a **ContentPanel**t pedig módosítsuk **Grid**ről **Canvas** típusúra!

```
<Canvas x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <Border Name="brd" Width="100" Height="100" BorderBrush="BlanchedAlmond"
        Background="{StaticResource PhoneAccentBrush}" />
</Canvas>
```

Az átméretezést egy **ScaleTransform** segítségével szeretnénk végrehajtani – egészítsük ki ezzel a **Bordert**!

```
<Canvas x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <Border Name="brd" Width="100" Height="100" BorderBrush="BlanchedAlmond"
        Background="{StaticResource PhoneAccentBrush}"
        RenderTransformOrigin=".5,.5">
        <Border.RenderTransform>
            <ScaleTransform x:Name="st" />
        </Border.RenderTransform>
    </Border>
</Canvas>
```

Iratkozzunk fel a három **Manipulation** eseményre a **Border**en!

```
<Border Name="brd" Width="100" Height="100" BorderBrush="BlanchedAlmond"
    Background="{StaticResource PhoneAccentBrush}"
    ManipulationStarted="brd_ManipulationStarted"
    ManipulationDelta="brd_ManipulationDelta"
    ManipulationCompleted="brd_ManipulationCompleted">
    ...
</Border>
```

A manipuláció indulásakor és végén vizuális visszajelzést akarunk adni a felhasználónak arról, hogy ha mozgatja ujjait, melyik objektum változik. Induláskor láthatóvá tesszük a **Border** keretét, a végén pedig ismét eltüntetjük. Csak egy-egy sor nem túl bonyolult C#-kódra lesz szükségünk:

```
private void brd_ManipulationStarted(object sender, ManipulationStartedEventArgs e)
{
    brd.BorderThickness = new Thickness(5);
}

private void brd_ManipulationCompleted(object sender, ManipulationCompletedEventArgs e)
{
    brd.BorderThickness = new Thickness(0);
}
```

A **ManipulationDelta**, vagyis az ujjak elmozdulását jelző esemény kezelőjének megírása ennél kicsit bonyolultabb. Első körben csak az elmozdítást (vagyis a „Drag” effekt megvalósítását) nézzük meg. Az eltolást a **brd** nevű **Border**ünk **Canvas.Left** és **Canvas.Top** tulajdonságainak változtatásával állítjuk – vagyis a **Border**t az őt tartalmazó **Canvas** bal felső sarkához képest mozgatjuk. Mivel ez a tulajdonság ún. *attached property* (csatolt tulajdonság, amelyet nem az az objektum definiál, amelyen használjuk), nem közvetlenül a **Border**en találjuk, hanem a **GetValue** és **SetValue** metódusok segítségével kérhetjük le, illetve állíthatjuk be:

```
private void brd_ManipulationDelta(object sender, ManipulationDeltaEventArgs e)
{
    brd.SetValue(Canvas.LeftProperty, (double)brd.GetValue(
        Canvas.LeftProperty) + e.DeltaManipulation.Translation.X * st.ScaleX);
    brd.SetValue(Canvas.TopProperty, (double)brd.GetValue(
        Canvas.TopProperty) + e.DeltaManipulation.Translation.Y * st.ScaleY);
}
```

Mivel minden egyes elmozduláskor szeretnénk az előző pozícióhoz képest elmozdítani a vizuális elemet, a **DeltaManipulation**ot vesszük igénybe. Ennek **Translation** tulajdonsága egy **X** és egy **Y** értékkel adja meg, hogy hány képponttal kell arrébb mozgatnunk a **Border**t.

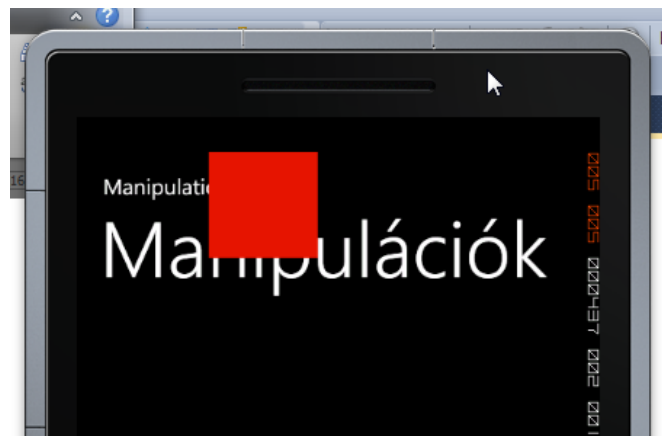
Hasonló módon valósíthatjuk meg a többujjas átméretezést. Itt a **Border**be ágyazott **ScaleTransform** objektum **ScaleX** és **ScaleY** tulajdonságát állítgatjuk. Mivel arányokról van szó (az esemény argumentuma azt adja meg, hogy a korábbi hányszorosára nőtt a méret), nem összeadunk, hanem az előbb említett tulajdonságokat felszorozzuk a megfelelő skálázási komponenssel. Külön kell kezelnünk a

nulla átméretezés esetét: ki kell szűrni, különben a négyzet eltűnne, amikor az ujjak közötti távolság nem változna érzékelhető mértékben.

```
private void brd_ManipulationDelta(object sender, ManipulationDeltaEventArgs e)
{
    brd.SetValue(Canvas.LeftProperty, (double)brd.GetValue(
        Canvas.LeftProperty) + e.DeltaManipulation.Translation.X * st.ScaleX);
    brd.SetValue(Canvas.TopProperty, (double)brd.GetValue(
        Canvas.TopProperty) + e.DeltaManipulation.Translation.Y * st.ScaleY);

    if (e.DeltaManipulation.Scale.X != 0) st.ScaleX *= e.DeltaManipulation.Scale.X;
    if (e.DeltaManipulation.Scale.Y != 0) st.ScaleY *= e.DeltaManipulation.Scale.Y;
}
```

Ennyi az egész, a program tesztelhető! Figyeljük meg, hogy az elmozdítás történhet a **Canvas**hoz képest akár negatív irányba is, vagyis ki tudjuk rángatni a **Bordert** a **Canvas**ról! Ha az átméretezés működését is tesztelni szeretnénk, mindenképpen szükségünk lesz egy telefonra is (7-3 ábra).



7-3 ábra: A négyzet, eredeti pozíciójából elmozgatva

## A nyers érintési adatok kezelése

Ahogy láthattuk, a **Manipulation** események segítségével jó néhány, magasabb szinten nem támogatott ujjmozdulat kezelése megvalósítható.

Azonban ekkor nem férünk hozzá a részletekhez, amelyek leírják, hogy pontosan mi is történik az érintőfelületen. Az események csak annyit közölnek velünk, hogy a több ujj elmozdulásából mit számoltak ki. Gyakran ez elég is, akad azonban olyan szituáció, amikor pontosabb, mélyebb szintű adatokra van szükségünk. Ekkor siet segítségünkre a **Touch** osztály.

### A **Touch.FrameReported** esemény

A **Touch** osztály csupán egyetlen eseménnyel bír, ez a **FrameReported**. Bármilyen változás történik az érintőfelületen – egy ujj megérinti a képernyőt, elmozdul, stb. –, kapunk róla egy **FrameReported** eseményt. Ennek eseményargumentuma **TouchFrameEventArgs** típusú. Egyetlen tulajdonsága a **TimeStamp** nevű integer, melynek segítségével az események egymástól való időben egymástól való távolságát mérhetjük le.

Ha ki akarjuk kapcsolni, hogy a magasabb szintű eseményeket is feldobja a rendszer, a **TouchFrameEventArgs** osztály **SuspendMousePromotionUntilTouchUp** metódusának meghívásával érhetjük ezt el. Amíg minden ujj el nem engedi a kijelzőt, csak **FrameReported** eseményeket kapunk.

A **TouchFrameEventArgs** két olyan metódust biztosít, amelyekkel követhetjük az ujjak pillanatnyi állapotát. A **GetPrimaryTouchPoint** az elsődleges érintési pontot adja vissza – amelyik legkorábban megérintette a képernyőt. A **GetTouchPoints** az összes aktuális érintési pontot visszaadja. Mindkét metódus egy **UIElement** paramétert vár: ehhez az elemhez képest adja vissza a pontokat.

Az előbbi metódusok visszatérési típusa **TouchPoint**, illetve **TouchPointCollection**. A **TouchPoint** típusú objektumok **Action** tulajdonsága megadja, hogy éppen milyen esemény történt az adott ponton. Ennek három értéke lehet: **Down** (érintés), **Move** (elmozdulás) és **Up** (felengedés). A **Position** tulajdonság megadja, hogy hol történt az esemény, a **Size** pedig, hogy mekkora a lenyomott felület. A **TouchDevice** tulajdonság kicsit félrevezető nevű: nem az érintést kiváltó eszközt azonosítja (hiszen az mindig ugyanaz lenne), hanem az érintést kiváltó ujjat. Ezzel tudjuk megkülönböztetni, hogy az egyes **TouchPoint**ok mely ujjhoz tartoznak.

Ennyi alapozás után már csak egy kis logika kell egy faék egyszerűségű, de több érintést kezelő rajzprogram létrehozásához. Amikor a felhasználó megérinti a képernyőt, egy nagyobbacska kört jelenítünk meg az ujjá alatt, majd ahogy mozgatja az ujját a kijelzőn, kisebb köröket. Amikor egy ujját felengedi, az ahhoz az ujjhoz tartozó köröket eltüntetjük. Emellett persze az egyazon ujjhoz tartozó körök mindig ugyanazzal a színnel jelennek meg, de minden ujjé mással.

A Windows Phone Application sablonnal létrehozott alkalmazás **MainPage.xaml** fájljában cseréljük le a **ContentPanel** típusát **Canvas**ra! A XAML-ben ezzel el is végeztük minden dolgunkat:

```
<Canvas x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0"></Canvas>
```

A mögöttes kódban először is fel kell iratkoznunk a **Touch** osztály **FrameReported** eseményére. Ezt például a **MainPage** konstruktorában tehetjük meg:

```
public MainPage()
{
    InitializeComponent();
    Touch.FrameReported += new TouchFrameEventHandler(Touch_FrameReported);
}
```

Ezután hozzuk létre az eseménykezelő metódust, és kérjük le benne az érintési pontokat a **ContentPanel** (a **Canvasunk**) bal felső sarkához képest!

```
void Touch_FrameReported(object sender, TouchFrameEventArgs e)
{
    var pts = e.GetTouchPoints(ContentPanel);
}
```

Ezután menjünk végig az érintési pontokon! Először azokat kezeljük, amelyek újak, vagyis ahol éppen megérintették a képernyőt! Ezt az **Action** tulajdonság **Down** értéke jelzi. Rajzoljunk egy 50\*50 képpont méretű kört a lenyomás helyén! A színt egy kis segédosztállyal választjuk ki, amelyet kézzel írunk meg. Minimális **Reflection**nel, a **Colors** felsorolt típus tulajdonságainak megszámlálásával visszaadunk egy színt az integer értékből, mindezt egy bővítő függvénnyel:

```
using System.Linq;
using System.Windows.Media;

namespace LowLevelTouch
{
    /// <summary>
    /// Integerek színné alakítása.
    /// </summary>
    public static class IntColorConversion
    {
        static int colorscount = 0;

        static IntColorConversion()
        {
            colorscount = typeof(Colors).GetProperties().Count();
        }
    }
}
```

```

    public static Color ToColor(this int id)
    {
        return (Color)typeof(Colors).GetProperties()[id + 1 % colorscount]
            .GetValue(null, null);
    }
}

```

Ez után a kis kitérő után már meg tudjuk írni a kört létrehozó kódot. Két apróságra ügyeljünk: tároljuk el az újonnan létrehozott kör **Tag** tulajdonságában a **TouchDevice.Id**-t, illetve ne felejtsük el megjeleníteni a kört azzal, hogy hozzáadjuk a **ContentPanel** gyermekelemeihez!

```

void Touch_FrameReported(object sender, TouchFrameEventArgs e)
{
    var pts = e.GetTouchPoints(ContentPanel);

    foreach (TouchPoint p in pts)
    {
        if (p.Action == TouchAction.Down)
        {
            Ellipse el = new Ellipse { Width = 50, Height = 50, Tag = p.TouchDevice.Id,
                Fill = new SolidColorBrush(p.TouchDevice.Id.ToColor()) };
            el.SetValue(Canvas.LeftProperty, p.Position.X - 25);
            el.SetValue(Canvas.TopProperty, p.Position.Y - 25);
            ContentPanel.Children.Add(el);
        }
    }
}

```

A **Tag** tulajdonság igazi funkcióval nem rendelkezik – pont olyan helyzetekre készült, amikor szeretnénk valamilyen, a vizuális elemhez kapcsolódó adatot, objektumot eltárolni, hozzárendelni az elemhez.

A már korábban a kijelzőn lévő ujjak további útjának kirajzolása hasonló elven történik. A **Move Action** értékkel rendelkező elemek pozíciójára egy-egy kisebb kört rajzolunk:

```

void Touch_FrameReported(object sender, TouchFrameEventArgs e)
{
    var pts = e.GetTouchPoints(ContentPanel);

    foreach (TouchPoint p in pts)
    {
        if (p.Action == TouchAction.Down)
        {
            Ellipse el = new Ellipse { Width = 50, Height = 50, Tag = p.TouchDevice.Id,
                Fill = new SolidColorBrush(p.TouchDevice.Id.ToColor()) };
            el.SetValue(Canvas.LeftProperty, p.Position.X - 25);
            el.SetValue(Canvas.TopProperty, p.Position.Y - 25);
            ContentPanel.Children.Add(el);
        }
        else if (p.Action == TouchAction.Move)
        {
            Ellipse el = new Ellipse { Width = 20, Height = 20, Tag = p.TouchDevice.Id,
                Fill = new SolidColorBrush(p.TouchDevice.Id.ToColor()) };
            el.SetValue(Canvas.LeftProperty, p.Position.X - 10);
            el.SetValue(Canvas.TopProperty, p.Position.Y - 10);
            ContentPanel.Children.Add(el);
        }
    }
}

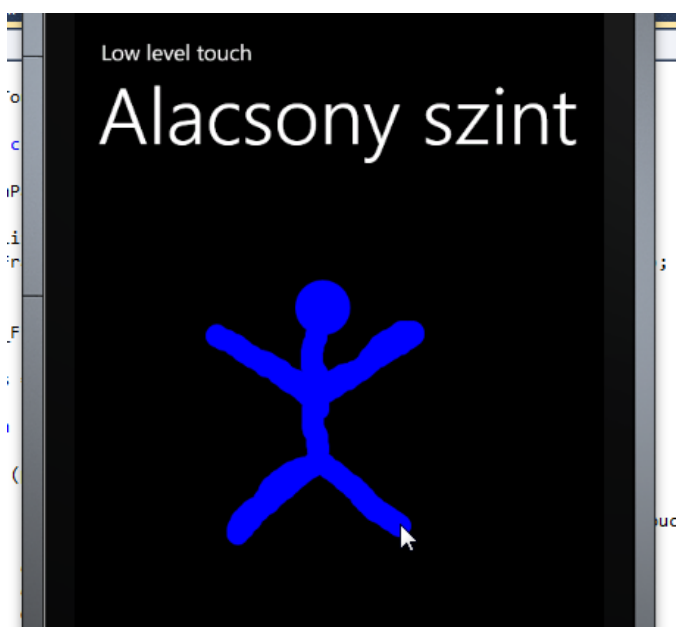
```

Az utolsó lépés, hogy amikor egy ujjbegy elereszti a kijelzőt, törölünk minden pontot, ahol korábban megfordult. Ezért tároltuk el az **Ellipse**-ek **Tag** tulajdonságában a **TouchDevice.Id**-t. Csupán annyi a dolgunk, hogy amikor egy **Action** az **Up** értéket vette fel, végigszaladjunk a **Canvas**-ban lévő **Ellipse**-eken, és amelyeknek a **Tag** értéke megegyezik az aktuális **Tag**gel, azt töröljük.

```
void Touch_FrameReported(object sender, TouchFrameEventArgs e)
{
    var pts = e.GetTouchPoints(ContentPanel);

    foreach (TouchPoint p in pts)
    {
        if (p.Action == TouchAction.Down)
        {
            Ellipse el = new Ellipse { Width = 50, Height = 50, Tag = p.TouchDevice.Id,
                Fill = new SolidColorBrush(p.TouchDevice.Id.ToColor()) };
            el.SetValue(Canvas.LeftProperty, p.Position.X - 25);
            el.SetValue(Canvas.TopProperty, p.Position.Y - 25);
            ContentPanel.Children.Add(el);
        }
        else if (p.Action == TouchAction.Move)
        {
            Ellipse el = new Ellipse { Width = 20, Height = 20, Tag = p.TouchDevice.Id,
                Fill = new SolidColorBrush(p.TouchDevice.Id.ToColor()) };
            el.SetValue(Canvas.LeftProperty, p.Position.X - 10);
            el.SetValue(Canvas.TopProperty, p.Position.Y - 10);
            ContentPanel.Children.Add(el);
        }
        else if (p.Action == TouchAction.Up)
        {
            var q = (from item in ContentPanel.Children
                where item is Ellipse && (int)((Ellipse)item).Tag == p.TouchDevice.Id
                select item as Ellipse).ToList();
            foreach (Ellipse el in q) ContentPanel.Children.Remove(el);
        }
    }
}
```

Az alkalmazás kész, tesztelhető. Ha több szint is szeretnénk látni, akkor persze mindenképpen telefonon kell tesztelnünk (7-4 ábra).



7-4 ábra – Nem Imagine Cup logó



Vegyük észre, hogy ha nagyon gyorsan rajzolunk, a körök elszakadhatnak egymástól! Ez azt mutatja, hogy gyorsabban húztuk végig ujjunkat a kijelzőn, mint ahogy a 60Hz frissítésű panel követni tudta volna. Kicsit bonyolultabb algoritmusokkal (legutolsó érintési pontok eltávolítása, Line-ok használata) ez is kiküszöbölhető.

Jó tudni: Ha a teljesítmény is szempont, érdemes elgondolkodnunk rajta, mely esemény-réteget használjuk fel. A Microsoft ajánlása ezzel – és a telefon más teljesítményigényes alkalmazásainak helyes használatával – kapcsolatban az alábbi linken érhető el: [http://msdn.microsoft.com/en-us/library/ff967560\(v=VS.92\).aspx#BKMK\\_UserInput](http://msdn.microsoft.com/en-us/library/ff967560(v=VS.92).aspx#BKMK_UserInput)

Az érintőképernyő felhasználási módjainak megismerése után hatoljunk be az eszközök belsejébe!

## Helymeghatározás

A Windows Phone-telefonok alapfelszereltségének része egy Assisted GPS, mely lehetővé teszi a Föld körül keringő GPS műholdak által a telefon pontos helyének meghatározását. A helymeghatározás azonban ennél jóval többet jelent.

Amikor a Windows Phone-on egy alkalmazás igénybe veszi a helymeghatározást, a telefon csak a legvégső esetben nyúl a GPS-hez. Hacsak nem kér a program nagy pontosságú helymeghatározást, az operációs rendszer először a cellainformációk és az esetleg elérhető, ismert wifi-pontok alapján határozza meg a helyzetet. Ez, bár tény, hogy nem méterre pontos, de általában elég – cserébe pedig gyors, és keveset fogyaszt. A GPS lassabban áll be, hiszen meg kell találnia 3-4 műholdat, illetve több energiát fogyaszt, hiszen például a cellainformáció keresése csupán az amúgy is bekapcsolt állapotban lévő mobilrendszert veszi igénybe.

A helymeghatározás használata nagyon egyszerű. A **GeoCoordinateWatcher** osztályon keresztül nyerhetjük ki a szükséges információkat a telefonból. A **PositionChanged** eseményt akkor dobja fel egy **GeoCoordinateWatcher** objektum, amikor a **MovementThreshold** tulajdonságban beállított, méterben mért távolságra kerültünk az előző eseményben rögzített pozíciótól. Az esemény csak akkor kerül kiváltásra, ha előzőleg meghívtuk a **GeoCoordinateWatcher Start** metódusát (és a **Stop**ot még nem). Fontos tulajdonság még a **DesiredAccuracy**. Ez adja meg, hogy mennyire pontos a helymeghatározás, vagyis, hogy igénybe veszi-e a GPS-t a telefon. Ez a tulajdonság csak lekérdezhető, de a **GeoCoordinateWatcher** konstruktorában be is állíthatjuk (két értéke: **Default** és **High**).

A **PositionChanged** esemény második argumentuma

**GeoPositionChangedEventArgs<GeoCoordinate>** típusú. Ennek egyetlen tulajdonsága **Position** típusú, mely egy **Timestamp**et tartalmaz (egy **DateTimeOffset**, ami megmondja, mikor történt a mérés), valamint egy **Location** tulajdonságot, mely **GeoCoordinate** típusú.

A **GeoCoordinate** egy példányából kinyerhetjük a helymeghatározás vízszintes és függőleges pontosságát (**HorizontalAccuracy** és **VerticalAccuracy** tulajdonságok), valamint természetesen a hosszúsági és szélességi értékeket (**Longitude** és **Latitude** tulajdonságok). Ezenfelül pedig információt szolgáltat arról is, hogy a tengerszinthez képest milyen magasságban van a telefon (**Altitude** tulajdonság), merre tart (**Course**) és milyen gyorsan halad ebbe az irányba (**Speed**).

Hozzunk létre egy alkalmazást a Windows Phone Application sablonnal! A **MainPage ContentPanel Grid**jét töltsük fel néhány elemmel az alábbi módon:

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
  <Grid.RowDefinitions>
    <RowDefinition Height="80" />
    <RowDefinition Height="80" />
    <RowDefinition Height="80" />
    <RowDefinition Height="80" />
  </Grid.RowDefinitions>
  <Grid.ColumnDefinitions>
    <ColumnDefinition />
    <ColumnDefinition />
  </Grid.ColumnDefinitions>
```

```
</Grid.ColumnDefinitions>
<TextBlock Name="tbkLat" />
<TextBlock Name="tbkLong" Grid.Column="1" />

<TextBlock Name="tbkHorAccu" Grid.Row="1" />
<TextBlock Name="tbkVertAccu" Grid.Row="1" Grid.Column="1" />

<TextBlock Name="tbkTimeStamp" Grid.Row="2" />
<TextBlock Name="tbkSpeed" Grid.Row="2" Grid.Column="1" />

<Button Name="btnGetCoordinates" Content="Indítás"
        Click="btnGetCoordinates_Click" Grid.Row="3" />
<CheckBox Name="cbHighAccuracy" Content="Nagy pontosság"
        Grid.Row="3" Grid.Column="1" />
</Grid>
```

Ahhoz, hogy a **Geolocation** API-t felhasználhassuk, szükségünk lesz a **System.Device** dll-re, ezt adjuk hozzá a projekt referenciáihoz, majd ezek után vegyük fel a **System.Device.Location** névteret!

```
using System.Device.Location;
```

A **MainPage**-ben hozzunk létre egy **GeoCoordinateWatcher** referenciát! Ha a XAML-tervező nem hozta létre automatikusan, akkor írjuk meg az egyelőre üres Click-kezelő metódust is!

```
public partial class MainPage : PhoneApplicationPage
{
    GeoCoordinateWatcher gcw = null;

    public MainPage()
    {
        InitializeComponent();
    }

    private void btnGetCoordinates_Click(object sender, RoutedEventArgs e)
    { }
}
```

Ezután a gomb eseménykezelőjében, ha még null értékű a **gcw** referenciánk, hozzunk létre egy **GeoCoordinateWatcher** példányt; a pontosságot attól függően állítsuk be, hogy a felhasználó bekattintotta-e az erre vonatkozó jelölőnégyzetet! Írjunk fel egy metódust a **PositionChanged** eseményre, és indítsuk el a figyelőt! Végül pedig állítsuk át a gomb szövegét!

A null értéket tesztelő elágazás **else** ágában iratkozzunk le az eseményről, állítsuk le a figyelőt, szabadítsuk fel azt, és írjuk vissza a gomb feliratát!

```
public partial class MainPage : PhoneApplicationPage
{
    GeoCoordinateWatcher gcw = null;

    public MainPage()
    {
        InitializeComponent();
    }

    private void btnGetCoordinates_Click(object sender, RoutedEventArgs e)
    {
        if (gcw == null)
        {
            gcw = new GeoCoordinateWatcher((bool)cbHighAccuracy.IsChecked ?
                GeoPositionAccuracy.High : GeoPositionAccuracy.Default);
```

```

        gcw.PositionChanged += gcw_PositionChanged;
        gcw.Start();
        btnGetCoordinates.Content = "Leállítás";
    }
    else
    {
        gcw.PositionChanged -= gcw_PositionChanged;
        gcw.Stop();
        gcw.Dispose();
        btnGetCoordinates.Content = "Indítás";
    }
}

void gcw_PositionChanged(object sender, GeoPositionChangedEventArgs<GeoCoordinate> e)
{ }
}

```

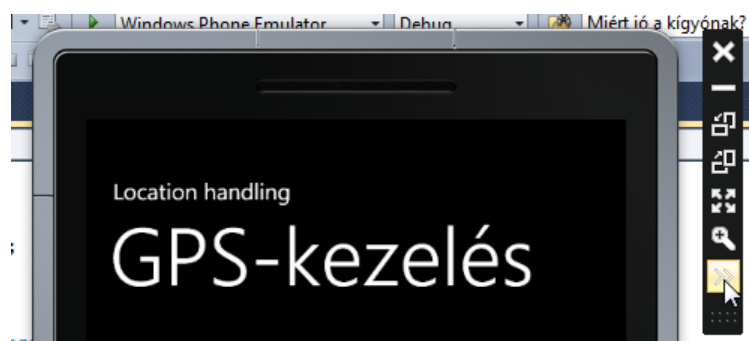
Már csak annyi dolgunk maradt, hogy amikor a **gcw** feldobja az eseményt, megfelelően kezeljük: kiírjuk a kapott adatokat a képernyőre.

```

void gcw_PositionChanged(object sender, GeoPositionChangedEventArgs<GeoCoordinate> e)
{
    tbkLat.Text = e.Position.Location.Latitude.ToString();
    tbkLong.Text = e.Position.Location.Longitude.ToString();
    tbkHorAccu.Text = e.Position.Location.HorizontalAccuracy.ToString() + " m";
    tbkVertAccu.Text = e.Position.Location.VerticalAccuracy.ToString() + " m";
    tbkTimeStamp.Text = e.Position.Timestamp.DateTime.ToLongTimeString();
    tbkSpeed.Text = e.Position.Location.Speed + " m/s";
}

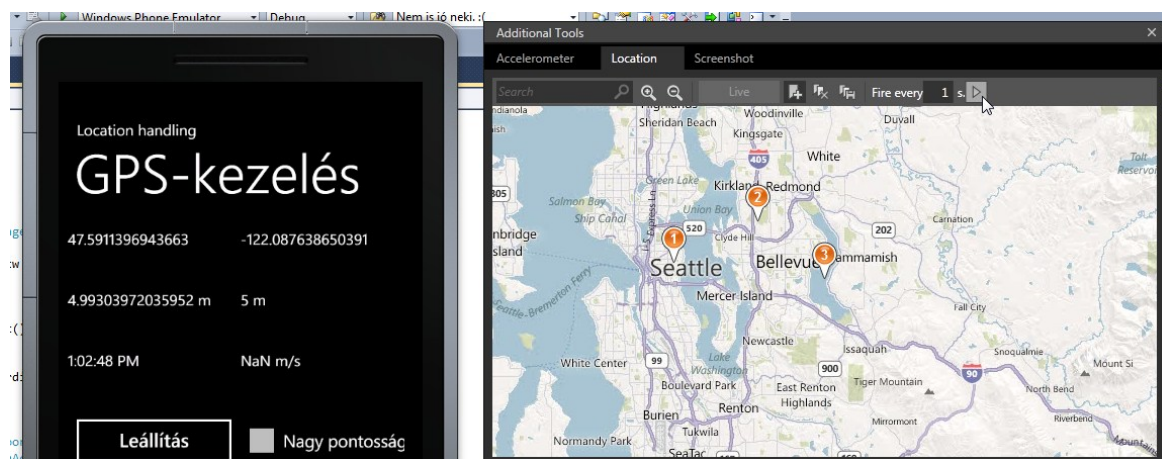
```

Az alkalmazást – ahogy minden érzékelőket használó alkalmazást – érdemes fizikai eszközön tesztelni, de ha épp nincs kedvünk vagy lehetőségünk rá, az emulátor is megfelelő lehet. Az elindított emulátor mellett látható dupla nyílra (7-5 ábra) kattintás hatására megjelenő ablakban válasszuk a **Location** fület, majd itt, a térképre kattintgatva jelöljük ki a pontokat, amit az emulátornak végig kell járnia!



7-5 ábra – Az *Additional Tools* megnyitása

Ha megvannak a pontok, kapcsoljuk ki a fent látható „Live” gombot, és kattintsunk a lejátszásra! Ha az emulátorban elindítottuk a **GeoCoordinateWatcher**-t, láthatjuk, ahogy változnak a koordináták a szimuláció közben (7-6 ábra).



7-6 ábra – A felvett útvonal lejátszása

Ne felejtsük el leállítani az érzékelőket, amikor nincs rájuk szükség! A tökéletes helyek erre az alkalmazás **Deactivated** és **Closing** eseményei, illetve a **PhoneApplicationPage**-ek **OnNavigatedFrom** eseménye. Ez nemcsak a helymeghatározó, hanem a helyzetmeghatározó érzékelőkre is vonatkozik.

## Helyzetmeghatározás

Nem a GPS az egyetlen szenzor, amit elérünk! Telefonunk helyzetének meghatározása – vagyis, hogy éppen hogyan tartja és mozgatja a felhasználó – legalább olyan fontos információkkal szolgálhat, mint a helye. A Windows Phone 7-nel szerelt eszközöknek tartalmazniuk kell többek között egy gyorsulásmérőt (accelerometer), közelségi szenzort, de ezenkívül még lehet bennük iránytű (magnetometer), illetve giroszkóp. Nem mindegyik érhető el ezek közül.

Néha szokás a szenzorok közé sorolni a kamerát és a mikrofont, jelen könyvben azonban máshol mutatjuk be ezek használatát.

## A gyorsulásmérő használata

A gyorsulásmérő megadja, hogy milyen irányban hat gyorsulás a telefonra; tehát alapesetben ez egy a Föld középpontja felé mutató vektor. Ahogy forgatjuk a telefont, a vektor X, Y és Z komponensei is változnak.

A gyorsulásmérőt az **Accelerometer** osztályon keresztül érjük el. Ennek **IsSupported** nevű statikus tulajdonsága megmondja, hogy elérhető-e a gyorsulásmérő. Ez a tulajdonság egyébként a többi érzékelő-osztályon is megtalálható; érdemes ennek ellenőrzésével kezdeni a munkát.

Egy **Accelerometer** objektum létrehozása után a **CurrentValueChanged** eseményen keresztül kapunk értesítést arról, ha megváltozott az eszköz helyzete – de itt is a **Start** és a **Stop** metódusok hívása között történik csak figyelés.

A **CurrentValueChanged** esemény második argumentuma

**SensorReadingEventArgs<AccelerometerReading>** típusú. Ennek **SensorReading** tulajdonságában egy **Timestamp** nevű tulajdonságot találunk (**DateTimeOffset**, hogy mennyi idő telt el az előző mérés óta), illetve egy **Acceleration** nevűt, amelyben az X, Y és Z komponenseket érjük el.

A szokásos Windows Phone Application sablonra építve egy egyszerű ügyességi alkalmazást hozunk létre. Egy kép „ugrál” a képernyőn, és ezt kell eltalálni egy célkeresztrel, melyet a gyorsulásmérő segítségével (vagyis a telefon forgatásával) mozgatunk.

Adjunk egy célt és egy célkeresztet ábrázoló képet a projekthez, majd a **MainPage ContentPanel**jét alakítsuk át **Canvas**sá! Helyezzünk el benne két **Image** vezérlőt, amelyek jelenítsék meg a képeket, illetve egy **TextBlock**ot, amin majd a célba talált lövéseket számoljuk! Ezenfelül pedig iratkozzunk fel a **ContentPanel Tap** eseményére:

```
<Canvas x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0" Tap="ContentPanel_Tap">
    <TextBlock Name="tbkHits" Text="0" />
    <Image Name="target" Source="kenny.png" />
    <Image Name="crosshair" Source="crosshair-red.png" />
</Canvas>
```

A célpontot azonos időközönként szeretnénk véletlenszerűen elhelyezni, tehát egy **Timer**re és egy **Random**ra lesz szükségünk – valamint természetesen egy **Accelerometer**re és egy integerre, amivel számlálunk.

```
public partial class MainPage : PhoneApplicationPage
{
    Timer timer = null;
    Accelerometer accel = null;
    Random rand = new Random();
    int hits = 0;

    public MainPage()
    {
        InitializeComponent();
    }

    private void ContentPanel_Tap(object sender, GestureEventArgs e)
    { }
}
```

Az **Accelerometer** a **Microsoft.Devices.Sensors** dll-ben található. Adjuk hozzá ezt a referenciákhoz, majd a **using**ok közé rakjuk be a hasonló nevű névteret! Ezenfelül még a **Microsoft.Xna.Framework** nevű dll-t kell felvennünk a referenciák közé.

```
using Microsoft.Devices.Sensors;
```

Iratkozzunk fel az oldal **Loaded** eseményére!

```
<phone:PhoneApplicationPage
    x:Class="AccelerometerHandling.MainPage"
    ...
    shell:SystemTray.IsVisible="True" Loaded="PhoneApplicationPage_Loaded">
```

Az eseménykezelőben inicializáljuk az objektumokat, és indítsuk el a gyorsulásmérőt!

```
public partial class MainPage : PhoneApplicationPage
{
    ...
    private void PhoneApplicationPage_Loaded(object sender, RoutedEventArgs e)
    {
        if (Accelerometer.IsSupported)
        {
            accel = new Accelerometer();
            timer = new Timer(TimerEvent, null, 5000, 5000);
            accel.CurrentValueChanged += new accel_CurrentValueChanged();
            accel.Start();
        }
        else MessageBox.Show("Nincs gyorsulásmérő. :(");
    }
    void accel_CurrentValueChanged(object sender,
        SensorReadingEventArgs<AccelerometerReading> e)
    { }
}
```

```
private void TimerEvent(object o)
{ }

private void ContentPanel_Tap(object sender, GestureEventArgs e)
{ }
}
```

A **Timer** eseményének kezelőjében helyezzük át a célképet egy véletlenszerű pozícióra! A **Random** osztály **NextDouble** metódusa egy 0 és 1 közötti értéket ad, ezt kell felszoroznunk a **ContentPanel** szélességével és magasságával, kivonva belőle a célkép szélességét és magasságát. Mivel az esemény nem a UI-szálon következik be, a **Dispatcher** objektumon keresztül tudjuk ezt megtenni. Ha közvetlenül próbáljuk, egy **InvalidOperationException** lesz a jutalmunk.

```
private void TimerEvent(object o)
{
    Dispatcher.BeginInvoke(() =>
    {
        target.SetValue(Canvas.LeftProperty, rand.NextDouble() * 430);
        target.SetValue(Canvas.TopProperty, rand.NextDouble() * 600);
    });
}
```

Ezután ellenőrizzük le, hogy amikor a felhasználó megtapintja a kijelzőt, a két kép fed-e egymást, vagyis a célkereszt a célkép felett van-e! Ha igen, növeljük meg a számláló értékét, és írjuk ki a képernyőre az aktuális értéket!

```
private void ContentPanel_Tap(object sender, GestureEventArgs e)
{
    if (Math.Abs((double)target.GetValue(Canvas.LeftProperty) -
        (double)crosshair.GetValue(Canvas.LeftProperty)) < 20 &&
        Math.Abs((double)target.GetValue(Canvas.TopProperty) -
        (double)crosshair.GetValue(Canvas.TopProperty)) < 20)
    {
        hits++;
        tbkHits.Text = hits.ToString();
    }
}
```

Már csak a lényeg maradt hátra: kezeljük le a gyorsulásmérő változását, mozgassuk a célkeresztet! Mivel a gyorsulásmérő eseménye is háttérszálon fut le, ezt a kódot is a **Dispatcher**en keresztül tudjuk csak úgy futtatni, hogy probléma nélkül hozzáférjen a UI-szálhoz. A kódban kiszámoljuk, hogy a jelenlegi **X** és **Z** tengelyen vett gyorsulással hány képponttal szükségeltetik arrébb tolni a célkeresztet, de mielőtt még ezt megtennénk, leellenőrizzük, hogy ezzel nem csúszna-e ki a célkereszt a **ContentPanel** valamelyik szélén:

```
void accel_CurrentValueChanged(object sender, SensorReadingEventArgs<AccelerometerReading> e)
{
    Dispatcher.BeginInvoke(() =>
    {
        double newLeft = (double)crosshair.GetValue(Canvas.LeftProperty) +
            e.SensorReading.Acceleration.X * 10;
        double newTop = (double)crosshair.GetValue(Canvas.TopProperty) +
            e.SensorReading.Acceleration.Z * 10;

        if (newLeft < 0) newLeft = 0;
        else if (newLeft > 450) newLeft = 450;
        if (newTop < 0) newTop = 0;
        else if (newTop > 700) newTop = 700;

        crosshair.SetValue(Canvas.LeftProperty, newLeft);
    });
}
```

```

        crosshair.SetValue(Canvas.TopProperty, newTop);
    });
}

```

Az alkalmazás kész, tesztelhető. Ha nem akarjuk vagy tudjuk fizikai eszközön tesztelni, ismét az Advanced Toolsra lesz szükségünk, hogy irányítsuk a gyorsulásmérőt.



7-7 ábra – Nem egy Korszerű Hadviselés 3, de azért élvezhető

## A giroszkóp használata

Míg a gyorsulásmérő a telefon helyzetét mérte, a giroszkóp segítségével megállapíthatjuk, hogy a felhasználó hogyan mozgatja térben a telefont. Mivel a giroszkóp nem tartozik a telefonok alapfelszereltségébe, ezért ne építsünk rá olyan programot, ami működéséhez mással nem tudja pótolni ezt az érzékelőt.

A giroszkóp használata programozási szinten alig tér el valamiben a gyorsulásmérőétől! A giroszkópot a **Gyroscope** osztály egy példányán keresztül érjük el. A **CurrentValueChanged** eseményére való feliratkozás és a **Start** metódus meghívása után egy háttérszálon kapjuk az eseményeket. A mintavételezés a **TimeBetweenUpdates** tulajdonságban megadott intervallummal történik.

A **CurrentValueChanged** esemény második argumentuma jelen esetben

**SensorReadingEventArgs<GyroscopeReading>** típusú. Ennek **SensorReading** tulajdonsága az, ami **GyroscopeReading** típusú, és tárolja a számunkra hasznos információkat. Ezek között jelen esetben is van egy **Timestamp** nevű, a korábbiakkal azonos funkciójú **DateTimeOffset**. Másik tulajdonsága pedig a **Vector3** típusú **RotationRate**. Ez a korábban már megismert X, Y és Z nevű, **float** típusú mezőket tartalmazza, melyek az egyes tengelyeken mért forgási sebességet adják meg radián per másodpercen.

Példaalkalmazásunk most is a Windows Phone Application sablonra épül, **ContentPanel**jében pedig elhelyezünk egy kört:

```

<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <Ellipse Name="ellipse" Width="100" Height="100" Fill="{StaticResource PhoneAccentBrush}"
    />
</Grid>

```



## 7. További telefonos funkciók használata

Emellett feliratkozunk az oldal betöltődését jelző eseményre:

```
<phone:PhoneApplicationPage x:Class="GyroHandling.MainPage"
...
Loaded="PhoneApplicationPage_Loaded">
```

Adjuk hozzá a projekthez a **Microsoft.Devices.Sensors** és a **Microsoft.Xna.Framework** referenciákat, és a **using**ok közé tegyük be a **Microsoft.Devices.Sensors** névteret!

```
using Microsoft.Devices.Sensors;
```

Adjunk hozzá egy **Gyroscope** referenciát az oldalhoz, és a **Loaded** eseménykezelőjében – ha van giroszkóp – hozzuk létre, iratkozzunk fel **CurrentValueChanged** eseményére, és indítsuk el!

```
public partial class MainPage : PhoneApplicationPage
{
    Gyroscope gyro = null;

    public MainPage()
    {
        InitializeComponent();
    }

    private void PhoneApplicationPage_Loaded(object sender, System.Windows.RoutedEventArgs e)
    {
        if (Gyroscope.IsSupported)
        {
            gyro = new Gyroscope();
            gyro.TimeBetweenUpdates = new TimeSpan(0, 0, 0, 0, 50);
            gyro.CurrentValueChanged += g_CurrentValueChanged;
            gyro.Start();
        }
        else MessageBox.Show("Kevés a giroszkóp.");
    }

    void g_CurrentValueChanged(object sender, SensorReadingEventArgs<GyroscopeReading> e)
    { }
}
```

Ezután már csak kezelni kell az eseményt. Amikor oldalt forgatják a telefont, oldalirányban növeljük meg a korábban létrehozott kör méretét, amikor előre-hátra döntenek, akkor pedig függőlegesen:

```
void g_CurrentValueChanged(object sender, SensorReadingEventArgs<GyroscopeReading> e)
{
    Dispatcher.BeginInvoke(() =>
    {
        ellipse.Width = 100 + 350 * e.SensorReading.RotationRate.X;
        ellipse.Height = 100 + 600 * e.SensorReading.RotationRate.Z;
    });
}
```

Az alkalmazás kész, már csak egy olyan telefont kell találni, amin tesztelni is tudjuk.

### Az iránytű használata

Az előzőektől eltérő működésű érzékelő a magnetométer, leánykori nevén iránytű. Az iránytűhöz tartozó objektum létrehozása és a kapcsolódó infrastrukturális kód a már megszokott. Az iránytűt a **Compass**



osztály jelképezi. Statikus **IsSupported** tulajdonsága megadja, hogy támogatott-e ez a szenzor az eszközön. Sajnos az emulátor ezt sem támogatja...

A példányosítás után itt is egy **CurrentValueChanged** nevű eseményre tudunk feliratkozni, illetve a **Start** és **Stop** metódusokkal elindítani, leállítani az irány figyelését. Érdekes még tudni, hogy feliratkozhatunk a **Calibrate** eseményre is, mely arról tájékoztat, hogy az operációs rendszer éppen kalibrálja az iránytűt.

A **CurrentValueChanged** esemény második argumentuma

**SensorReadingEventArgs<CompassReading>** típusú. **SensorReading** tulajdonságában ott van a jól ismert **Timestamp** tulajdonság. Emellett pedig három másik, amelyekből az irányt kaphatjuk meg. A **MagneticHeading** nevű lebegőpontos érték megadja, hogy fokokban mérve merre néz a telefon a mágneses északi sarkhoz képest. A **TrueHeading** ugyanezt teszi, csak a geográfiai északi sarkkal. Végül a **MagnetometerReading** nevű, **Vector3** típusú tulajdonság megadja, hogy a telefon X, Y és Z tengelyén mérve hány microtesla érzékelhető. Az utolsó tulajdonság pedig a **HeadingAccuracy** nevű **double** érték, mely fokokban mérve megmondja, hogy mekkora a mérés pontatlansága.

Akkor készítsük el a Magnetométert! A Windows Phone Application sablon **ContentPanel**jén jelenítsünk meg egy képet! Ennek átlátszóságát szeretnénk úgy beállítani, hogy amikor pontosan észak – pontosabban a mágneses északi sark – felé nézünk, akkor jelenjen meg teljes egészében, amikor pedig elfordulunk északtól, egyre halványabb (átlászóbb) legyen; dél felé nézve teljesen tűnjön el. Ezenkívül pedig egy **TextBlock**ban megjelenítjük a valódi északi sarkhoz viszonyított irányunkat. Az **Image** és **TextBlock** elemeket adjuk hozzá a **ContentPanel**hez, illetve iratkozzunk fel az oldal **Loaded** eseményére:

```
<phone:PhoneApplicationPage
    x:Class="MagnetometerHandling.MainPage"
    ...
    Loaded="PhoneApplicationPage_Loaded">
    <Grid x:Name="LayoutRoot" Background="Transparent">
        ...
        <Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
            <Image Name="img" Source="magneto.png" />
            <TextBlock Name="tbkOpacity" Text="0" HorizontalAlignment="Center"
                VerticalAlignment="Center" FontSize="56" />
        </Grid>
    </Grid>
</phone:PhoneApplicationPage>
```

A szokásos módon, a projekt referenciái közé vegyük fel a **Microsoft.Devices.Sensors** nevű dll-t, a **using**ok közé pedig az azonos nevű névteret!

```
using Microsoft.Devices.Sensors;
```

Adjunk hozzá egy **Compass** mezőt az oldalhoz, majd a **Loaded** eseménykezelőben állítsuk azt be! Iratkozzunk fel a **CurrentValueChanged** eseményére, és indítsuk el azt – ha támogatott az iránytű:

```
public partial class MainPage : PhoneApplicationPage
{
    Compass compass = null;

    public MainPage()
    {
        InitializeComponent();
    }

    private void PhoneApplicationPage_Loaded(object sender, System.Windows.RoutedEventArgs e)
    {
        if (Compass.IsSupported)
```

```
{
    compass = new Compass();
    compass.CurrentValueChanged += compass_CurrentValueChanged;
    compass.Start();
}
else MessageBox.Show("Keress olyan mohát, amelynek a déli oldala fás!");
}

void compass_CurrentValueChanged(object sender, SensorReadingEventArgs<CompassReading> e)
{ }
}
```

A **CurrentValueChanged** esemény a megszokott módon egy háttérzálon történik, így belsejében – szintén megszokott módon – a **Dispatcher** és némi matematika segítségével állítjuk be a kép **Opacity** és a **TextBlock Text** tulajdonságát:

```
void compass_CurrentValueChanged(object sender, SensorReadingEventArgs<CompassReading> e)
{
    Dispatcher.BeginInvoke(() =>
    {
        tbkOpacity.Text = e.SensorReading.TrueHeading.ToString();
        double maghead = 0;
        if (e.SensorReading.MagneticHeading == 0)
            maghead = 1;
        else if (e.SensorReading.MagneticHeading < 180)
            maghead = 1 - (e.SensorReading.MagneticHeading / 180);
        else
            maghead = (e.SensorReading.MagneticHeading - 180) / 180;
        img.Opacity = maghead;
    });
}
```

Kész is a program, tesztelhető – de csak telefonon, mivel az emulátoron nem támogatott az iránytű (7-8 ábra).



7-8 ábra: MÉRJÜK MEG A MÁGNESES ERŐKET

## Az érzékelők együttes használata a Motion API segítségével

Az előzőekben leírt érzékelők hasznos információkkal szolgálnak a telefon helyzetét illetően, de mindegyik csak egy-egy aspektusban tudja vizsgálni azt. Ahhoz, hogy az alkalmazás számára hasznos információkat nyerjünk ki az érzékelőktől érkező adatokból, gyakran sok matematikázásra van szükség, illetve arra, hogy több érzékelő adatait is összevegyük.

A fejlesztők munkáját megkönnyítendő a Microsoft a Windows Phone 7.5-ben beépített egy magasabb réteget is a helyzetérzékelők programozási felülete (vagyis az **Accelerometer**, a **Gyroscope** és a **Compass** osztályok) fölé. Ez a Motion API, mely a helyzetérzékelők adatait felhasználva könnyen kezelhető, releváns információkat ad. Például nem szükséges a gyorsulás komponenseiből kiszámolnunk, hogy éppen merre, mennyire fordult el a telefon, hanem a Motion API-n keresztül egyetlen tulajdonságból kiolvasható ez az adat. Emellett megkapjuk a „nyers” adatokat is, tehát egyetlen objektumon keresztül elérhetünk mindent, amire szükségünk van a helyzetmeghatározáshoz.

Ráadásul a Motion API használata teljesen megegyezik a korábban látott programozási interfészekével, tehát ha korábban már kezeltük a gyorsulásmérőt vagy társait, minden ismerős lesz.

A Motion API-t a **Motion** osztályon keresztül érjük el. Ennek statikus **IsSupported** tulajdonságából tudhatjuk meg, hogy támogatott-e a Motion API az eszközön. A **Start** és a **Stop** metódusainak meghívása között dobja fel a **CurrentValueChanged** eseményt, melynek **SensorReading** tulajdonsága a következő tulajdonságokat tartalmazza:

7-3 táblázat: A MotionReading osztály tulajdonságai

Tulajdonság	Típus	Szolgáltatott információ
<b>Attitude</b>	<b>AttitudeReading</b>	Az eszköz helyzetét adja meg.
<b>DeviceAcceleration</b>	<b>Vector3</b>	Az eszköz gyorsulása – ez a gravitáció nélkül értendő.
<b>DeviceRotationRate</b>	<b>Vector3</b>	Az eszköz három tengelye körül vett forgási együtthatók.
<b>Gravity</b>	<b>Vector3</b>	A gravitáció iránya.
<b>Timestamp</b>	<b>DateTimeOffset</b>	Az előző érzékelő-esemény óta eltelt idő.

Ahogy látható, a **Vector3** típusú tulajdonságok tulajdonképpen a gyorsulásmérő és a giroszkóp adatait adják vissza – előbbit némileg módosított formában, különválasztva a gravitáció irányát és a tényleges gyorsulást, ezzel is időt spórolva meg a fejlesztőnek. Amennyiben nincs giroszkóp a telefonban, a **DeviceRotationRate** egy nullvektort tartalmaz.

Az igazi nóvum az **Attitude**, melynek három fontos tulajdonsága a **Pitch**, a **Roll** és a **Yaw**. Ezek az egyes tengelyek körül vett elfordulások, radiánban mérve. A **Pitch** a telefon X-tengelye, a **Roll** az Y-tengely, a **Yaw** pedig a Z-tengely mentén vett elfordulás. Ezek segítségével mindenféle további számítás nélkül megkapjuk a telefon helyzetét a Földhöz képest.

Készítsünk egy alkalmazást a Visual Studio Windows Phone Application sablonjának segítségével! Iratkozzunk fel a **MainPage Loaded** eseményére, illetve a **ContentPanel**ben helyezzünk el egy képet! Ez utóbbit szeretnénk a Motion API segítségével mozgatni, pontosabban forgatni tengelyei körül úgy, ahogy a telefon forog:

```
<phone:PhoneApplicationPage
    x:Class="MotionApiHandling.MainPage"
    ...
    Loaded="PhoneApplicationPage_Loaded">
    ...
    <Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
        <Image Name="img" Source="arrow.png" Height="550"></Image>
    </Grid>
</phone:PhoneApplicationPage>
```

## 7. További telefonos funkciók használata

A telefon megdöntését (amelyet a **Yaw** tulajdonsággal állítunk be) egy egyszerű **RotateTransform** segítségével kezelhetjük. A másik két tengelyen való elforgatás ennél kicsit trükkösebb, hiszen a képet itt háromdimenziós objektumként kellene kezelnünk. A Silverlight nem támogatja a tényleges háromdimenziós forgatást. Ellenben a vizuális elemek **Projection** tulajdonsága segítségével lehetőségünk van az objektumok virtuális, térbelinek tűnő elforgatására.

Helyezzünk el egy **RotateTransform**ot és egy **PlaneProjection**ot az **Image**-ben!

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <Image Name="img" Source="arrow.png" Height="550">
        <Image.RenderTransform>
            <RotateTransform x:Name="rt" />
        </Image.RenderTransform>
        <Image.Projection>
            <PlaneProjection x:Name="pp" />
        </Image.Projection>
    </Image>
</Grid>
```

A XAML rész kész; adjunk hozzá a projekthez egy referenciát a **Microsoft.Devices.Sensors** és a **Microsoft.Xna.Framework** dll-ekre! A mögöttes kódban emeljük is be a két, ezekkel azonos nevű névteret a **using**ok közé!

```
using Microsoft.Devices.Sensors;
using Microsoft.Xna.Framework;
```

Adjunk hozzá a **MainPage**-hez egy **Motion** referenciát, és a **Loaded** eseménykezelőben, ha elérhető a Motion API, példányosítsuk az objektumot, iratkozzunk fel **CurrentValueChanged** eseményére, és indítsuk el a figyelést!

```
public partial class MainPage : PhoneApplicationPage
{
    Motion m = null;

    public MainPage()
    {
        InitializeComponent();
    }

    private void PhoneApplicationPage_Loaded(object sender, RoutedEventArgs e)
    {
        if (Motion.IsSupported)
        {
            m = new Motion() { TimeBetweenUpdates = new TimeSpan(0, 0, 0, 0, 50) };
            m.CurrentValueChanged += m_CurrentValueChanged;
            m.Start();
        }
        else MessageBox.Show("Motion API not supported. :'(");
    }

    void m_CurrentValueChanged(object sender, SensorReadingEventArgs<MotionReading> e)
    { }
}
```

A **CurrentValueChanged** esemény kezelőmetódusában pedig – a megszokott módon, a **Dispatcher** segítségével – a **PlaneProjection** és a **RotateTransform** megfelelő értékeit adjuk meg. A felhasználó felé néző Z-tengelyen vett elforgatást a **Yaw** adja meg, ezt a **RotateTransform**mal állítjuk be. A vízszintes, X-tengelyen vett elforgatás a **Pitch**, az Y-tengelyen vett elforgatás pedig a **Roll**. Ezeket rendre a **PlaneProjection** **RotationX** és **RotationY** értékéhez rendeljük hozzá.

Mivel az értékeket radiánban kapjuk, de fokban kell megadni a **PlaneProjection** és a **RotateTransform** számára, át kell alakítanunk őket. Erre a legegyszerűbb módszer, ha felhasználjuk a **Microsoft.Xna.Framework** névtérben található **MathHelper** osztályt, melynek statikus tagjai az ilyen és ehhez hasonló mindennapi transzformációk elvégzését könnyítik meg. Konkrétan a **ToDegrees** metódusra lesz szükségünk; ez egy radiánban vett értéket fogad, és fokokban véve adja vissza.

```
void m_CurrentValueChanged(object sender, SensorReadingEventArgs<MotionReading> e)
{
    Dispatcher.BeginInvoke(() =>
    {
        pp.RotationX = MathHelper.ToDegrees(e.SensorReading.Attitude.Pitch);
        pp.RotationY = MathHelper.ToDegrees(e.SensorReading.Attitude.Roll);
        rt.Angle = MathHelper.ToDegrees(e.SensorReading.Attitude.Yaw);
    });
}
```

Ha van fizikai eszközünk, amire fel tudjuk tölteni az alkalmazást, akkor indulhat a tesztelés. Az emulátor sajnos egyelőre nem támogatja a Motion API-t.

## Összefoglalás

Ebben a fejezetben megismerkedhettünk a Windows Phone 7-nel szerelt eszközökbe kötelezően vagy opcionálisan beépített érzékelők használatával. Először az érintőképernyő eseményeinek három rétegét tekintettük át a magas szintű **Tap** és társaitól a többérintéses kezelést lehetővé tevő **Manipulation** eseményeken keresztül a legalsó szinten lévő **Touch.FrameReported** eseményig, mellyel hozzáférünk a kijelző által szolgáltatott nyers adatokhoz.

Ezután áttekintettük a telefon helyének megállapítását lehetővé tevő, és az A-GPS-t a cellainformációkkal és ismert wifik alapján történő lokalizálással egyesítő **GeoLocation** API használatát a **GeoLocationWatcher** osztályon keresztül.

Végül pedig számba vettük, hogy milyen egyéb, helyzetmeghatározó érzékelők és programozási felületek állnak rendelkezésünkre. A telefonra ható erőket – így a gravitációt is – mérő gyorsulásmérő, valamint a mágneses erők irányát megadó magnetométer minden telefon alapfelszereltsége, a telefon forgását mérő giroszkóp már opcionálisan beépíthető komponens. Ezeknek az eszközöknek a kezelése nagyon hasonló. A Windows Phone 7.5-ben megjelent Motion API révén pedig mindenezeket a nyers adatokat akkumulálva, releváns információk formájában nyerhetjük ki az operációs rendszerből.



# 8. Adatkezelés

Minden nagyobb alkalmazásnál eljön az az idő, amikor tárolni kell adatokat a készüléken, legyen szó egyszerű beállításokról vagy összetett adatokról, amit relációs adatbázisban tárolunk. Ebben a fejezetben megismerkedhetünk azzal, hogyan kezeljük adatokat a Windows Phone 7 készülékünkön. A fejezetben kitérünk a következő témakörökre:

- IsolatedStorage megismerése
- ApplicationSettings használata
- Hogyan olvassunk és írjunk fájlokat
- IsolatedStorage Tool-ok megismerése
- Lokális adatbázisok használata

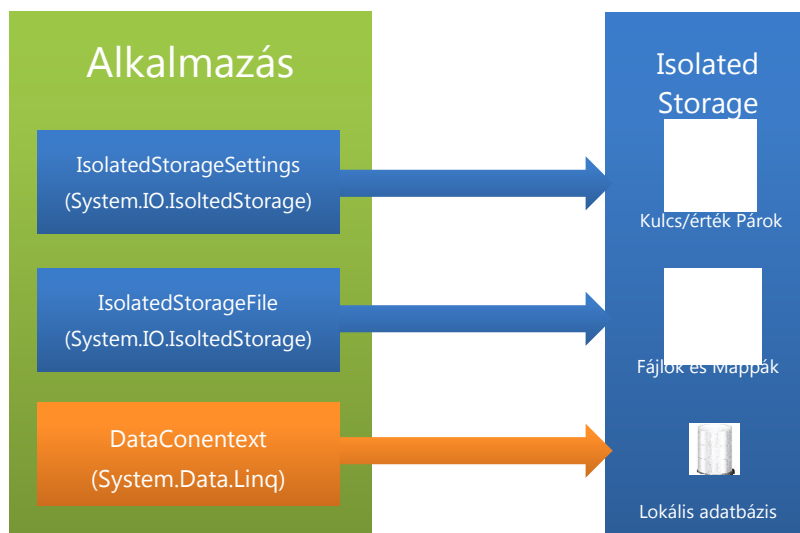
## Isolated Storage

A Windows Phone 7-nél nem beszélhetünk klasszikus értelemben a fájlkezelésről: ha I/O műveleteket szeretnénk elvégezni a fájlrendszeren, akkor csak az Isolated Storage áll a rendelkezésünkre. Ahogy a neve is mutatja, ez egy tárhely (Storage), ahol az alkalmazásunk képes írni, olvasni, valamint az általános fájlműveleteket is el tudja végezni. Minden alkalmazás kérhet magának egy-egy ilyen elkülönített tárhelyet, vagyis az alkalmazások nem férhetnek hozzá egymás tárhelyéhez. Minden alkalmazás csak a saját Isolated Storage-ében tud garázdálkodni, ott viszont úgy, ahogy csak szeretné. Közvetlen módon tehát nem érhetjük el a teljes fájlrendszert, csak és kizárólag az alkalmazásunk által használt Isolated Storage-ünket, így más alkalmazások semmilyen módon nem befolyásolhatják az alkalmazásunk működését, ezzel növelve a biztonságot és a fájlkárosodás elkerülését. A Windows Phone 7 Isolated Storage megoldása nagyon hasonló a Silverlightos Isolated Storage megoldásához, mindössze annyi a különbség, hogy itt nincs megszabva kvóta, mint a Silverlightnál, vagyis itt egy alkalmazás korlátlanul gazdálkodhat a tárhellyel. Bár nincs méretkorlát az alkalmazások készítői számára, néhány szempontot célszerű szem előtt tartani:

- Csak akkor használjuk a fájlrendszert, ha tényleg szükséges!
- A fájlkezelés legyen transzparens a felhasználók számára!
- Ha átmeneti (Temp) fájlokat készítünk, akkor lehetőleg azonnal töröljük azokat, amint már nem használjuk, és nincs is többé szükség rájuk!
- Bizonyos esetekben célszerű ezeket a fájlokat szinkronizálni vagy archiválni a felhőbe (Cloud), így ha az alkalmazás törlésre kerül, a felhasználó adatai még megmaradnak, illetve ha a felhasználó másik kliensen jelentkezik be, akkor a megszokott környezetével találkozhat.

A Windows Phone SDK 7.1 verziója óta háromféle adatkezelési módszer létezik a készüléken (8-1 ábra):

- **Settings:** Kulcs/érték adatokat tárolhatunk benne (**IsolatedStorageSettings**). Leginkább az alkalmazásunk beállításainak mentésére használjuk.
- **Files and Folders:** Fájlokat és mappákat kezelhetünk a segítségével (**IsolatedStorageFile**).
- **Relation Data:** Relációs adatokat tárolhatunk egy lokális adatbázisban (SQL Server Compact Edition), amiket a Linq To SQL technológiával érhetünk el.



**8-1 ábra: A Windows Phone 7 adatkezelési lehetőségei**

A Windows Phone első, 7.0 változatában csak a *Settings* és a *Files and Folders* adatkezelési módszer állt a rendelkezésünkre, de a Mango frissítéstől kezdve arra is van lehetőségünk, hogy egy lokális adatbázisban relációs módon tároljunk adatokat. A következő fejezetekben megismerkedünk az összes módszerrel, amellyel adatokat kezelhetünk a Windows Phone készüléken.

**Kérdés:** Ha lenne két olyan WP7-es alkalmazás, ami ugyanazt a fájl alapú adatforrást használja, meg tudnám osztani valamilyen módon az **IsolatedStorage**-ot?

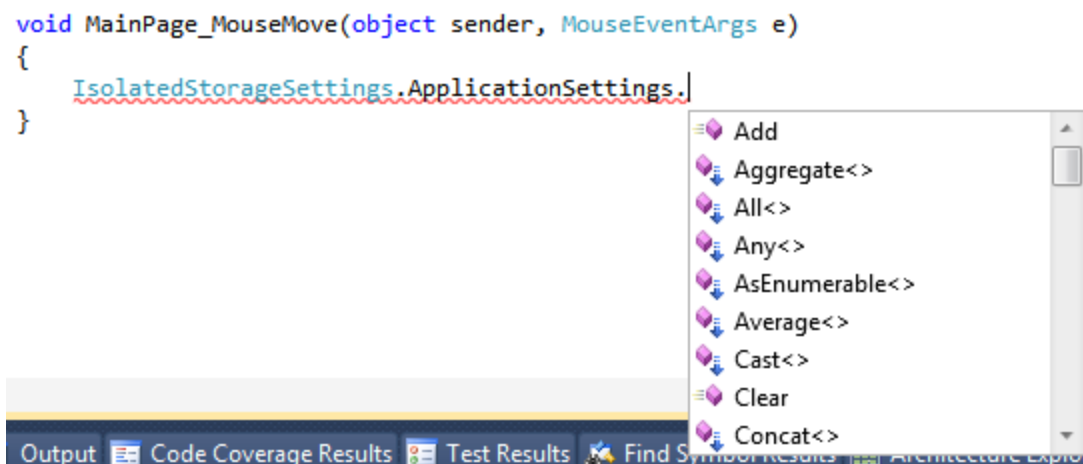
**Válasz:** Nem. Ebben az esetben például egy webszolgáltatáson keresztül oldhatjuk meg a fájlok közös használatát.

## Application Settings

A fájlrendszert kétféleképpen érhetjük el: az első, egyszerűbb eset egy kulcs/érték pár tárolási módszer, a második pedig a tradicionális fájlkezelés, ahol olvashatjuk és írhatjuk a fájl stream-et. Ebben a részben az első módszert, az *Application Settings*-et fogjuk megismerni.

Amikor csak egyszerű alkalmazás-beállításokat szeretnénk elmenteni vagy lekérdezni, akkor az **ApplicationSettings** kulcs/érték pár tárolási módszer a legcélravezetőbb megoldás. Például: ha egy felhasználónak egy szolgáltatás bejelentkezéséhez egy szövegdobozba kell beírnia a felhasználói nevét, akkor a mobilon a munkát egyszerűbbé tehetjük számára azzal, hogy ha már egyszer beírta, a háttérben eltároljuk ezt a felhasználónevet. Ha később újra elindítja az alkalmazást, akkor a korábban megadott és eltárolt felhasználónévvel automatikusan ki lesz töltve az adott szövegdoboz. Az **ApplicationSettings** tulajdonképpen egy szótár, amely megvalósítja az **IEnumerable<T>** interfészt, tehát minden olyan műveletet elvégezhetünk rajta, amit egy normál gyűjteményen is, amint azt a 8-2 ábra is illusztrálja.





8-2 ábra: Bővítő metódusok egész sora áll a rendelkezésünkre

Az **ApplicationSettings** egy speciális **Dictionary<TKey, TValue>** objektum, amely egy kulcs/érték párost tud tárolni, és ezeket az értékpárokat az adott alkalmazás **IsolatedStorage**-ébe menti el. Az alábbi kódrészlet az **IsolatedStorageSettings** felépítését mutatja be:

```
namespace System.IO.IsolatedStorage
{
    public sealed class IsolatedStorageSettings : IDictionary<string, object>,
        ICollection<KeyValuePair<string, object>>, IEnumerable<KeyValuePair<string, object>>,
        IDictionary, ICollection, IEnumerable
    {
        public static IsolatedStorageSettings ApplicationSettings { get; }
        public int Count { get; }
        public ICollection Keys { get; }
        public ICollection Values { get; }

        public object this[string key] { get; set; }

        public void Add(string key, object value);
        public void Clear();
        public bool Contains(string key);
        public bool Remove(string key);
        public void Save();
        public bool TryGetValue<T>(string key, out T value);
    }
}
```

Ahhoz, hogy az **IsolatedStorage**-et elérjük, a **System.IO.IsolatedStorage** névtérre, a fájlművelethez pedig a **System.IO** névtérre lesz szükségünk.

Egy új kulcs/érték párt többféle módon hozhatunk létre: egyszerűen az **Add** metódus meghívásával, ebben az esetben az első paraméter egy kulcs, aminek **egyedinek** kell lennie. Ha már létezik a kulcs, akkor egy **System.ArgumentException**-t kaphatunk, a második paraméter pedig egy tetszőleges objektum lehet. Az alábbi példában egy egyszerű stringet láthatunk:

```
// Felhasználónév hozzáadása - Új kulcs létrehozása
IsolatedStorageSettings.ApplicationSettings.Add("UserName", "Attila");
```

Mivel ez egy **Dictionary**, ezért a rá vonatkozó szabályok élnek, így például egy indexer segítségével is hozzáadhatunk egy új kulcs/érték-párt (így tudjuk a már létező kulcsok értékét is módosítani). Az

indexeres megoldásnál, ha az adott kulcs nem létezik, akkor az létrejön, abban az esetben, ha már létezik a kulcs, akkor a tartalmát felülírja a meghatározott értékkel.

```
// Felhasználónév mentése /vagy módosítása
IsolatedStorageSettings.ApplicationSettings["UserName"] = "Attila";
```

Az adott kulcsok értékét persze le is kérdezhettük, ehhez is használhatunk két módszert: az egyik az itt bemutatott indexeres módszer:

```
//Felhasználóné lekérdezése
var userName = IsolatedStorageSettings.ApplicationSettings["UserName"];
```

A másik lehetőség a **TryGetValue** metódus használata:

```
string userValue;
IsolatedStorageSettings.ApplicationSettings.TryGetValue<String>("Kulcs", out userValue);
```

Az Indexeres módszer esetén, ha a kulcs nem található meg a szótárban, akkor egy **KeyNotFoundException**-t fogunk kapni, míg ha a **TryGetValue** metódust használjuk, nem kapunk hibát. Ebben az esetben, ha a kulcs nem létezik, akkor a **userValue** változó értéke üres (null) lesz. A **TryGetValue** nagyon hasonlít a **TryParse** metódusokra, amit a .NET-es típusoknál használhatunk. Ez is egy **Boolean** értékkel tér vissza: ha sikerült a kulcsot megtalálni és kiolvasni a tartalmát, akkor **true**-val, egyébként **false**-szal.

Abban az esetben, ha meg akarunk győződni arról, valóban létezik-e a kulcs, a **Contains** metódust használhatjuk fel. Ez a metódus **true** értékkel tér vissza, ha létezik az adott kulcs, **false** értékkel, ha nem.

```
if (IsolatedStorageSettings.ApplicationSettings.Contains("UserName"))
{
    //Létezik a UserName kulcs. Olvassuk ki pl. a tartalmát és írassuk ki
    MessageBox.Show(IsolatedStorageSettings.ApplicationSettings["UserNme"].ToString());
}
else
{
    //Nem létezik
    MessageBox.Show("Nincs ilyen kulcs!");
}
```

Az előző példákban csak egy egyszerű string adatot tároltunk el, de természetesen komplex objektumokat is eltárolhatunk az **ApplicationSettings**-ben. A következő példában egy saját osztályt hozunk létre, és azt tároljuk el, majd olvassuk ki. Ebben az esetben nem kell törődnünk az adatsorosítással, ezt az **ApplicationSettings** saját maga megoldja.

```
public class Person
{
    public string Name { get; set; }
    public int Age { get; set; }
}
...

Person myPerson = new Person()
{
    Name = "Jhon",
    Age = 32
};
//Adat mentése
IsolatedStorageSettings.ApplicationSettings["PersonData"] = myPerson;
```

```
// Adat lekérdezése
Person newPerson = IsolatedStorageSettings.ApplicationSettings["PersonData"] as Person;
MessageBox.Show(newPerson.Name);
```

Az **ApplicationSettings** az alkalmazás bezárásakor elmenti az adatokat, viszont ha az alkalmazás futása közben nem várt hiba történik, és az alkalmazás leáll, ezek az információk nem kerülnek elmentésre. Ha biztosra szeretnénk menni, akkor ki kell kényszerítenünk a mentést az **ApplicationSettings Save** metódusával.

```
IsolatedStorageSettings.ApplicationSettings.Save();
```

A mentett adatokat persze el is távolíthatjuk az **ApplicationSettings**-ből. Eltávolíthatunk egy kulcsot, ilyenkor a **Remove** metódust kell használnunk:

```
IsolatedStorageSettings.ApplicationSettings.Remove("UserName");
```

Ha az összes értéket el akarjuk távolítani, akkor a **Clear** metódust kell meghívunk:

```
IsolatedStorageSettings.ApplicationSettings.Clear();
```

Mivel az **ApplicationSettings** egy szótár, ezért mind a kulcsokon, mind az értékeken végig tudunk haladni a **foreach** utasítással:

```
foreach (var key in IsolatedStorageSettings.ApplicationSettings.Keys)
{
    // Feldolgozás
}

foreach (var value in IsolatedStorageSettings.ApplicationSettings.Values)
{
    // Feldolgozás
}
```

Egy alkalmazásnál sokszor használhatjuk az **ApplicationSettings**-et, melynek folyamatos kiírása fárasztó lehet, ezért ezt egyszerűbben az alábbi módon is megoldhatjuk – így több billentyű leütését is megspórolhatjuk:

```
var settings = IsolatedStorageSettings.ApplicationSettings;
settings["Kulcs"] = "Érték";
string myValue = settings["Kulcs"].ToString();
```

Az előző rövid példákban látható volt, hogy az **IsolatedStorage** használata mennyire egyszerű. Egy dolgot azonban még tisztáznunk kell, ez pedig az adatkötés technikája! Az előzőekben csak explicit módon mentettünk el és olvastunk ki adatokat, viszont sokszor fordul elő, hogy jó volna ezt centralizálni, és az **ApplicationSettings** elérését egyszerűsíteni és ésszerűsíteni. A következő példában egy **Settings** osztályt hozunk létre, amiben általános és/vagy ellenőrző módon tudjuk elérni az általunk használt kulcs/érték párokat. Ez az osztály egy általánosan használható **RetrieveSettings** segédmetódust tartalmaz, amely a paraméterében átadott kulcs értékét adja vissza. Ha az adott kulcs nem található meg a szótárban, akkor az adott típus alapértelmezett értékével (ebben az esetben ez null) tér vissza.

```
public class Settings
{
    private const string ApplicationTitleKey = "ApplicationTitle";
    private const string PersonKey = "Person";
```

```
private T RetriveSettings<T>(string settingKey)
{
    object settingValue;
    if (IsolatedStorageSettings.ApplicationSettings.TryGetValue(settingKey, out
settingValue))
    {
        return (T)settingValue;
    }
    return default(T);
}

public string ApplicationTitle
{
    get
    {
        return RetriveSettings<string>(ApplicationTitleKey);
    }
    set
    {
        IsolatedStorageSettings.ApplicationSettings[ApplicationTitleKey] = value;
    }
}

public Person MyPerson
{
    get
    {
        return RetriveSettings<Person>(PersonKey);
    }
    set
    {
        IsolatedStorageSettings.ApplicationSettings[PersonKey] = value;
    }
}
}
```

Az osztályban minden kulcsnak különálló tulajdonságot készítünk, így könnyebb az alkalmazásunkban ezeket a beállításokat írni és olvasni, ráadásul elkerüljük a szöveges értékű kulcsok elírását is. Mostantól tehát minden kulcsnak külön tulajdonságot fogunk készíteni, így a **Settings** osztályon keresztül az összes kulcsértéket el tudjuk érni.

Felmerülhet bennünk a kérdés, hogy a **Settings** osztály miért nem lett static? Így sokkal egyszerűbb lenne elérni a tulajdonságok értékét, mint mindig példányosítani egy **Settings**-et! A válasz egyszerű: célravezetőbb ezt az osztályt az **App.xaml** fájlban deklaratív módon példányosítani, így csak az alkalmazás indulásakor egyszer készítünk egy **Settings** példányt, és erre ezt követően az alkalmazásunkban bárhol hivatkozhatunk kódból és XAML-ből egyaránt:

```
...
xmlns:local="clr-namespace:PhoneAppTest"
>

<Application.Resources>
    <local:Settings x:Key="MySettings" />
</Application.Resources>
```

Miután az App.xaml-ben elkészítettük a példányt, nyissuk meg az egyik oldalt (jelen esetben a MainPage.xaml), és a PhoneApplicationPage DataContext-jének adjuk át a MySettings-et az alábbi módon!

```
<phone:PhoneApplicationPage
  x:Class="PhoneAppTest.MainPage"
  ...
  SupportedOrientations="Portrait" Orientation="Portrait"
  shell:SystemTray.IsVisible="True" DataContext="{StaticResource MySettings}">
```

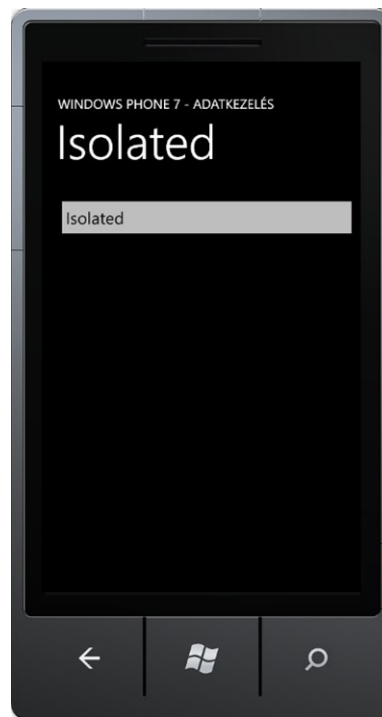
Ezt követően az oldalra helyezzünk el egy **TextBox**-ot, és a **Text** tulajdonságát kössük az **ApplicationTitle**-höz. Ebben az esetben a **Binding** módja **TwoWay**, azaz a kötés kétirányú.

```
<TextBox Text="{Binding ApplicationTitle, Mode=TwoWay}" Height="72" HorizontalAlignment="Left"
  Margin="6,6,0,0" Name="textBox1" VerticalAlignment="Top" Width="460" />
```

A **PageTitle** **Textblock** **Text** tulajdonságához is kössük az **ApplicationTitle**-t:

```
<TextBlock x:Name="PageTitle" Text="{Binding ApplicationTitle}" Margin="9,-7,0,0"
  Style="{StaticResource PhoneTextTitle1Style}"/>
```

Ha elindítjuk az alkalmazást (6-3 ábra), és a szövegdobozba beírunk valamit, az a háttérben tárolásra fog kerülni. Írjunk oda egy tetszőleges szöveget, majd kattintsunk a Start gombra az emulátoron, ekkor az alkalmazás háttérbe vonul! Az emulátoron újra indítsuk el az alkalmazást, és láthatjuk, hogy az oldal címe és a szövegdoboz tartalma az **ApplicationSettings**-ből kiolvasott érték lesz, amint azt a 8-3 ábrán láthatjuk.



**8-3 ábra: Application Settings demó alkalmazás**

**Gyakorló feladat:** A **Settings** osztályban van egy **MyPerson** tulajdonság is. Ehhez készítsük el a felhasználói felületet, és mentjük el az értékét!

## IsolatedStorageFileStream

Bár könnyű kulcs értékeket tárolni, olykor ez kevés lehet a számunkra, és hagyományos fájlok elérésére volna szükségünk a fájlrendszeren. Ebben a részben a tradicionális, *stream*-alapú fájlkezelést fogjuk közelebbről megismerni. Ahhoz, hogy egy új fájlt létrehozzunk, az **IsolatedStorageFileStream** osztályra lesz szükségünk. Ennek az osztálynak a példányosításakor kell meghatároznunk a fájl nevét, és annak elérési módját (**FileMode** enumeráció), valamint át kell adnunk egy **IsolatedStorage** példányt is. Ebben az egyszerű példában egy fájlt hozunk létre, mely a „Hello World” szöveget tartalmazza:

```
IsolatedStorageFile isf = IsolatedStorageFile.GetUserStoreForApplication();

using (IsolatedStorageFileStream myFileStream = new IsolatedStorageFileStream("data.txt",
    FileMode.Create, isf))
{
    StreamWriter sw = new StreamWriter(myFileStream);
    sw.WriteLine("Hello World");
    sw.Flush();
    sw.Close();
}
```

A **GetUserStoreForApplication** metódus egy **IsolatedStorageFile** példányt ad vissza. A fájlba egy **StreamWriter** példány segítségével írhatunk. Ennek át kell adnunk egy **Stream**-et, ez ebben az esetben egy **IsolatedStorageFileStream**, majd a **StreamWriter** példány **WriteLine** metódusával írhatunk egy sort a fájlba.

A fájlból olvasás hasonlóan egyszerű, annyi a különbség, hogy a **FileMode** ebben az esetben az **Open** értékét viseli, valamint egy **StreamReader** példányt használunk az olvasáshoz.

```
IsolatedStorageFile isf = IsolatedStorageFile.GetUserStoreForApplication();

using (IsolatedStorageFileStream myFileStream = new IsolatedStorageFileStream("data.txt",
    FileMode.Open, isf))
{
    StreamReader sr = new StreamReader(myFileStream);
    string text = sr.ReadToEnd();
    MessageBox.Show(text);
}
```

Az alábbi táblázatban az **IsolatedStorageFile** tulajdonságait foglaljuk össze:

Tulajdonság	Leírás
<b>AvailableFreeSpace</b>	Az <b>IsolatedStorage</b> -ben rendelkezésre álló szabad terület mérete.
<b>IsEnabled</b>	Lekérdezhetjük, hogy az <b>IsolatedStorage</b> engedélyezve van-e.
<b>Quota</b>	A maximálisan felhasználható szabad terület mérete.
<b>UsedSize</b>	A már foglalt terület nagysága.

Az alábbi táblázatban az **IsolatedStorageFile** fontosabb metódusait találjuk:

Metódus	Leírás
<b>CopyFile(String, String)</b>	Egy létező fájl másolása.
<b>CopyFile(String, String, Boolean)</b>	Egy létező fájl másolása. Ha a cél útvonalon létezik a fájl, akkor opcionálisan felülírhatjuk.
<b>CreateDirectory</b>	Mappa készítése

Metódus	Leírás
CreateFile	Új fájl készítése
DeleteDirectory	Mappa törlése
DeleteFile	Fájl törlése
DirectoryExist	Megvizsgálhatjuk, hogy az adott mappa létezik-e.
FileExist	Megvizsgálhatjuk, hogy az adott fájl létezik-e.
GetCreationTime	Visszaadja a fájl vagy mappa készítési idejét.
GetDirectoryNames()	Visszaadja a mappák neveit (Gyökérkönyvtárból kiindulva).
GetDirectoryNames(String)	Visszaadja a mappák neveit egy meghatározott minta alapján.
GetFileNames()	Visszaadja a fájlok neveit (Gyökérkönyvtárból kiindulva).
GetFileNames(String)	Visszaadja a fájlok neveit egy meghatározott minta alapján.
GetLastAccessTime	Visszaadja, hogy a fájlhoz vagy mappához mikor fértek hozzá utoljára.
GetUserStoreForApplication	A felhasználóhoz tartozó izolált tárhelyet kezelő <b>IsolatedStorageFile</b> példányt kérhetünk a segítségével.
MoveDirectory	Mappa (és tartalmának) áthelyezése egy új útvonalra
MoveFile	Fájl mozgatása egy új helyre, opcionálisan a fájl nevét is megváltoztathatjuk.
OpenFile(String, FileMode)	Fájl megnyitása egy meghatározott módon
OpenFile(String, FileMode, FileAccess)	Fájl megnyitása egy meghatározott módon- és hozzáféréssel.
Remove	Az <b>IsolatedStorage</b> és tartalmának eltávolítása

Nézzünk néhány gyakori felhasználási esetet!

Könyvtár létrehozáshoz a **CreateDirectory** metódust használhatjuk fel, ahol a metódusnak a könyvtár nevét adjuk át:

```
IsolatedStorageFile myISF = IsolatedStorageFile.GetUserStoreForApplication();
myISF.CreateDirectory("UjKonvtar");
```

Megadhatunk összetett könyvtár útvonalat is:

```
myISF.CreateDirectory("Folder1/Folder2/Folder3/UjKonvtar");
```

Egy könyvtár törlése is legalább ilyen egyszerű, ehhez a **DeleteDirectory** metódust használhatjuk fel:

```
myISF.DeleteDirectory("UjKonvtar");
```

Mind a könyvtár létrehozásakor, mind a könyvtár törlésekor célszerű ellenőrizni, hogy az adott könyvtár létezik-e:

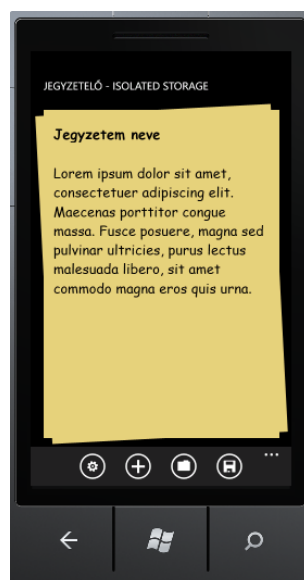
```
public void CreateDirectory(string directoryName_in) {
    try {
        IsolatedStorageFile myIsolatedStorage =
        IsolatedStorageFile.GetUserStoreForApplication();
        if(!string.IsNullOrEmpty(directoryName_in) &&
        !myIsolatedStorage.DirectoryExists(directoryName_in))
        {
            myIsolatedStorage.CreateDirectory(directoryName_in);
        }
    }
    catch (Exception ex)
    {
        // Kivétel
    }
}

public void DeleteDirectory(string directoryName_in){
    try {
        IsolatedStorageFile myIsolatedStorage =
        IsolatedStorageFile.GetUserStoreForApplication();
        if (!string.IsNullOrEmpty(directoryName_in) &&
        myIsolatedStorage.DirectoryExists(directoryName_in)) {
            myIsolatedStorage.DeleteDirectory(directoryName_in);
        }
    }
    catch (Exception ex) {
        // Kivétel
    }
}
```

Láthatjuk, hogy a Windows Phone 7 alatt a fájlkezelés rendkívül egyszerű és gyorsan elsajátítható. Ráadásul, akik korábban Silverlight alatt használták ezt a funkciót, azok számára ez sok újdonságot nem is rejteget. Gyakorlottaknak és kezdőknek egyaránt könnyen és hatékonyan használható API áll a rendelkezésére.

### ***Isolated Storage gyakorlat***

A következő gyakorlatban egyszerű jegyzetelő alkalmazást fogunk elkészíteni (8-4 ábra). Az alkalmazás felhasználói felülete előre el van készítve. A projekt kiindulási anyagát a következő oldalról tölthetjük le: <http://devportal.hu/wp7>.



**8-4 Az elkészített jegyzetelő alkalmazás**



A gyakorlatban csak az adattárolással kapcsolatos kódokat fogjuk közösen megírni, az alábbi lépésekben:

- Nyissuk meg az **IsolatedStorageDemo.sln** fájlt (**Begin** mappa)! Ha valamit a gyakorlat lépései során nem sikerülne pontosan követnünk, az **End** mappában megtalálható és kipróbálható az alkalmazás végleges formája.
- Fordítsuk le, és indítsuk el az alkalmazást! Egyszerű jegyzetelő felületet kapunk, de jelenleg még nem tudjuk sem elmenteni, sem betölteni jegyzeteinket.
- Itt az ideje ezeket a funkciókat megvalósítani! Zárjuk be a futó alkalmazást, és térjünk vissza a projekthez! Nyissuk meg az **App.xaml.cs** fájlt, és navigáljunk el az **Application\_Startup** eseményhez, itt fogjuk a jegyzeteink számára elkészíteni a „Notes” mappát. Az alkalmazás minden indulásakor megvizsgáljuk a DirectoryExist metódussal, hogy a Notes mappa létezik-e. Ha nem létezik (pl. első indulás), akkor a CreateDirectory metódussal létrehozzuk:

```
IsolatedStorageFile isf = IsolatedStorageFile.GetUserStoreForApplication();
if (!isf.DirectoryExists("Notes"))
{
    isf.CreateDirectory("Notes");
}
```

- Nyissuk meg a **MainPage.xaml.cs** fájlt, és oldjuk fel a **System.IO** és a **System.IO.IsolatedStorage** névteret! (A feladatban a többi lapjához már hozzáadtuk ezeket a névtereket.)

```
using System.IO;
using System.IO.IsolatedStorage;
```

- A **MainPage.xaml.cs**-ben navigáljunk el a **btnSave\_Click** eseményhez, ugyanis itt fogjuk elmenteni a jegyzetünket a fájlrendszerbe. Először a path változóban létrehozunk egy útvonalat a jegyzeteink számára: azok a Notes mappába fognak kerülni, és **.dat** lesz a kiterjesztésük. Ezt követően létrehozunk egy **IsolatedStorageFileStream**-et. Ezt legegyszerűbben a **StreamWriter**-rel tudjuk írni, így a **mySw** példány **WriteLine** metódus paraméterének átadjuk a **txtDocument.Text** tulajdonságának értékét. Ha minden sikeresen lezajlott, akkor egy **MessageBox**-ban ezt megüzenjük a felhasználónak.

```
string path = "Notes\\" + txtTitle.Text + ".dat";
IsolatedStorageFile myISFile = IsolatedStorageFile.GetUserStoreForApplication();
using (IsolatedStorageFileStream myFs = new IsolatedStorageFileStream(path, FileMode.Create,
    FileAccess.Write, myISFile))
{
    using (StreamWriter mySw = new StreamWriter(myFs))
    {
        mySw.WriteLine(txtDocument.Text);
        mySw.Flush();
    }
}
MessageBox.Show("A jegyzetedet elmentettük!", "Isolated Storage Demo", MessageBoxButton.OK);
```

- Most már elkészítettük az alkalmazásunk jegyzet (Notes) mappáját, és el is tudjuk menteni a jegyzeteinket. Itt az ideje megírni azt is, hogy a korábban elmentett jegyzeteinket vissza tudjuk tölteni. Nyissuk meg a **Pages/SavedItems.xaml.cs** fájlt, és navigáljunk el a **SavedItems** konstruktorába: itt fogjuk kiolvasni a fájlrendszerről, hogy jelenleg milyen jegyzeteink vannak mentve. Ehhez az **IsolatedStorageFile** példány **GetFileNames** metódusát fogjuk segítségül hívni az alábbi módon:

```
IsolatedStorageFile isf = IsolatedStorageFile.GetUserStoreForApplication();
lstMyNotes.ItemsSource = isf.GetFileNames("Notes\\*.dat");
```

A **GetFileNames** visszatérési értéke egy **String** tömb, ezt adjuk át a **lstMyNotes** listának.

- g. Az előző pontnál megjelenítettük a jegyzeteink fájljait egy listában, most pedig megírjuk azt az egyszerű funkciót, amivel a kiválasztott fájlt átadjuk a **MainPage**-nek, ami majd megnyitja az adott jegyzetet. Navigáljunk el a **lstMyNotes\_SelectionChanged** eseményhez, és a metódus törzsébe írjuk a következőt:

```
string selectedItem = lstMyNotes.SelectedItem as string;
if (selectedItem != null)
{
    NavigationService.Navigate(new Uri("/MainPage.xaml?OpenFile="+selectedItem,
    UriKind.Relative));
}
```

Itt egyszerűen a kiválasztott fájl nevét átadjuk a **MainPage**-nek, még hozzá a **QueryString** átadási mintát követve.

- h. Természetesen, ha átadtuk ezt a paramétert, akkor azt le is kell kérdeznünk, és az átadott értéknek megfelelően megnyitni az adott fájlt. Térjünk vissza a **MainPage.xaml.cs** –hez, és navigáljunk el az **OnNavigatedTo** eseményhez, és itt létrehozunk egy **fileName** nev string változót. A **NavigationContext.QueryString.TryGetValue** metódusával megpróbáljuk elérni az **OpenFile** paraméter értékét. A fájl olvasása a fájlírás analógiájára történik, itt azonban az **IsolatedStorageFileStream FileMode** felsorolt típusából az **Open** értéket választjuk, és **StreamWriter** helyett **StreamReader**-t használunk. A **StreamReader ReadToEnd** metódusának segítségével a fájl teljes tartalmát kiolvassuk, ezt követően átadjuk a **txtDocument**-nek. A **txtTitle** esetén a **fileName** szövegből eltüntetjük a **.dat** kiterjesztést, hogy elegánsabban jelenjen meg a fejléc:

```
string fileName = string.Empty;
if (NavigationContext.QueryString.TryGetValue("OpenFile", out fileName))
{
    IsolatedStorageFile isf = IsolatedStorageFile.GetUserStoreForApplication();
    using (IsolatedStorageFileStream myFs = new IsolatedStorageFileStream("//Notes//" +
    fileName, FileMode.Open, isf))
    {
        using (StreamReader sr = new StreamReader(myFs))
        {
            txtTitle.Text = fileName.Replace(".dat", "");
            txtDocument.Text = sr.ReadToEnd();
        }
    }
}
```

- i. Ezzel elkészült az alkalmazásunk. Fordítsuk le a forráskódot (Ctrl+Shift+B), és ha valamilyen hibát tapasztalunk, nézzük át újra a kódot és javítsuk azt! Ha minden rendben lezajlott, akkor indítsuk el az alkalmazásunkat (F5)! Az megjelenik kezdőképernyőjén a jegyzeteink listájával, amint azt a 8-5 ábra mutatja.



**8-5 ábra: Jegyzetek megnyitása**

- j. Az alkalmazás betöltése után írjunk egy jegyzetet, majd kattintsunk a mentés gombra! Készítsünk egy új jegyzetet (+), ezt követően kattintsunk a megnyitás gombra! A megjelenő ablakban a jegyzeteink lesznek láthatók. Válasszunk ki egy jegyzetet, és a következő pillanatban már a főképernyőn láthatjuk a jegyzetünk tartalmát! Ha ezeket a lépéseket sikerült elvégeznünk, akkor eddig a feladatot helyesen oldottuk meg. Zárjuk be az alkalmazást, és térjünk vissza a Visual Studio-hoz!

A fájlkezeléshez hozzátartozik a beállítások kezelése is. A következőkben a jegyzetelő alkalmazásunkat kiegészítjük egy olyan beállítással is, melynek segítségével a felhasználó meghatározhatja, hogy a jegyzetek betűszíne milyen legyen.

Nyissuk meg a **Pages/Settings.xaml.cs** fájlt, és navigáljunk el az **IpForeground\_SelectionChanged** metódushoz, majd a feltételbe írjuk a következő kód sorokat:

```
IsolatedStorageSettings.ApplicationSettings["Foreground"] = selectedItem.Key;
NavigationService.Navigate(new Uri("/MainPage.xaml", UriKind.Relative));
```

- k. Itt az **IsolatedStorageSettings.ApplicationSettings Foreground** kulcsához hozzáadjuk a kiválasztott szín kulcsának értékét, majd visszavigálunk a **MainPage** oldalra.
- l. Ha már kiválasztottunk egy színt, és el is van tárolva, akkor azt a következő Settings page-re navigáláskor jelenítsük meg a felhasználó számára! Navigáljunk el a Settings konstruktorához, és töltsük be a használt szín indexét:

```
IsolatedStorageSettings.ApplicationSettings.TryGetValue<int>("Foreground",
    out currentIndex);
```

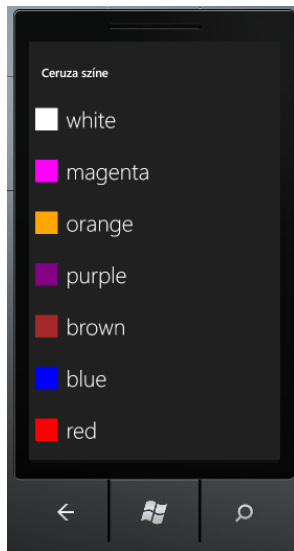
Itt az **IsolatedStorageSettings.ApplicationSettings.TryGetValue** metódusát használjuk. Használhatnánk az indexeres formát is, de a **TryGetValue** egyik nagy előnye az, hogy ha az adott kulcs (esetünkben **Foreground**) még nem létezik, akkor sem jelez hibát, a változó pedig az alapértelmezett értékét kapja (ebben az esetben a 0-t, ami kötés után a fekete színt reprezentálja).

- m. Természetesen a **MainPage** –nél is alkalmaznunk kell a kiválasztott színt: nyissuk meg a **MainPage.xaml.cs** fájlt, és navigáljunk el az **OnNavigatedTo** eseményhez! Írjuk hozzá a következő kódsort:

```
IsolatedStorageSettings.ApplicationSettings.TryGetValue<int>("Foreground", out
    foregroundColorIndex);
```

Csakúgy, mint a **Settings** page-nél, itt is kiolvassuk a foreground tulajdonságot, majd alkalmazzuk a szövegdobozon.

- n. Immár kész a teljes alkalmazás. Fordítsuk le a forráskódot (Ctrl+Shift+B), és ha valamilyen hibát tapasztalunk, nézzük át újra a kódot, és javítsuk azt! Ha minden rendben lezajlott, akkor indítsuk el az alkalmazásunkat (F5)!
- o. Készítsünk egy jegyzetet, majd mentjük el! Válasszuk ki a beállítások menüpontot, és a megjelenő ablakban válasszuk ki egy nekünk tetsző színt (8-6 ábra)! Ha kiválasztottunk egy új színt, akkor az alkalmazás rögtön visszanavigál a MainPage oldalra, ahol láthatjuk, hogy a jegyzetünk betűszíne megegyezik az általunk kiválasztott színnel.



8-6 ábra: Színválasztó lap

Ezen a gyakorlaton keresztül is láthattuk, hogy a fájlkezelés rendkívül egyszerű a Windows Phone –on. Minden alkalmazás rendelkezhet IsolatedStorage-dzsall, ahol fájlműveleteket végezhet, de ebből a kontextusból nem tud kibújni.

### Önálló feladatok

Fejlesszük tovább a jegyzetelő alkalmazást! Nincs megkötés arra, hogy mit és hogyan, legyünk kreatívak! Módosítsuk szabadon a UI-t, vagy vigyünk fel új menüpontokat! Rajtunk múlik a feladat, a minél leleményesebb megoldása a lényeg.

1. Készítsük el az alkalmazást úgy, hogy tudjunk jegyzetet törölni!
2. A jegyzet megnyitásánál jelenítsük meg valamilyen formában a jegyzet készítésének dátumát!
3. Tegyük megváltoztathatóvá jegyzetünk betűtípusát!

### IsolatedStorage – Tool

Azoknál az alkalmazásoknál, ahol sok a fájlművelet, hasznos lenne látni, hogy az alkalmazás használata során milyen fájlok keletkeztek, hogy azok az általunk meghatározott útvonalon jöttek-e létre, és hogy tartalmuk helyes-e. A könnyebb használat érdekében a Mango frissítés fejlesztőkészletének része lett az Isolated Storage Explorer Tool (**ISetool.exe**). Ez egy parancssori eszköz, amely segítségével műveleteket tudunk végezni a készülék vagy az emulátor Isolated Storage-án. Például: kilistázhatjuk a mappák tartalmát, másolhatunk, törölhetünk és minden alapvető fájlműveletet elvégezhetünk a segítségével. Az Isolated Storage Explorer Tool-t csak a regisztrált fejlesztői készülékeken lehet használni. Ezt a Toolt a **\$Program Files\Microsoft SDKs\Windows Phone\v7.1\Tools\IsolatedStorageExplorerTool** útvonal alatt találhatjuk. Az Isolated Storage Explorer használatához a következő szintaxist lehet használnunk:

```
ISetool.exe <ts|rs|dir[:device-folder]> <xd|de> <Product GUID> [<desktop-path>]
```

Parancs	Leírás
TS	Snapshot készítése
RS	Snapshot helyreállítása
DIR	Mappák és fájlok listázása
Device-folder	Mappa meghatározása
XD	Emulátoron történő használat
DE	Készüléken történő használat
Product GUID	A termék ProductID-je (WPAppManifest.xml)
Desktop-path	Egy könyvtár a számítógépünkön, ahová másolhatunk vagy írhatunk fájlokat az IsolatedStorage-ből.

Nézzük meg ezt az eszközt használat közben!

1. Készítsünk egy egyszerű Windows Phone 7 alkalmazást!
2. Keressük ki a ProductID-t a WPAppManifest.xml fájlból:

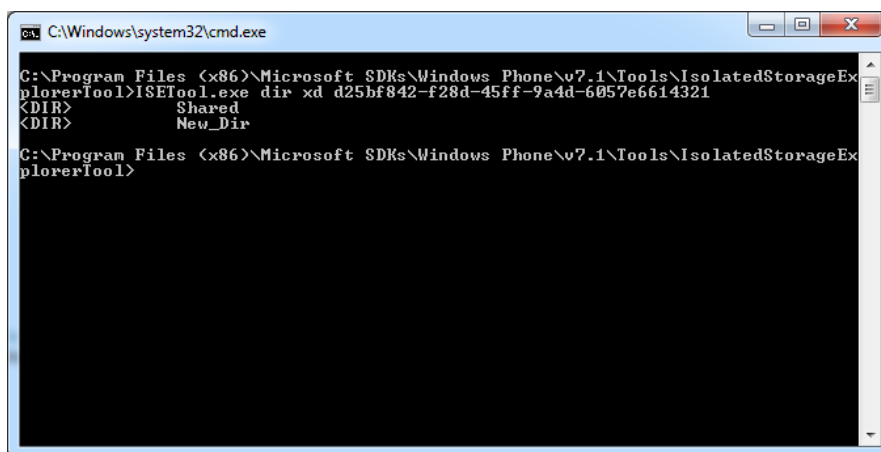
```
<App xmlns="" ProductID="{d25bf842-f28d-45ff-9a4d-6057e6614321}" Title="PhoneApp2"
```

3. Nyissunk meg egy parancssort!
4. Navigáljunk el a Windows Phone 7 SDK IsolatedStorageExplorer mappájába:  
**\$Program Files\Microsoft SDKs\Windows Phone\v7.1\Tools\IsolatedStorageExplorerTool**
5. Indítsuk el az alkalmazást az alábbi módon:

```
ISETool.exe dir xd d25bf842-f28d-45ff-9a4d-6057e6614321
```

(Amennyiben fizikai készülékről szeretnénk lekérdezni ezeket az információkat, az „xd” helyett „de” kapcsolót kell használnunk.) **Ne felejtjük el átírni a saját ProductID-nkat!**

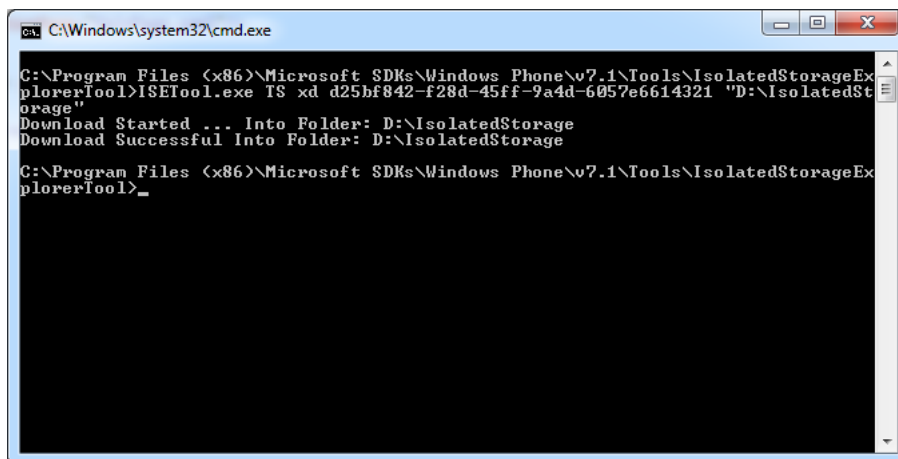
Az eredményt a 8-7 ábrán láthatjuk.



**8-7 ábra: Mappák és fájlok listázása emulátoron**

6. Listázni már tudunk, nézzük meg, hogyan másolhatunk fájlokat a telefonról a fájlrendszerünkre! Ezzel a funkcióval lementhetjük az alkalmazásunk mappáját, és a helyi gépünkön szabadon megvizsgálhatjuk azok tartalmát. Ehhez mindösszesen a következő parancsot kell kiadnunk:  
**ISETool.exe ts <xd|de> <Product GUID> <desktop-path>**, például (8-8 ábra):

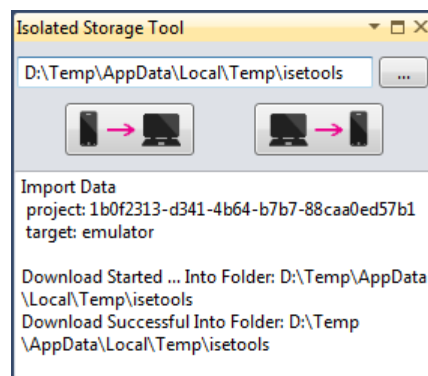
```
ISetool.exe TS xd d25bf842-f28d-45ff-9a4d-6057e6614321 "D:\IsolatedStorage"
```



**8-8 ábra: Fájlok lementése emulátorról**

Ezen a parancssori eszközön kívül természetesen vannak grafikus modulok is, sőt, olyan eszköz is létezik, amely a Visual Studio-ba épül be. Ilyen eszköz például az Isolated Storage Tool (8-9 ábra), amely szintén lehetőséget biztosít arra, hogy adatokat töltsünk le a telefonunkról (vagy emulátorunkról a számítógépünkre), illetve lehetőséget biztosít arra is, hogy a gépünkről töltsünk fel adatokat a telefonon futó alkalmazásunk könyvtárába. Ezt a kiegészítőt a következő oldalról tölthetjük le:

<http://visualstudiogallery.msdn.microsoft.com/fcd19b08-f8bc-4397-84bc-c10cd44ca673?SRC=VSI&DE>



**8-9 ábra: A Visual Studióba beépülő Isolated Storage Tool**

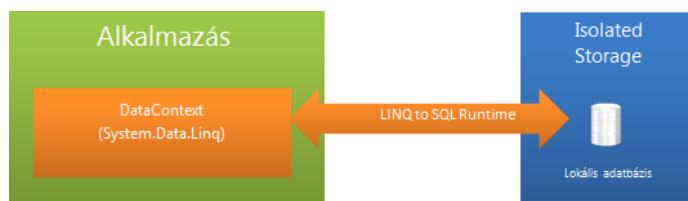
## Lokális adatbázisok használata Windows Phone 7-en

Lokális relációs adatbázis készítésére és használatára a Windows Phone 7 első (7.0 –ás) változatában nem volt beépített lehetőség, a Mango frissítéssel viszont az Isolated Storage-ben az alkalmazásunk elhelyezhet relációs adatbázist is. Ezt Microsoft SQL Server Compact Edition (CE) teszi lehetővé, amely egy fájl alapú, az alkalmazásba beépülő módon működő relációs adatbázis-kezelő. Ezt az adatbázist csak a Linq To SQL API segítségével érhetjük el, és így objektumorientált módon kezelhetjük adatainkat. Az SQL Servernél megszokott Transact SQL segítségével közvetlenül nem érhetjük el az adatbázist. Az adatbázist csak saját alkalmazásunk írhatja és olvashatja, azt –ugyanúgy, mint az Isolated Storage esetén – nem oszthatjuk meg a többi alkalmazással!

### Linq To SQL

A Linq To SQL egy ún. ORM (Object Relational Mapping) keretrendszer, amelyet a .NET Framework tartalmaz. A Linq To SQL lehetőséget biztosít arra, hogy kódunkban beágyazva fogalmazzunk meg lekérdezéseket, és azokat lefuttassuk az adott adatbázismotoron. Ahhoz, hogy ez működjön, az

adatbázisok világát közelebb kell hoznunk az objektumorientált világhoz, ugyanis az adatbázisoknál táblák vannak, amelyekben sorok és oszlopok találhatók, míg egy objektumorientált programban entitások és tulajdonságok vannak. Ezt a két világot meg kell feleltetni egymásnak, hogy dolgozni tudjunk az adatokon. Ezt a megfeleltetést nevezzük *mapping*nek. A megfeleltetésen kívül szükségünk lesz egy proxy objektumra is, ami megvalósítja a kapcsolatot az adatbázissal és az egyéb specifikus funkciókat is elvégzi, ezt *DataContext*nek nevezzük (8-10 ábra).



**8-10 ábra: Lokális adatbázisok**

## **DataContext**

A **DataContext** egy *proxy* objektum, amely az adatbázist reprezentálja. A **DataContext Table** típusú objektumokat tartalmaz, amely az adatbázis tábláit reprezentálja. Minden **Table** objektum entitásokból épül fel, amelyek pedig a táblák oszlopait írják le. Minden entitás egy „Plain Old CLR Object” (azaz POCO), amelyek speciális attribútumokkal van ellátva. Ezekkel az attribútumokkal definiáljuk, hogy az adott objektum milyen módon reprezentálja az adatbázist. Ennek megértéséhez képzeljük el az alábbi példát:

Van egy **Users** táblánk, melynek van egy **Name** és egy **Age** oszlopa. Ezt a táblát a **DataContext**-ben egy **User** osztály reprezentálja, amelynek két tulajdonsága van: a **Name** és az **Age**. Azon kívül, hogy van egy ilyen osztályunk, azt speciális attribútumokkal kell ellátnunk az adatbázis speciális megkötéseinek, tulajdonságainak leírásához.

## **Mapping**

A mapping maga a megfeleltetés egy adattábla és az azt reprezentáló objektum között. Egyszerű POCO (Plain Old CLR Object) objektumokat és tulajdonságaikat speciális jelzésekkel – attribútumokkal – jelölhetünk meg. Az osztályt ellátjuk egy **Table** attribútummal, ezzel jelezve, hogy ez az osztály tulajdonképpen az adatbázis egy táblájának reprezentációja lesz. Az osztály tulajdonságait pedig a **Column** attribútummal látjuk el, hogy jelezzük, ezek a tulajdonságok képzik az adatbázis oszlopait.

A **Table** attribútumnál egy **Name** tulajdonságot is megadhatunk. Ha megadjuk, akkor az adatbázisban található tábla neve az lesz, amit megadtunk. Ha nem adunk meg nevet, akkor automatikusan az osztály neve lesz adatbázis táblájának a neve.

Nézzünk egy példát, amelyben a **Countries** táblát a **Country** osztálynak feleltetjük meg! Az itt leírt **Country** osztály az alkalmazás futtatásánál egy **Countries** nevű táblát hoz létre az adatbázisban:

```
[Table(Name="Countries")]
public class Country
{
}
```

Ebben a példában a **Country** nevű adatbázis táblához a megegyező nevű osztály tulajdonságait rendeljük:

```
[Table]
public class Country
{
}
```

A táblákat és az üzleti objektumainkat már egymáshoz rendeltük, viszont a táblák oszlopait még nem rendeltük tulajdonságokhoz! Ehhez a **Column** attribútumot hívjuk segítségül! Csak akkor láthatjuk el a tulajdonságainkat ezzel az attribútummal, ha az osztály el van látva a **Table** jelölővel. A **Column** attribútumnál további lehetőségeink is vannak: meghatározhatjuk például, hogy az adott oszlop elsődleges kulcsú-e, adatbázis állítja-e elő az adott oszlop értékét (Például: **AutoIncrement** –es ID mező). A **Name** tulajdonságával meghatározhatjuk, hogy az adott nevű adattábla oszlop melyik tulajdonsághoz legyen rendelve:

```
[Table]
public class City
{
    [Column(IsPrimaryKey=true, IsDbGenerated=true)]
    public int ID { get; set; }
}
```

A **Column** attribútumnál határozhatjuk meg azt is, hogy az adott érték lehet-e null:

```
[Column(CanBeNull=true)]
public string Name { get; set; }
```

Az alábbi táblázat a rendelkezésre álló attribútumokat sorolja fel, és képességeit írja le:

Attribútum	Leírás
[Table]	Táblák megfeleltetésére szolgál.
[Column]	Oszlopok megfeleltetésére szolgál.
[Index(Columns="Column1,Column2 DESC", IsUnique=true, Name="MultiColumnIndex")]	További indexek meghatározására szolgál, minden index egy vagy több oszlopot fedhet le.
[Association(Storage="ThisEntityRefName", ThisKey="ThisEntityID", OtherKey="TargetEntityID")]	Táblák közötti kapcsolat meghatározására szolgál (pl. idegen kulcs → elsődleges kulcs kapcsolat).

### Adatbázis létrehozása

Ahhoz, hogy létrehozzunk egy adatbázist, egy DataContext típust és legalább egy entitás osztályt létre kell hoznunk. Ezek az osztályok lesznek felelősek a mappingért, ennek megfelelően ezeket az objektumokat el kell látni a megfelelő attribútumokkal. A leképezéshez szükségünk lesz a **System.Data.Mapping.Linq** névtérre.

Az adatbázist úgy tudjuk létrehozni, hogy készítünk egy egyszerű osztályt, és ellátjuk a megfelelő attribútumokkal. Például ha szeretnénk létrehozni egy táblát az ügyfeleink nyilvántartására, ehhez létrehozzunk egy **Customer** osztályt, amibe felveszünk néhány tulajdonságot.

Nézzünk egy konkrét példát az adatbázis leképezésére! Létrehozzuk a **Customer** osztályt, amelyet ellátunk egy **Table** attribútummal. Ebben meghatározzuk, hogy ez az osztály a **Customers** táblához fog kapcsolódni. Az osztályban elhelyezhetünk saját tulajdonságokat, melyeket a **Column** attribútummal rendelünk az adatbázis oszlopaihoz.

```
using System.Data.Linq.Mapping;

namespace PhoneDatabase
{
    [Table(Name="Customers")]
    public class Customer
    {
        [Column(IsPrimaryKey=true)]
```



```

        public int CustomerID { get; set; }
        [Column]
        public string CompanyName { get; set; }
        [Column]
        public string ContactName { get; set; }
        [Column]
        public string City { get; set; }
    }
}

```

A **CustomerID** tulajdonságot a tábla elsődleges kulcsaként jelöljük meg. Ahhoz, hogy létrehozzuk, vagy lekérdezéseket fogalmazzunk meg ezen a táblán, szükségünk lesz egy **DataContext**-re is. Ehhez létre kell hoznunk egy újabb osztályt – ezt nevezzük el mondjuk **CustomerDataContext**-nek! Ezt az osztályt a **DataContext** osztályból kell származtatnunk (a **DataContext** a **System.Data.Linq** névtérben található). Hozzuk létre a **Customers** tulajdonságot, amely legyen **Table<Customer>** típusú! Ez reprezentálja a **Customers** táblában lévő adatokat. A tulajdonságot a konstruktorban inicializáljuk. Ennek a konstruktornak paraméterként átadjuk az adatbázis eléréséhez szükséges **connectionString** paramétert, amelyet átadunk a **DataContext** konstruktorának:

```

using System.Data.Linq;

namespace PhoneDatabase
{
    public class CustomerDataContext : DataContext
    {
        private Table<Customer> customers;
        public Table<Customer> Customers
        {
            get { return customers; }
        }

        public CustomerDataContext(string connectionString) : base (connectionString)
        {
            customers = GetTable<Customer>();
        }
    }
}

```

Most már kész az entitás osztályunk, és kész a **DataContext** is. Itt az ideje az adatbázist fizikailag is létrehozni! Ehhez a **DataContext** példány **CreateDatabase** metódusát kell meghívunk. Célszerű az adatbázis létrehozása előtt ellenőrizni, hogy az adott adatbázis létezik-e. **DataContext** példányosításakor meg kell adnunk a connection stringet, ami ebben az esetben **isostore:/Customer.sdf**, azaz az Isolated Storage legfelső mappájában egy **Customer.sdf** fájlra hivatkozunk.

```

using (CustomerDataContext db = new CustomerDataContext("isostore:/Customer.sdf"))
{
    if (!db.DatabaseExists())
    {
        db.CreateDatabase();
    }
}

```

Most már van egy adatbázisunk, amiben van egy **Customer** tábla, de macerás munka ezt minden esetben manuálisan létrehozni. Nincs lehetőség valami kényelmesebb, megszokottabb módszerre? De van! A következő részben ezzel fogunk megismerkedni.

## Adatbázis létrehozása II.

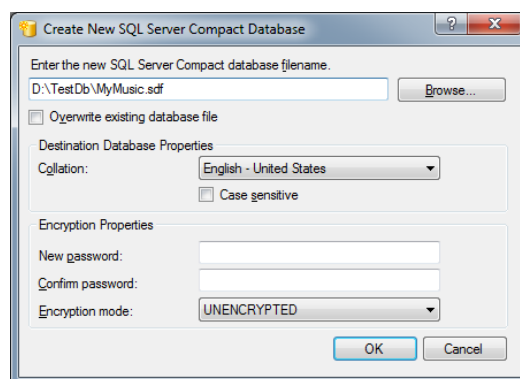
Bár elég egyszerű létrehozni adatbázist úgy, hogy van egy entitás osztályunk és egy **DataContext** példányunk, mégis egy bonyolultabb adatbázis szerkezetének létrehozása sok munkával járhat, különösen, ha nagyon sok táblánk van, és a táblák között kapcsolatok is vannak. Nemcsak létre tudunk hozni adatbázist, arra is van lehetőségünk, hogy már meglévő adatbázist (**sdf**) hozzáadjunk a projekthez és kezeljük azt. Nézzük meg ezt az esetet lépésről lépésre! A célunk tehát az, hogy létrehozzunk egy SQL Server CE adatbázist, előállítsuk hozzá az entitás osztályokat és a **DataContext**et, majd felhasználjuk egy WP7 alkalmazásban.

1. Indítsuk el az SQL Server Management Studio 2008 Express változatát *(ha ez az alkalmazás nincs meg, akkor legegyszerűbben a Web Platform Installer-rel telepíthetjük fel!)*
2. Indítás után egy Login képernyő fogad. Itt a Server type-nál válasszuk az **SQL Server Compact** –ot (8-11 ábra)!



8-11 ábra: A Connect To SQL Server dialógus

3. A Database file-nál válasszuk a **<New Database...>** menüpontot, ekkor felugrik a **Create New SQL Server Compact Database** ablak (8-12 ábra). A fájl neve ebben az esetben a **D** meghajtó **TestDb** mappájára mutat, ebben a mappában fogja a **MyMusic.sdf** adatbázist létrehozni a Management Studio.

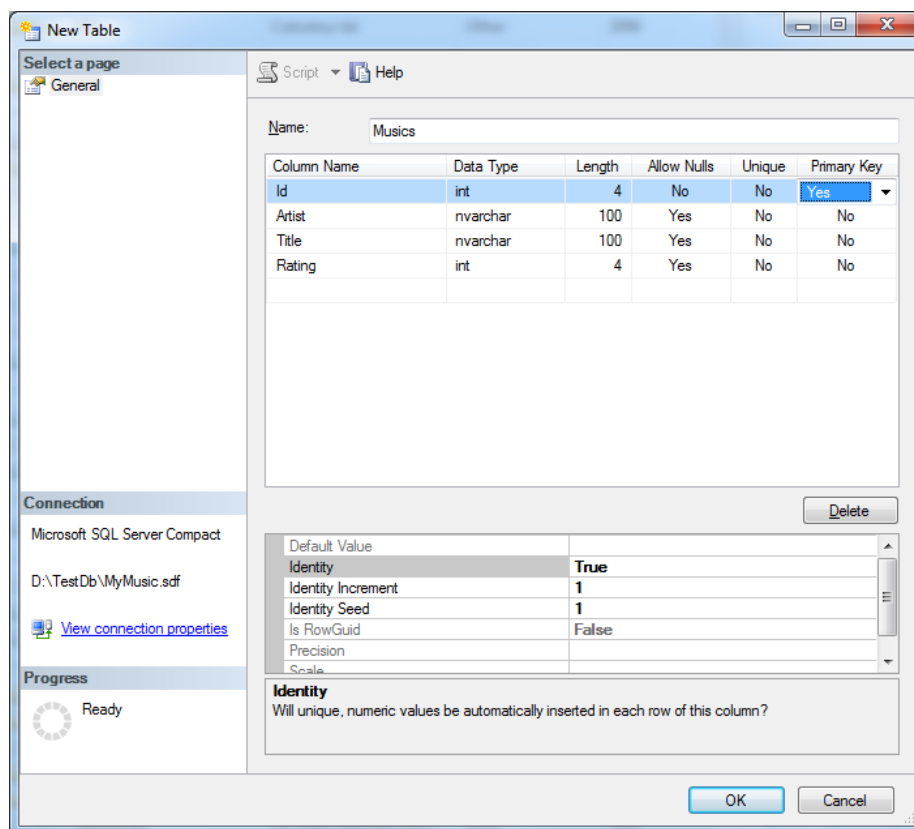


8-12 ábra: Create New SQL Server Compact Database

Most nem foglalkozunk a titkosítással, így az Encryption mode maradhat UNENCRYPTED. Ezt követően kattintsunk az OK gombra! Ugyan most kapunk egy figyelmeztetést, hogy az adatbázisunk nem lesz levédve, de ezzel most ne törődjünk, és kattintsunk a **Yes** gombra! A későbbiek folyamán a titkosítással is fogunk foglalkozni.

4. Ezt követően visszaugrunk a **Connect To Server** ablakhoz, itt kattintsunk a **Connect** gombra, ekkor az SQL Server Management Studio-ból kezelhetjük ezt az adatbázist.
5. Az **Object Explorer**-ben kattintsunk a **Tables** elemre jobb egérgombbal, majd a megjelenő helyi menüben válasszuk ki a **New Table** menüpontot!

- A megjelenő **New Table** ablakban definiáljuk a **Musics** adattáblát, ahogyan azt a 8-13 ábra mutatja!



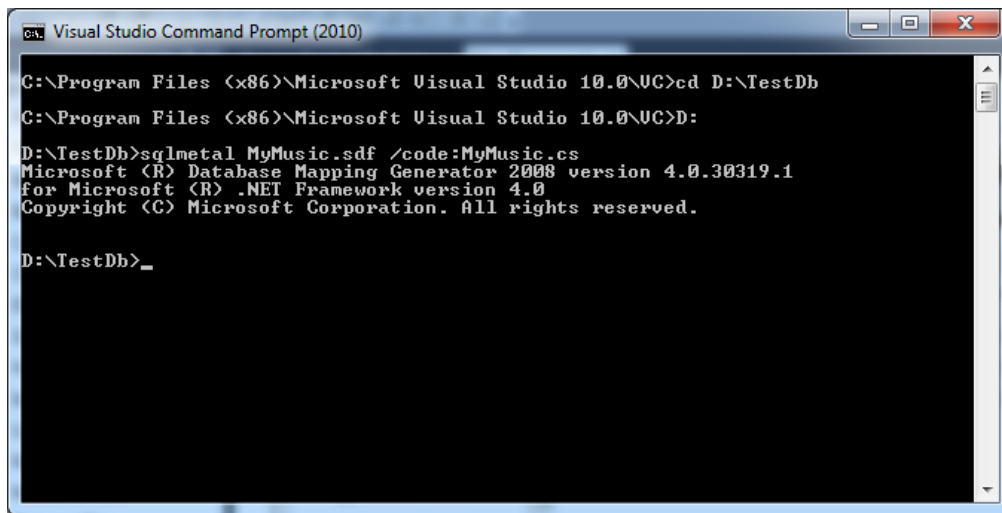
**8-13 - A Musics adattábla definiálása**

Az **ID** oszlop egy **int** típusú elsődleges kulcs, **True** értékre állított **Identity** tulajdonsággal és annak alapértelmezett értékeivel. Az **Artist** és a **Title** mezők **nvarchar** típusúak, a **Rating** mező **int** típusú. Kattintsunk az OK gombra!

- Most már elkészült az adatbázisunk, bezárhatjuk az SQL Server Management Studio-t. Az adatbázist Windows Phone alól viszont csak Linq To SQL-lel érhetjük el. Ehhez létre kell hoznunk az entitásokat reprezentáló osztályokat. Persze itt is megtehetnénk, hogy kézzel generáljuk az entitás osztályokat és a **DataContext**et, de van szerencsére egy **SQLMetal** nevezetű eszköz is, ami **bár hivatalosan még mindig nem támogatott eszköz, a Windows Phone 7 fejlesztés során mégis nagy hasznát tudjuk venni**.
- Indítsuk el a Visual Studio 2010 Command prompt-ot!
- A Command Prompt-ban az egyszerűség kedvéért navigáljunk el abba a mappába, ahol az adatbázist elkészítettük (Ez jelen esetben a **D:\TestDb** mappa), majd indítsuk el az **SqlMetal.exe** alkalmazást az alábbi paraméterekkel:

```
sqlmetal MyMusic.sdf /code:MyMusic
```

Ebben az esetben a **MyMusic.sdf**-ből készítünk **MyMusic.cs** fájlt (8-14 ábra). A forrásfájlban lesznek az entitáosztályaink, valamint a **DataContext** is. Ez a forrásfájl jelenleg az adatbázis mellett található.



8-14 ábra: Az SqlMetal használat közben

10. Most már kész az adatbázisunk és a hozzá tartozó osztályok is elkészültek! Itt az ideje használatba venni ezeket: indítsuk el a Visual Studio-t, és készítsünk egy új Windows Phone Application projektet (legyen a neve **DatabaseSample**)!
11. Amint elkészült az alkalmazás sablonja, adjuk hozzá az adatbázist és az **SqlMetal** által generált forrásfájlt! Ehhez a Solution Explorer-en kattintsunk jobb egér gombbal, majd a megjelenő helyi menüben válasszuk ki az **Add Existing Item** menüpontot, és válasszuk ki a fentebb említett fájlokat!
12. Adjuk hozzá a projekthez a **System.Data.Linq** assembly-t is! Jobb egérgomb a **References**-re a Solution Explorerben, és válasszuk ki az **Add Reference** menüpontot!
13. A megjelenő ablakban kattintsunk a **.Net** fülre, és válasszuk ki a **System.Data.Linq** névteret!
14. Ezt követően, ha megpróbálnánk lefordítani az alkalmazást, a következő hibaüzenetet kapnánk:

The type or namespace name 'IDbConnection' does not exist in the namespace 'System.Data' (are you missing an assembly reference?)

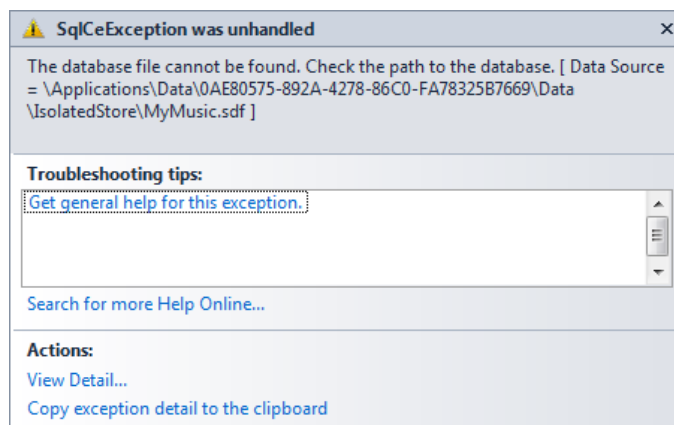
15. Ez azért van, mert a Windows Phone 7 **System.Data** névterében valóban nem létezik **IDbConnection**, az **SqlMetal** viszont legenerálta azt. Mint korábban is említettük, az **SqlMetal** nem generál teljesen kompatibilis kódot a Windows Phone 7-tel, hivatalosan nem is támogatott eszköz, mégis megkönnyíti a fejlesztést. Ahhoz, hogy ez a DataContext helyes legyen, alakítsuk át a két konstruktort az alábbi módon:

```
public MyMusic(System.Data.IDbConnection connection) :
    base(connection, mappingSource)
{
    OnCreated();
}

public MyMusic(System.Data.IDbConnection connection, System.Data.Linq.Mapping.MappingSource
mappingSource) :
    base(connection, mappingSource)
{
    OnCreated();
}
```

Ettől a ponttól kezdve már használatba vehetjük az adatbázist. Igen ám, de ha megpróbálnánk elérni azt, akkor a 8-15 ábrán láthatóhoz hasonló hibát kapnánk:

```
MyMusic db = new MyMusic("isostore:/MyMusic.sdf");
```

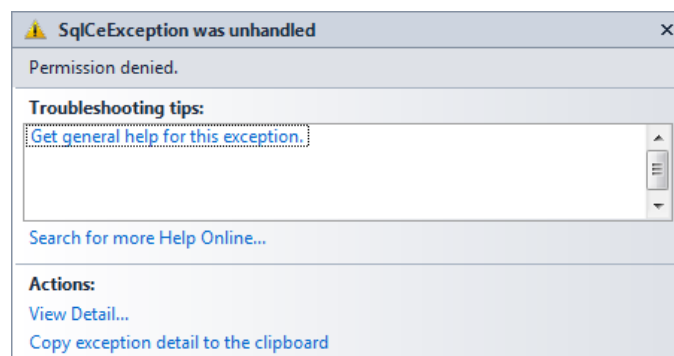


8-15 ábra: *SqlException was unhandled. The Database file cannot be found.*

16. Ezt a hibát azért kapjuk, mert az IsolatedStorage-ban nem szerepel ez az adatbázis. Bár Content-re van állítva az adatbázis Build Action tulajdonsága, és az XAP-ban is benne van ez az adatbázis, mégsem található az Isolated Storage-ban! Ahhoz, hogy ezt az adatbázist használjuk (Reference Database), egy speciális connection stringet kell megadnunk:

```
MyMusic db = new MyMusic("Data Source = 'appdata:/MyMusic.sdf'; File Mode = read only");
```

17. Az adatbázist most már elérhetjük, de csak lekérdezéseket fogalmazhatunk meg rajta, azaz **csak olvasható** az adatbázis tartalma. Amint módosítani szeretnénk, azonnal egy **Permission Denied** hibaüzenetet kapnánk, amint azt a 8-16 ábra is mutatja. (Az adatok módosításáról és hozzáadásáról a következő alfejezetben olvashatunk.)



8-16 ábra: *Permission denied üzenet az adatbázis módosításakor*

18. Ahhoz, hogy ezt elkerüljük, az adatbázist át kell másolnunk az Isolated Storage-ba. Ehhez használhatjuk például az alábbi segédmetódust:

```
public class DatabaseHelper
{
    public static void MoveReferenceDatabase()
    {
        IsolatedStorageFile iso = IsolatedStorageFile.GetUserStoreForApplication();

        using (Stream input = Application.GetResourceStream(new Uri("MyMusic.sdf",
            UriKind.Relative)).Stream)
        {
            using (IsolatedStorageFileStream output = iso.CreateFile("MyMusic.sdf"))
            {
                byte[] readBuffer = new byte[4096];
                int bytesRead = -1;
            }
        }
    }
}
```

```
        while ((bytesRead = input.Read(readBuffer, 0, readBuffer.Length)) > 0)
        {
            output.Write(readBuffer, 0, bytesRead);
        }
    }
}
```

19. Amint meghívjuk a **MoveReferenceDatabase** metódust, az átmásolja az adatbázist az Isolated Storage-ba, és ezek után a megszokott módon tudjuk lekérdezni, módosítani az adatbázis tartalmát. Ne felejtsük el, hogy ebben az esetben a connection stringet a hagyományost isostore értékre kell visszaállítanunk:

```
MyMusic db = new MyMusic("isostore:/MyMusic.sdf");
```

Láthattuk, hogy milyen lépéseket kell megtennünk ahhoz, hogy már meglévő adatbázist használjunk fel a Windows Phone 7 készülékünkön. Bár ez a módszer nem teljesen támogatott, mégis nagyon sokszor hasznos lehet. Ha csak olvasható adatbázist akarunk használni, akkor egyszerűen csak hozzáadjuk az adatbázist és az legenerál egy forrásfájlt a projektünkhöz, ha azonban módosítani is szeretnénk, akkor egy segédmetódust is definiálnunk kell az adatbázist mozgatásához.

Az adatbázisunk még üres, itt az ideje megismerkedni azzal, hogy hogyan töltsük fel az adatbázist adatokkal, és hogy hogyan kérdezzük le azokat!

### ***Adatok felvitele – INSERT***

A Linq To SQL használata során nem kell INSERT SQL Scriptet írunk, mint a T-SQL-nél! Egyszerűen csak annyi a dolgunk, hogy példányosítjuk a megfelelő entitás osztályt, és a tulajdonságait feltöltjük a kívánt értékekkel, majd ezt az osztálypéldányt átadjuk az **InsertOnSubmit** metódusnak. Az adatbázisban mindaddig nem kerül az új rekord letárolásra, amíg a **SubmitChanges** metódust meg nem hívjuk:

```
Customer myCust = new Customer()
{
    CustomerID = 1,
    CompanyName = "Livesoft",
    ContactName = "Attila Turoczy",
    City = "Budapest"
};

db.Customers.InsertOnSubmit(myCust);
db.SubmitChanges();
```

### ***Adatok lekérdezése – SELECT***

Az adatok lekérdezéséhez is a Linq To SQL-t hívjuk segítségül. A lekérdezések szintaktikája nagyon egyszerű: akár Linq szintakszis használatával, akár bővítő metódusokkal le tudjuk kérdezni az adatainkat. Nézzünk néhány lekérdezést! A következőkben mindkét lekérdezési formát bemutatjuk.

Az összes adat lekérdezése:

```
var result = from c in db.Customers
              select c;

-----
var result = db.Customers.Select(s => s);
```

Egy oszlop lekérdezése:

```
var result = from c in db.Customers
              select c.CompanyName;
-----
var result = db.Customers.Select(s => s.CompanyName);
```

Ha az eredményhalmazt szűkíteni szeretnénk, akkor egy **where** feltételt kell meghatározunk:

```
var result = from c in db.Customers
              where c.City == "Budapest"
              select c;
-----
var result = db.Customers.Where(s => s.City == "Budapest");
```

Sorbarendezés:

```
var result = from c in db.Customers
              orderby c.City
              select c;
-----
var result = db.Customers.OrderBy(s => s.City);
```

Látható, hogy a szintaktika logikusan felépített, használata nagyon egyszerű.

## ***Adatok módosítása – UPDATE***

Ha frissíteni szeretnénk az adatbázis tartalmát, akkor is nagyon egyszerű dolgunk van: csakúgy, mint korábban, itt sem kell SQL Script-eket írunk, mindössze annyi a dolgunk, hogy a lekérdezett adatok tulajdonságát átírjuk a megfelelő értékre, majd meghívjuk a **SubmitChanges** metódust.

Az alábbi példában lekérdezzük az összes ügyfelünket, majd minden ügyfelünk városát átírjuk Seattle-re, és a végén meghívjuk a **SubmitChanges** metódust:

```
var result = from c in db.Customers
              select c;

foreach (var item in result)
{
    item.City = "Seattle";
}

db.SubmitChanges();
```

## ***Adatok törlése – DELETE***

Ha az adatbázisunkból szeretnénk törölni adatokat, az is rendkívül egyszerűen kivitelezhető. Válasszuk ki azt az rekordot (objektumot), amit törölni szeretnénk, adjuk át paraméteréül a **DeleteOnSubmit** metódusnak, és hívjuk meg a **SubmitChanges**-t!

```
var selectedItem = (from c in db.Customers
                    where c.CustomerID == 1
                    select c).First();

db.Customers.DeleteOnSubmit(selectedItem);
db.SubmitChanges();
```

### Az adatbázis biztonsága

A lokális adatbázis lehetőséget biztosít arra, hogy az adatbázis tartalma titkosítva legyen, és azt csak jelszóval tudjuk elérni. Ha az adatbázis titkosított, csak jelszóval érhetjük el. Ezt a jelszót a connection string-ben kell átadnunk! Az adatbázis titkosításához AES-128, míg a jelszóhoz az SHA-256 titkosítási eljárást használja a rendszer. Az alábbi példában látható, hogyan hozunk létre egy biztonságos adatbázist:

```
CustomerDataContext db = new CustomerDataContext("Data
Source='isostore:/Customer.sdf';Password='Pa$$word1'");
if (!db.DatabaseExists())
{
    db.CreateDatabase();
}
```

### Adatbázis kezelés gyakorlat

A következő gyakorlatban egy mini bevásárlólista alkalmazást fogunk elkészíteni. Az alkalmazás felhasználó felülete már előre el van készítve (8-17 ábra), nekünk csak az adatbázist kell megterveznünk, illetve a lekérdezéseket megírni. A kiinduló projektet a <http://devportal.hu/wp7> oldalról tölthetjük le.



8-17 ábra: A bevásárló alkalmazás egy képernyője

1. Indítsuk el a Visual Studio 2010-et és töltsük be a **LocalDatabaseDemo.sln** fájlt!
2. Nyissuk meg a **ShopItem.cs** fájlt az **Entities** mappán belül, majd oldjuk fel a **System.Data.Linq.Mapping** névteret!

```
using System.Data.Linq.Mapping;
```

3. A ShopItem osztály definícióját lássuk el a **[Table]** attribútummal!

```
[Table]
public class ShopItem {
```

4. Meg kell határoznunk, hogy milyen oszlopokkal szeretnénk dolgozni. A **ShopItem** törzsét a következő módon alakítsuk ki:

```
[Column(IsPrimaryKey=true, IsDbGenerated=true)]
public int Id { get; set; }
[Column]
public string Title { get; set; }
```



```
[Column]
public int Price { get; set; }
[Column]
public bool IsBought { get; set; }
```

A **Title** tulajdonságban tároljuk el a vásárolni kívánt termék nevét, a **Price** tulajdonságban a termék hozzávetőleges árát, míg az **IsBought** azt mutatja, hogy az adott terméket megvettük-e már. Látható, hogy a tulajdonságok mindegyikét egy **[Column]** attribútummal láttuk el. Speciális eset az **Id** tulajdonság, ugyanis nemcsak azt jelöljük, hogy ez egy oszlop, hanem azt is, hogy ez egyúttal egy olyan elsődleges kulcs, amely értékét az adatbázis fogja előállítani. Ezzel kész az entitásosztály definíciója.

5. Létre kell hoznunk egy **DataContext**et is, hogy az adatbázishoz tudjunk kapcsolódni. Nyissuk meg a **ShopDataContext.cs** fájlt, majd oldjuk fel a **System.Data.Linq** névteret!

```
using System.Data.Linq;
```

6. A **ShopDataContext** osztályt származtassuk le a **DataContext** osztályból!

```
public class ShopDataContext : DataContext {
```

7. A **ShopDataContext** kódját pedig a következőképpen írjuk meg:

```
private Table<ShopItem> shopItem;

public Table<ShopItem> ShopItem
{
    get { return shopItem; }
}

public ShopDataContext(string connectionString) : base(connectionString)
{
    this.shopItem = GetTable<ShopItem>();
}
```

8. Nyissuk meg az **App.xaml.cs** fájlt, és az **Application\_Startup** metódusban hozzuk létre az adatbázisunkat a **CreateDatabase** metódus segítségével! Abban az esetben, ha már létezik az adatbázis, nem történik semmi.

```
ShopDataContext db = new ShopDataContext("isostore:/shop.sdf");
if (!db.DatabaseExists())
{
    db.CreateDatabase();
}
```

9. Már elkészült az adatbázisunk, de még üres. Töltsük fel adatokkal! Nyissuk meg a **Pages** mappa **NewShopItem.xaml.cs** fájlját, és navigáljunk el a **save\_Click** eseményhez, majd adjuk hozzá az alábbi kódsorokat:

```
try
{
    ShopDataContext db = new ShopDataContext("isostore:/shop.sdf");
    ShopItem shopItem = new ShopItem()
    {
        Title = txtTitle.Text,
        Price = Convert.ToInt32(txtPrice.Text)
    };
};
```

```
db.ShopItem.InsertOnSubmit(shopItem);
db.SubmitChanges();

NavigationService.Navigate(new Uri("/MainPage.xaml", UriKind.Relative));
}
catch (Exception ex)
{
    MessageBox.Show("Hiba: " + ex.Message, "Hiba!", MessageBoxButton.OK);
}
```

Itt egy `ShopDataContext`-et példányosítunk, aminek megadunk egy connection string-et (ezt globálisan is megtehetnénk). Ezt követően egy **ShopItem** példányt hozunk létre, amely tulajdonságainak az értékeit a felhasználói felületi hozzá tartozó elemei határozzák meg. Ha ez megvan, meghívjuk az **InsertOnSubmit** metódust, majd a **SubmitChanges** metódussal eltároljuk az adatbázisba. Ezt követően visszavigálunk az eredeti oldalra.

Nem töltöttük ki az **IsBought** tulajdonság értékét, így az alapértelmezett értéke a **false** lesz, azaz még nem vásároltuk meg az adott terméket.

10. Itt az ideje megjeleníteni az adatainkat. A konstruktor előtt most hozunk létre egy `ShopDataContext` példányt! (Task – 9)

```
ShopDataContext db = new ShopDataContext("isostore:/shop.sdf");
```

11. Navigáljunk el a **GetData** metódusba, és kérdezzük le azokat a termékeket az adatbázisból, amiket még nem vettünk meg!

```
lstShopItems.DataContext = from c in db.ShopItem
                           where c.IsBought == false
                           select c;
```

12. Azért, hogy lássuk, hogy mennyivel tud többet a Linq, fejlesszük tovább a **GetData** metódust: egy feltételben vizsgáljuk meg, hogy hány darab olyan termékünk van, amit nem vettünk még meg! Ha ennek az értéke nagyobb, mint nulla, akkor megvizsgáljuk azt is, hogy a kiválasztott termékeknek mennyi a teljes ára:

```
if (db.ShopItem.Where(s => s.IsBought == false).Count() > 0)
{
    txtSum.Text = String.Format("{0} Ft", db.ShopItem
        .Where(s => s.IsBought == false).Sum(s => s.Price));
}
else
{
    txtSum.Text = "0 Ft";
}
```

13. Már csak egy lépés maradt hátra, mégpedig az, hogy ha vásárolunk valamit a listáról, akkor kihúzzuk azokat a termékeket, amelyeket már betettünk a kosárba. Ehhez navigáljunk el a **btnCheck\_Click** eseményhez és írjuk be a következőt:

```
int selectedIndex = Convert.ToInt32((e.OriginalSource as Button).Content.ToString());

var selectedItem = (from c in db.ShopItem
                    where c.Id == selectedIndex
                    select c).FirstOrDefault();
```

```
if (selectedItem != null)
{
    selectedItem.IsBought = true;
    db.SubmitChanges();
}
GetData();
```

Első lépésben lekérdezzük a terméket reprezentáló gomb tartalmát. Az ugyan nem jelenik meg a felhasználói felületen, de a háttérben bele van írva az adott termék azonosítója (**Id** tulajdonság). Ezután lekérdezzük az adott azonosítójú terméket az adatbázisból. A Linq kifejezés végén egy **FirstOrDefault** bővítő metódus áll, amely az első elemet veszi ki a visszaadott eredmény gyűjteményből. Abban az esetben, ha nincs ilyen azonosítójú elem, akkor az objektum alapértelmezett értékével tér vissza. Ha ez az érték nem null, akkor az **IsBought** tulajdonságot **true** értékre állítjuk, majd meghívjuk a **SubmitChanges** metódust. Az egyszerűség kedvéért ezt követően meghívjuk a **GetData** metódust, amivel újra lekérdezzük az összes nem megvásárolt terméket és az ezekhez tartozó összeget.

14. Elkészült az alkalmazásunk. Indítsuk el, és nézzük meg, hogy hogyan működik! Kattintsunk az új ( + ) gombra, és adjunk hozzá egy új terméket! Amikor a mentés gombra kattintunk, akkor az adatokat elmentjük az adatbázisba. Ezt követően a főképernyőn találjuk magunkat, ahol az előzőleg felvitt termékek láthatóak. Vigyünk fel még néhány elemet, majd kattintsunk a termékek neve melletti pipára! Miután rányomtunk, az adott elem kikerül a listából, és az összesen értéke is a kipipált elem értékével csökken. Abban az esetben, ha ez a folyamat mégse így történne, nézzük át újra a lépéseket, és a hibákat javítsuk!

## Összefoglalás

Ebben a fejezetben megnéztük, hogy hogyan is kell a Windows Phone 7 alkalmazásokban adatot kezelni: hogyan kell adatokat lementeni, beállításokat eltárolni, valamint hogyan lehet az adatokat egy relációs adatbázisban eltárolni. Láthattuk, hogy a Windows Phone 7 készülékeken is a .NET-ben megszokott, letisztult megoldásokat kapjuk. Ahogyan Silverlight alatt is van Isolated Storage, itt elérhető a Linq To SQL technológia.



# 9. Kommunikáció szerverrel

A telefonok hardvere robbanásszerűen fejlődik. A Microsoft kötelezően előírta, hogy minden Windows Phone operációs rendszert futtató készüléknek legalább 256 megabájt memóriával és 1 GHz körüli processzorral kell rendelkeznie. Ez azt jelenti, hogy a telefonok nagyságrendileg olyan erősek, mint egy netbook. A legsúlyosabb korlátozást viszont nem is a számítási kapacitás adja, hanem az akkumulátor: ritka, hogy egy rendszeresen használt okostelefon 1-2 napnál tovább bírja újratöltés nélkül. Ezért nagyon fontos spórolnunk az akkumulátor kapacitásával, aminek legjobb módja, hogy a számításigényesebb feladatokat kiszervezzük – célszerűen szerveroldalra.

Egy másik ilyen lényeges trend a közösségi média rohamos terjedése. Legyen szó szinte bármilyen alkalmazásról, a felhasználók elvárják a többi felhasználóval, központi adatbázissal való kapcsolat lehetőségét.

Ezenfelül épp napjainkban lehetünk tanúi a számítási felhő térhódításának. A felhő segítségével percek alatt, olcsón, rugalmasan juthatunk számítási kapacitáshoz, tárhelyhez vagy épp adatbázishoz.

Fontos tehát, hogy Windows Phone alkalmazásunk kommunikálhasson külső szerverekkel. A platform ehhez gazdag támogatást nyújt. A fejezetben ennek különféle lehetőségeit ismerjük meg.

## A kommunikáció módjai

A legfontosabb kommunikációs módszerek egy alkalmazás fejlesztése során:

- **Webszolgáltatások:** A webfejlesztés világában régóta bevett technológia. A webszolgáltatások segítségével szerverünk kiajánlhat egy műveletet, amelynek megadja a nevét, paramétereit és visszatérési értékét (pl. `string GetName(int userId)`). Egy sor szabványos technológia segítségével kliensünk (ez esetben a telefon) ezt a metódust ugyanúgy hívhatja meg, mintha egy helyben, a telefonalkalmazásunkban lévő műveletről lenne szó. Így a webszolgáltatásokat könnyen alkalmazhatjuk, és a távoli szervert alkalmazásunk szerves részeként használhatjuk.
- **Webes tartalom közvetlen elérése:** Az internet tartalmának túlnyomó része HTML, és fájlok képeben érhető el. Remek példa erre az RSS feed. Ez egy széles körben elterjedt formátum, amivel időben gyorsan változó híreket írhatunk le, így az adott hírforrást szabványos módon olvashatjuk – miközben a hírportál honlapja természetesen lehet bármilyen egyedi. Windows Phone alkalmazásaink bármilyen internetes tartalmat letölthetnek, feldolgozhatnak, legyen szó RSS feedekről, HTML oldalakról vagy bármilyen más fájlról.
- **Adatbázis-elérés:** a Windows Phone 7.5 (Mango) már tartalmaz közvetlenül a telefonon futó SQL adatbázist. Ennek kapacitása természetesen szűkös, és nem alkalmas egy komolyabb alkalmazás (mondjuk egy raktárkezelő) központi adatbázisának, hiszen a többi telefonon futó alkalmazáspéldány nem fér hozzá. Ilyen esetekben egy központi szerveren futó adatbázist célszerű igénybe venni. Ennek elérését megoldhatnánk egy sor egyénileg fejlesztett webszolgáltatással (írás, olvasás, törlés minden egyes táblára), de ezzel nem kell fáradnunk. Az OData szabvány segítségével rengeteg adatforrást tudunk manipulálni a fejlesztő számára egyszerű módon. Az SQL Server, a SharePoint, valamint számos további Microsoft és külső fejlesztők által készített program és webes alkalmazás is használható OData kiszolgálóként. Ezek mindegyikéhez csatlakozhatunk telefonalkalmazásainkból.
- **Felhasználó-hitelesítés:** Ha internetre csatlakozó alkalmazást írunk, akkor óhatatlanul felmerül a felhasználó bejelentkeztetésének kérdése. Webszolgáltatások segítségével természetesen írhatunk saját hitelesítési mechanizmust, de ez nekünk és felhasználóinknak is kényelmetlen. Nekünk azért, mert sokat kell kódolnunk, és utána biztonsági szempontból

kritikus infrastruktúrát kell üzemeltetnünk. A felhasználóinknak pedig azért, mert még egy név/jelszó párost kell megjegyezniük. Ennél van jobb megoldás is: a Windows Azure egyik szolgáltatását kihasználva például felhasználóink meglévő Live ID, Google, Facebook vagy Yahoo fiókjukkal jelentkezhetnek be alkalmazásunkba.

- **Adattárolás:** Az OData segítségével könnyedén tárolhatunk adatsorokat, de a rekordok mellett gyakran fájlokat is el kell mentenünk. Ilyen lehet például a felhasználó által feltöltött fényképalbum, videó vagy dokumentum. Saját szerverrel és webszolgáltatással természetesen ez is megoldható, de a platform erre is nyújt kész megoldást. Egy osztálykönyvtár segítségével közvetlenül a telefonról írhatjuk és olvashatjuk a Windows Azure tárhelyszolgáltatását.

A fentiek segítségével szinte bármilyen kommunikációs igénynek meg tudunk felelni. A fejezet során a fentiek közül mindegyiket részletesen megismerjük.

A Windows Phone platform ezenkívül néhány egyedi telefon-szerver kommunikációs funkciót is biztosít. Ezekről a speciális funkciókról a könyv további fejezeteiben lehet olvasni:

- **Maps:** Telefonalkalmazásunkba beépíthetünk térkép-vezérlőt. Természetesen nem tárolódik a teljes világtérkép a telefon memóriájában, az épp megnézett terület a Bing Maps szolgáltatásból töltődik le. A Maps vezérlőelemről és képességeiről a 4. fejezetben van bővebben szó.
- **Push Notifications:** A Windows Phone platform kulcsfontosságú elemét alkotják a telefon kezdőképernyőjén látható lapkák, melyek szerveroldaltól is frissíthetők anélkül, hogy alkalmazásunknak futnia kellene. Ezekről a 10. fejezetben található részletes leírás.

## Webszolgáltatások használata

### A webszolgáltatások működése

Amint a bevezetőben már volt róla szó, egy webszolgáltatás nem más, mint egy szerveren lévő metódus, amit az interneten keresztül is meghívhatunk. Ennek működéséhez persze egy sor technológia kell. Ezek fordítják le az internet felől beérkező kérést a metódust futtató programnyelv számára értelmezhető formára, és ezek küldik aztán vissza a metódus futási eredményét a kérőnek.

A .NET-világban erre a Windows Communication Foundation-t (WCF) használjuk, de minden más szerveroldali technológia is ad lehetőséget webszolgáltatások készítésére. A fejlesztés során a programozó először meghatározza, hogy milyen metódusokat szeretne kiajánlani, majd elkészíti a szolgáltatást. Ilyenkor többnyire (a használt technológiától függően) egy XML formátumú leíró fájl jön létre. Az ebben közzétett információ tartalmazza a metódus visszatérési értékét, nevét és elvárt paramétereit (a bevezetőben látott példa: `string GetName(int userId)`).

Ezután a kliensoldalon az ottani fejlesztőkörnyezet számára megadjuk ezt a leíró fájlt. A fejlesztőkörnyezet ebből egy úgynevezett *proxy osztályt* generál. Ez az osztály már a kliensoldalon használt programnyelvben (pl. C#-ban) készül, és tartalmazza a leírófájlban található összes metódust. A kliensoldali programkódunkból ezeket a metódusokat tudjuk használni. Ha valamelyiket meghívjuk, akkor a proxy osztály generált kódja elküldi kérésünket az interneten keresztül, és befogadja a szervertől kapott választ.

Mindez azt jelenti, hogy a webszolgáltatások komplex internetes kódjával egyáltalán nem kell bajlódunk – az oda-vissza fordítást (az angol *plumbing* szó magyar fordításával „vízvezeték-szerelést”) elvégzi helyettünk a fejlesztőkörnyezet.

Amint láttuk, a webszolgáltatás meghívásakor a háttérben valójában két művelet történik: a proxy osztály egyrészt elküld egy kérést, másrészt megvárja és lefordítja a választ. Kódunk szemszögéből mindez tűnhet *szinkron* vagy *aszinkron* műveletnek.

Ha a proxy osztály szinkron működésű, akkor a két háttérműveletet egy egységként kezeli, azaz miután meghívtuk a proxy osztály valamelyik metódusát, kódunk addig nem is halad tovább, amíg a háttérben történő műveletek be nem fejeződtek. Ez programozói szempontból egyszerű, ám a felhasználói élményt

ronthatja. A webszolgáltatás meghívása és válasza között (akár a szerver terheltsége, akár az internetkapcsolat lassúsága miatt) néha másodpercek is eltelhetnek, és az alkalmazásunk ezalatt „lefagyottnak” tűnhet.

Ezt elkerülendő a Windows Phone által generált proxy osztályok csak aszinkron működésűek lehetnek, azaz a kérés elküldését és a válasz megérkezését különválasztják. Amikor meghívjuk a proxy valamelyik metódusát, fel kell iratkoznunk a válaszra. A válasz megérkezésekor az általunk megadott eseménykezelő fut majd le, és így tudjuk folytatni programunk működését. Mivel kódunkat emiatt szét kell darabolnunk, fejlesztői szempontból ez kicsit kényelmetlenebb, de a felhasználói élménynek nagyon jót tesz.

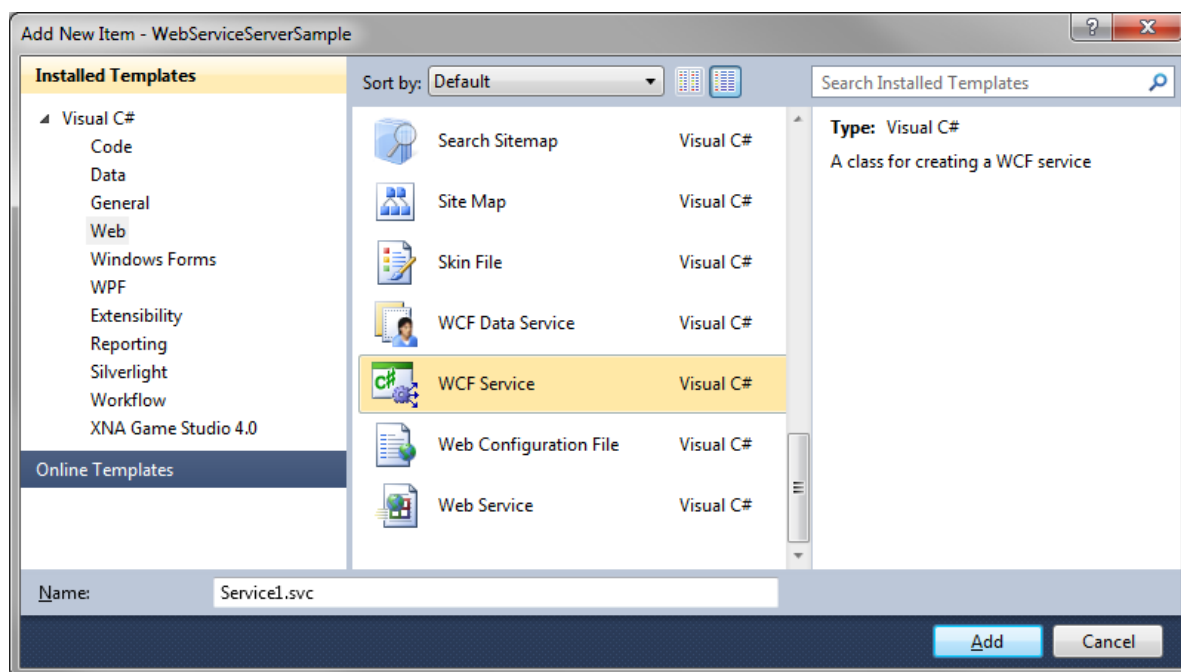
Jó hír, hogy a C# nyelv 5-ös, ezen sorok írásakor fejlesztés alatt álló verziójában szerepelni fog az **async** kulcsszó, ami nagyon megkönnyíti majd az aszinkron műveletekkel való programozást.

## Webszolgáltatás egy mintaalkalmazásban

Írjunk egy Windows Phone alkalmazást, ami egy webszolgáltatást hív meg!

Kezdjük a webszolgáltatással! Ehhez kell majd egy webfejlesztésre alkalmas Visual Studio verzió. Ha a Visual C# Express van a gépünkön a Windows Phone fejlesztéshez, akkor szükségünk lesz az ingyenes Visual Web Developer Express változatra is. Ha viszont a Visual Studio valamelyik fizetős verzióját használjuk, akkor az alkalmas lesz mindkét típusú fejlesztésre.

Hozzunk létre egy ASP.NET Empty Web Application típusú új projektet, és nevezzük ezt el **WebServiceServerSample**-nek! A projekt létrejöttékor üres. Hozzunk létre benne egy új webszolgáltatást! Ehhez vegyünk fel a projektbe egy WCF Service típusú elemet **SampleService** néven! A listában (lásd 9-1 ábra) láthatunk egy Web Service-nek keresztelt elemet is. Ez egy ASMX kiterjesztésű fájlt hoz létre, és ma már egy előző generációs technológiát képvisel. Mi a WCF-et használjuk majd, ami az aktuális webszolgáltatás-fejlesztő eszköz a .NET Frameworkön belül.



9-1 ábra: A webszolgáltatás hozzáadása

A szolgáltatás hozzáadása után három elem is létrejön projektünkben: az **ISampleService.cs** kódfájl, a **SampleService.svc** leírófájl és az ehhez tartozó **SampleService.svc.cs** kódfájl. Mindhárom elemnek fontos szerepe van a webszolgáltatás létrehozásában. Az **ISampleService.cs** állományban kell definiálnunk a kívánt metódusokat; ez csak egy interfészt ad majd, a tényleges kód nem ide kerül. A **SampleService.svc** jelenleg egy sorból áll, de futásidőben ezt a fájlt kell majd meghívni kívülről

a szolgáltatás eléréséhez, mert ezen a néven generálja majd a futtatókörnyezet a korábban már tárgyalt leírófájlt. Végül a **SampleService.svc.cs**-be kerül az **ISampleService.cs**-ben deklarált metódusok tényleges kódja. (Az **ISampleService.cs** nem létfontosságú, e fájl nélkül is működhet a szolgáltatás, de itt nem tárgyalt WCF konfigurációs okok miatt célszerű használni.)

Egy egyszerű fordítási szolgáltatást készítünk: a telefonalkalmazás felküld egy szót, mi pedig visszaküldjük azt angolul. Bemutató alkalmazásról lévén szó, „szótárunk” mindössze két szót tartalmaz majd. Az ehhez szükséges metódus egy **string** változót vár, és egy **string** változót is ad vissza. Ahhoz, hogy megírassuk, először deklarálnunk kell az **ISampleService.cs** interfészben.

A fájlt megnyitva láthatjuk, hogy egy **void DoWork()** nevű metódus már létezik, ami egy **[OperationContract]** attribútummal van ellátva. Az attribútum nagyon fontos, ez tudatja a futtatókörnyezettel, hogy a metódust meg szabad hívni az internet felől is (azaz ki kell kerülnie a leírófájlba). Nevezzük át ezt a metódust az általunk kívántra!

```
[OperationContract]
string Translate(string input);
```

Most írjuk meg a metódushoz tartozó kódot! Ehhez nyissuk meg a **SampleService.svc.cs** fájlt! Itt szintén látjuk majd a **DoWork()** metódust (ezúttal már attribútum nélkül). Töröljük ki, és helyettesítsük az alábbi kóddal!

```
public string Translate(string input)
{
    if (input == "alma")
    {
        return "apple";
    }
    else if (input == "szilva")
    {
        return "plum";
    }
    else
    {
        return "(ismeretlen)";
    }
}
```

Ezzel a szerveroldallal gyakorlatilag el is készültünk. Teszteljük működését! Az F5-tel indítsuk el alkalmazásunkat! Ennek hatására elindul egy helyi gépünkön futó, egyszerű, kifejezetten fejlesztésre való webservert. Beállítástól függően ez vagy az ASP.NET Development Server (Cassini) lesz, vagy az IIS Express. Néhány másodpercen belül megnyílik az alapértelmezett böngészőnk, és a 9-2 ábrához hasonló weboldal tárul elénk. Ez igazolja, hogy webszolgáltatásunk sikeresen lefordult, meghívható.





**9-2 ábra: A fejlesztői szerverből kiszolgált webszolgáltatás**

Ha egy üres oldalt vagy valamilyen hibaüzenetet látunk, akkor ellenőrizzük az URL-t! A **localhost** nevet követő portszám véletlenszerűen generált, és minden gépen más lesz. A **/SampleService.svc** elérési útnak viszont szerepelnie kell; ha nincs ott, kézzel írjuk be!

Az URL-t jegyezzük meg, később még szükség lesz rá!

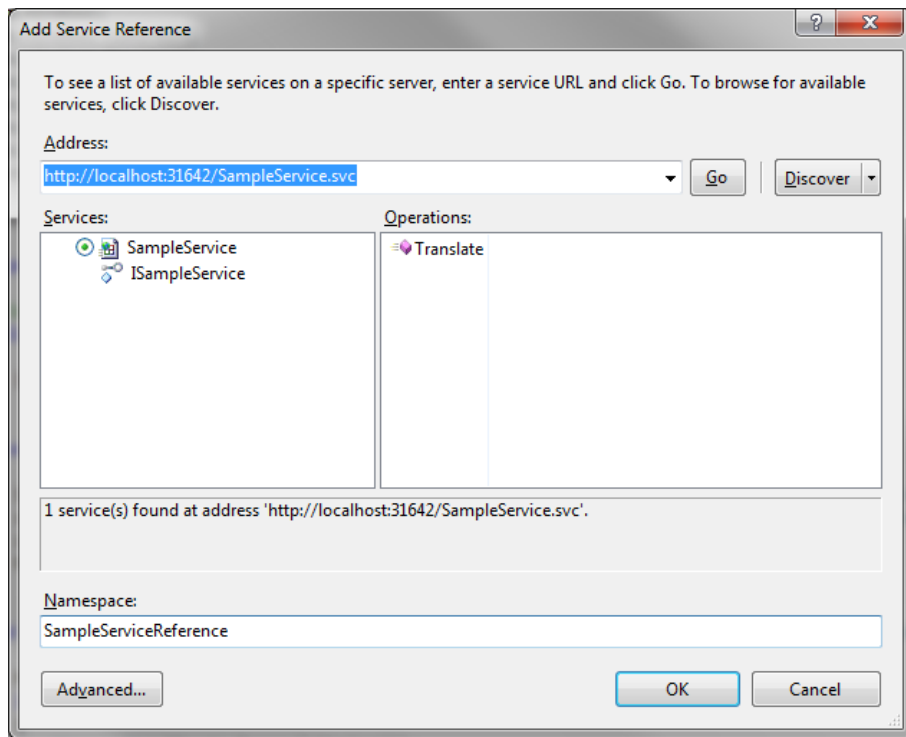
A fent leírtak gyakorlatilag változtatás nélkül működnek akkor is, ha a szerveroldalon Windows Azure-t használunk.

Most térjünk át a kliensoldalra, és írjuk meg a szerveret meghívó Windows Phone alkalmazást!

Ehhez nyissunk egy új Visual Studio projektet, amely legyen Windows Phone Application típusú, neve pedig **WebServiceClientSample**! A projekt létrehozását követően a Visual Studio megkérdezi majd, hogy a telefon melyik verziójához készítjük a projektet. Válasszuk a Windows Phone OS 7.1 (vagy valamelyik újabb) változatot!

Ahhoz, hogy használhassuk a webszolgáltatást, fel kell rá vennünk egy referenciát. E művelet során olvassa majd be a Visual Studio a szerverünk által publikált leíró állományt, és készíti el a bevezetőben említett proxy osztályt. Referencia felvételéhez kattintsunk jobb gombbal a projekt nevére (**WebServiceClientSample**) a Solution Explorer ablakban, és válasszuk az **Add Service Reference** menüpontot!

A megjelenő párbeszédpanel tetejére írjuk be a korábban kimásolt URL-t, majd kattintsunk a Go gombra! (Fontos, hogy webes projektünknek még futnia kell!) A Visual Studio letölti és feldolgozza a megcélzott webszolgáltatás adatait, majd a párbeszédpanel közepén található területen megjeleníti az általa támogatott metódusokat (lásd 9-3 ábra). Jelenleg egy ilyet tartalmaz a webszolgáltatásunk: az imént megírt **Translate()** metódust. Adjunk egy nevet a referenciának (**SampleServiceReference**), és kattintsunk az OK gombra!



9-3 ábra: A Service Reference-hozzáadó ablak

A Visual Studio legenerálja a proxy osztályt, és tulajdonságait elhelyezi egy új fájlban. Ezt **ServiceReferences.ClientConfig** néven találjuk meg projektünkben. Ha alkalmazásunkat élesben szeretnénk felhasználni, akkor ebben az állományban kell átírni a webszolgáltatás URL-jét a végleges címre.

Hozzuk létre programunk felhasználói felületét! Ehhez helyezzünk ki egy szövegdobozt és egy gombot a tervezőfelületre! A szövegdoboz neve legyen **MessageTextBox**, a gomb neve pedig legyen **TranslateButton**!

Az igazán elegáns megoldás érdekében ezután helyezzünk el egy **ProgressBar**-t is (ez az a vezérlőelem, ami a Windows Phone „mozgó pontjait” generálja, amikor a telefon dolgozik valamin)! A **ProgressBar** nem látható az eszköztáron, ezért kézzel kell hozzáadnunk alkalmazásunk XAML kódjához. Keressük meg a XAML kódban a gombunkat leíró sort (**<Button>**), és ez alá szúrjuk be az alábbi kódot:

```
<ProgressBar Name="MainProgressBar" IsIndeterminate="True"
Visibility="Collapsed"></ProgressBar>
```

Ezzel létrehoztunk egy határozatlan futású **ProgressBar**-t (azaz nem kell megadnunk, hogy épp hány százalékon áll, elég csak ki-be kapcsolnunk), amely **MainProgressBar** névre hallgat és alapértelmezésként nem látható.

Most kattintsunk duplán a gombra, hogy belépjünk a **Click** eseménykezelőjébe! Kódunknak meg kell hívnia a webszolgáltatást, de amint korábban már szó volt róla, a webszolgáltatás aszinkron. Ezért meghívásához először feliratkozunk arra az eseményre, ami a kérés befejezését jelzi majd, aztán ténylegesen elküldjük a kérést. Ezenfelül, hogy a felhasználó felé jelezhessük a háttérben zajló munkát, bekapcsoljuk a **ProgressBar**-t:

```
private void TranslateButton_Click(object sender, RoutedEventArgs e)
{
    SampleServiceReference.SampleServiceClient client =
        new SampleServiceReference.SampleServiceClient();
    client.TranslateCompleted +=
        new EventHandler<SampleServiceReference.TranslateCompletedEventArgs>(
            client_TranslateCompleted);
}
```

```

client.TranslateAsync(MessageTextBox.Text);
MainProgressBar.Visibility = Visibility.Visible;
}

```

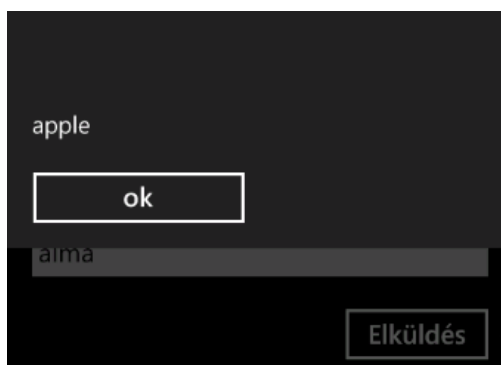
A kérés befejezésekor lefut az eseménykezelő. Ez mindössze annyit tesz, hogy megjeleníti a webszolgáltatástól kapott választ. Mivel a szolgáltatás meghívása az interneten keresztül történik, fel kell készülnünk az esetleges hibákra! Ha hiba történik, akkor nem hagyjuk programunkat lefagyni, hanem kezeljük a hibát, és megjelenítjük azt a felhasználó számára.

```

void client_TranslateCompleted(object sender,
SampleServiceReference.TranslateCompletedEventArgs e)
{
    MainProgressBar.Visibility = Visibility.Collapsed;
    if (e.Error != null)
    {
        MessageBox.Show(e.Error.ToString());
    }
    else
    {
        MessageBox.Show(e.Result);
    }
}

```

Ezzel elkészültünk a mintaalkalmazással. Teszteljük le! Indítsuk be a telefonalkalmazást (és győződjünk meg róla, hogy a webalkalmazás még mindig fut-e)! Írjunk be egy magyar szót (legyen „alma” vagy „szilva”)! Látni fogjuk, hogy alkalmazásunk elindítja a kérést, megjeleníti a **ProgressBar**-t, majd kisvártatva megkapja és megjeleníti a választ (lásd pl 9-4 ábra).



**9-4 ábra: A végrehajtott webszolgáltatás-hívás eredménye**

Ezzel megismertük egy új webszolgáltatás létrehozását. Ha ez később változik, akkor a proxy osztályt is frissítenünk kell. Ehhez kattintsunk jobb gombbal a létrejött referenciára, és válasszuk az Update Service Reference menüpontot!

A webszolgáltatások segítségével lényegében tetszőleges kommunikációt lebonyolíthatunk telefonunk és egy szerver között. Nemcsak a primitív adattípusok, hanem komplex változók is használhatók a kommunikáció során. Ilyen például egy generikus lista (**List<string>**), egy tömb (**int[]**), egy saját magunk által létrehozott adattípus (**Employee**) vagy szinte bármi más. Lényeg, hogy az átküldeni kívánt objektumnak sorosíthatónak kell lennie. A legtöbb .NET adattípus ilyen, a saját osztályainkat pedig mindössze el kell látnunk a **[Serializable]** attribútummal.

Ez a technológia egy óriási témakör, melynek most csak a felszínét karcolgattuk. Az itt leírtak már elegendőek egyszerű kliens-szerver kommunikáció létrehozásához. Aki ennél jobban szeretne elmélyedni a webszolgáltatások használatában, annak gazdag Windows Communication Foundation irodalom áll a rendelkezésére.

## Webes tartalom letöltése

A webszolgáltatások kiváló eszközt nyújtanak a saját szerverünkkel való kommunikációra. Gyakran előfordul azonban, hogy nem a saját szerverünktől szeretnénk lekérni valamit, hanem az interneten keresztül akarunk letölteni egy állományt. Ilyen lehet például egy kép, egy HTML oldal vagy valamilyen fájl. Ezeket az interneten használt HTTP protokoll segítségével tudjuk elérni. Ebben az alfejezetben az erre alkalmas módszereket ismerjük meg.

Alapvetően két, HTTP kommunikációra kitalált osztály létezik a Windows Phone platformban: a **HttpRequest/HttpResponse** páros, illetve a **WebClient** osztály. Ezek az osztályok nem telefon-specifikusak, így egyéb .NET-es tapasztalatainkból már ismerősek lehetnek. A **WebClient** gyakorlatilag nem más, mint egy absztrakciós réteg a **HttpRequest/HttpResponse** felett. Egyszerűbb feladatokra érdemes a **WebClient**-et használni, ha pedig pontosabb irányításra van szükségünk, alkalmazhatjuk a **HttpRequest/HttpResponse** osztályokat.

Kezdjük működésük megismerését az alacsonyabb szintű **HttpRequest/HttpResponse** párossal! Ezek segítségével az **example.com** weboldalt fogjuk majd letölteni és megjeleníteni. Az előző alfejezethez hasonlóan most is egy példaalkalmazást készítünk, mert talán ez a legjobb módja a használható technikák megismerésének.

Hozzunk létre egy Windows Phone Application típusú projektet **HttpSample** néven, Windows Phone OS 7.1 vagy újabb verzióval! Vegyünk fel a felületére egy **HttpButton** nevű gombot, „HttpRequest” felirattal, majd dupla kattintással lépünk be a **Click** eseménykezelőbe!

Ahogy a webszolgáltatásoknál, itt is kizárólag aszinkron metódusokat használhatunk. Ennek oka is hasonló: egy letöltés több másodpercet is igénybe vehet, az alkalmazásnak ez idő alatt nem szabad lefagyottnak tünnie.

A konkrét kód azonban kissé másképpen néz ki, mint a webszolgáltatások esetében. Ott feliratkoztunk a művelet befejezésének eseményére, amit egy eseménykezelővel kaptunk el. A **HttpRequest**-nél ezzel szemben először létrehozunk egy **HttpRequest** típusú osztályt, majd egy metódushívással közöljük, hogy szükségünk lenne a HTTP válaszra. Ezen metódushívásban átadunk egy úgynevezett *callback* függvényt – ezt hívja majd meg a futtatókörnyezet, ha a válasz beérkezett. Emellett egy másik paraméterben egy tetszőleges objektumot adhatunk meg, ezt az objektumot a meghívott *callback* függvény is megkapja, vagyis azt bármilyen állapot-információ átadására is használhatjuk. A létrehozott **HttpRequest** példánynak mindenképpen szerepelnie kell benne, mert ettől a példánytól tudjuk majd elkérni a beérkezett HTTP választ.

Írjuk meg a gomb **Click** eseménykezelőjét az alábbiaknak megfelelően:

```
private void HttpButton_Click(object sender, RoutedEventArgs e)
{
    HttpRequest request = (HttpRequest)HttpRequest.Create("http://www.example.com");
    request.BeginGetResponse(RequestCallback, request);
}
```

Most készítsük el a **RequestCallback** nevű metódust, amire hivatkoztunk! A logika kissé nyakatekert. Először is „kicsomagoljuk” a **HttpRequest** példányt, amit átadtunk a *callback* metódusnak. Utána ezen a példányon meghívjuk az **EndGetResponse()** metódust, amely egy **HttpResponse** példányt hoz létre. Ez tartalmazza a beérkezett HTTP választ. Azonban a **HttpResponse** példányban nem **string**-ként szerepel a megkapott weboldal, hanem egy **Stream**-ként, így ezt az adatfolyamot még ki kell olvasnunk egy **string** változóba.

Miután ezzel elkészültünk, a megkapott válasz első 50 karakterét kiíratjuk a felhasználó számára egy üzenetdobozban.

A **Stream** változó **string** változóba olvasásához szükségünk lesz a **System.IO** névtérre, ezért helyezzük az alábbi sort a kódfájl legtetetejére:

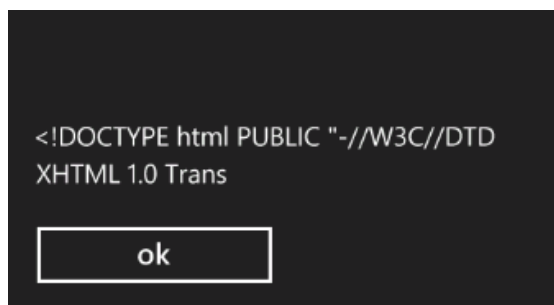
```
using System.IO;
```

Az alábbiakat pedig gépeljük be a **Click** eseménykezelő alá:

```
private void RequestCallback(IAsyncResult ar)
{
    HttpWebRequest request = (HttpWebRequest)ar.AsyncState;
    HttpWebResponse response = (HttpWebResponse)request.EndGetResponse(ar);
    string responseString = new StreamReader(response.GetResponseStream()).ReadToEnd();

    Dispatcher.BeginInvoke(() =>
    {
        MessageBox.Show(responseString.Substring(0, 50));
    });
}
```

Elkészültünk. Teszteljük alkalmazásunkat! Egyszerűen indítsuk el az F5 gomb segítségével, majd kattintsunk a „HttpWebRequest” gombra! Néhány másodperc elteltével beérkezik a HTTP válasz, programunk ezt beolvassa memóriába, majd kiírja az első 50 karakterét (9-5 ábra).



**9-5 ábra: A beérkezett HTTP válasz első 50 karaktere**

A felhasználó nem értesült arról, hogy a program épp dolgozik. Éles alkalmazásban célszerű lenne egy **ProgressBar** vezérlőelem segítségével tájékoztatni erről. Ennek használatát az előző fejezet részben láthattuk, most az egyszerűség kedvéért nem alkalmaztuk.

Amint láthattuk, a **HttpWebRequest/HttpWebResponse** osztályok használata meglehetősen nyakatekert, de segítségükkel alacsony szinten hozzáférünk a kommunikáció részleteihez. A **WebClient** osztály alkalmazása ennél lényegesen egyszerűbb. Lássuk, hogy működik!

Bővítsük ki alkalmazásunkat egy újabb gombbal! Ennek neve legyen **WebClientButton**, felirata pedig „WebClient”. Dupla kattintással nyissuk meg **Click** eseménykezelőjét! A **WebClient** osztály használata szintén aszinkron történik, de ennek modellje megegyezik a webszolgáltatásoknál már látottal. Egyszerűen feliratkozunk a kérés befejeződését jelző eseményre, elindítjuk a kérést, majd az eseménykezelő paraméterei között készen megkapjuk a letöltött HTML oldalt egy **string** változó formájában. Ehhez az alábbi kódra van szükségünk:

```
private void WebClientButton_Click(object sender, RoutedEventArgs e)
{
    WebClient client = new WebClient();
    client.DownloadStringCompleted += new
DownloadStringCompletedEventHandler(client_DownloadStringCompleted);
    client.DownloadStringAsync(new Uri("http://www.example.com"));
}

void client_DownloadStringCompleted(object sender, DownloadStringCompletedEventArgs e)
{
    Dispatcher.BeginInvoke(() =>
    {
```

```
        MessageBox.Show(e.Result.Substring(0, 50));  
    });  
}
```

Teszteljük ezt is! Ismét indítsuk be az alkalmazást, kattintsunk a WebClient gombra! Néhány másodperc elteltével megjelenik a válasz első 50 karaktere.

A fenti két példában szöveges információt töltöttünk le a webről. Ha egy RSS feed-et vagy egy HTML oldalt szeretnénk lekérni telefonalkalmazásunkból, ezt a módszert kell használnunk. Ha egy képet vagy valamilyen más fájlformátumot szeretnénk letölteni, azt is megtehetjük, de kicsit máshogy kell kiadnunk a kéréseket. A **HttpRequest** már eleve egy **Stream**-et ad vissza; ezt tetszésünk szerinti objektumra konvertálhatjuk a memóriában, vagy akár el is menthetjük a telefon Isolated Storage tárhelyére. A **WebClient**-nél pedig ne a **DownloadStringAsync()** metódust használjuk, hanem az **OpenReadAsync()** metódust, amellyel szintén egy **Stream**-et kapunk vissza!

Ezzel megismertük az internetről való letöltés módszereit. Az egyszerűbb, magától értetődő **WebClient** használata az esetek többségében megfelelő. Ha valamiért finoman hangolni szeretnénk a kimenő HTTP kéréseket (például szükségünk van a HTTP fejlécekre is), akkor alkalmazzuk a **HttpRequest/HttpResponse** párost!

## Adatelérés az OData protokollon keresztül

Az interneten vagy vállalatoknál számos olyan szoftver található, amittől adatrekordokat kérhetünk le. Az adatbázis-motoroktól például a bennük tárolt táblák sorait, a vállalati SharePoint-tól a különféle dokumentumokat, az eBay.com-ról a megvásárolható termékek katalógusát, a Netflix online filmnéző portálról a filmek listáját tölthetjük le, és így tovább.

Legyen szó bármilyen adatforrásról, az adatokon alapvetően négy műveletet tudunk végrehajtani: létrehozás (*Create*), olvasás (*Read*), frissítés (*Update*), törlés (*Delete*). Ezeket hívják *CRUD műveletek*nek.

Hiába adott azonban az elvégezhető műveletek köre, sokáig más és más módon tudtunk csak hozzáférni az egyes adatforrásokhoz alkalmazásainkból. Ha a helyi SQL Serverünket akartuk megcímezni, akkor ahhoz ADO.NET-et használtunk, a SharePoint-hoz különféle osztálykönyvtárakkal fértünk hozzá, míg ha az eBay katalógusra voltunk kíváncsiak, akkor valamilyen RSS feed-et próbáltunk letölteni.

Az OData protokoll arra a gondolatra épül, hogy legyen szó bármilyen adatforrásról, nem változnak a rajta elvégezhető műveletek. Az OData definiál egy interfészt, amit az adatforrások gazdái implementálhatnak. Aki ezt megteszi, annak adatait bármilyen platformról egységes módon lehet olvasni. Így a fejlesztőknek nem kell más és más technológiával kísérletezniük, ha adatokat szeretnének lekérdezni – legyen szó akár adatbázisról, akár internetes boltról, akár egy vállalati projektről, az adatelérés ugyanazon a szabványos módon működhet.

Az OData előírásai szerint az adatforrásnak HTTP kéréseket kell fogadnia, ahol az URL tartalmazza az elvégezni kívánt műveletet. Ha az OData szolgáltatás címe például **http://localhost:8080/owind.svc**, akkor az alábbi kéréssel kérdezhetjük le tőle a **Categories** nevű gyűjtemény tartalmát:

```
http://localhost:8080/owind.svc/Categories
```

Fontos észrevenni, hogy a gyűjtemény teljesen általános fogalom; a **Categories** gyűjtemény fizikailag bármi lehet. Állhat mögötte egy SQL tábla, egy XML fájl vagy akár egy szöveges fájl. Ez az információt fogyasztók számára teljesen lényegtelen, ugyanis a válasz szabványos XML-ben érkezik majd:

```
<feed xml:base="http://localhost:8080/owind.svc/" ...>  
  ...  
  <entry>  
    ...  
    <link rel="edit" title="Category" href="Categories(1)" />  
    ...  
    <content type="application/xml">
```

```
<m:properties>
  <d:CategoryID m:type="Edm.Int32">1</d:CategoryID>
  <d:CategoryName>Beverages</d:CategoryName>
  <d:Description>Soft drinks, coffees, teas, beers, and ales</d:Description>
  <d:Picture m:type="Edm.Binary">FRWvAAI...</d:Picture>
</m:properties>
</content>
</entry>
...
</feed>
```

Minden CRUD művelethez definiáltak a fentihez hasonló, az URL részeként kiadható parancsokat. Az OData szolgáltatónak ezeket kell megvalósítaniuk, az OData fogyasztók pedig ezeket használhatják.

Amint láthattuk, az OData adatforrásokat egyszerű HTTP kérésekkel tudjuk lekérdezni. Ez azt jelenti, hogy akár egy böngésző segítségével, akár az előző fejezet részben megismert **HttpRequest** osztállyal kiadhatunk nyers HTTP kéréseket, melyekre az OData adatforrás válaszolni fog. (És ez adja egyben az OData platformfüggetlenségét – HTTP kéréseket gyakorlatilag bármilyen platformról indíthatunk, legyen az Windows, Linux, Mac vagy akár milyen egyéb eszköz.) Ez azonban kényelmetlen. Így számos platformra, többek között a Windows Phone-ra is készült OData kliensoldali osztálykönyvtár. Az osztálykönyvtár segítségével nem kell a nyers HTTP kérésekkel bajlódni; objektumorientált módon, kódunkból tudjuk manipulálni az adatforrást, a rekordokat pedig C# osztályok képében kapjuk meg.

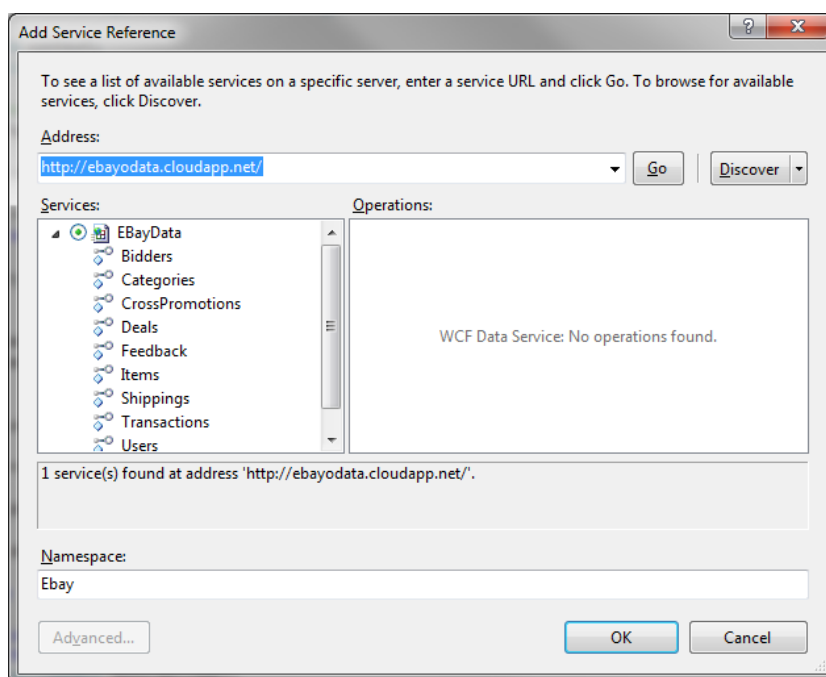
Az OData megismeréséhez egy mintaalkalmazást készítünk, amely az OData osztálykönyvtár igénybevitelével beolvassa az eBay.com aktuális akciót.

Ahogy a fejezetben már többször, nyissunk egy új Windows Phone Application típusú projektet! Legyen ennek neve **ODataSample**, és támogassa a Windows Phone OS 7.1 vagy újabb verzióját!

Az első lépésünk, hogy felvesszünk egy referenciát az OData végpontra. Ehhez (a korábban már látott webszolgáltatásokhoz hasonlóan) kattintsunk a projekt nevére a Visual Studióban, és válasszuk az Add Service Reference parancsot! A cím mezőbe írjuk be az eBay OData végpontjának címét:

**http://ebayodata.cloudapp.net/**, a Namespace mezőben pedig nevezzük el a referenciát **Ebay**-nek!

Amint a 9-6 ábrán is látható, a Visual Studio felismeri, hogy ez nem hagyományos webszolgáltatás, hanem adatszolgáltatás (lásd a panel jobb oldala).



9-6 ábra: Referencia felvétele az OData végpontra



Az OK gomb megnyomása után a Visual Studióhoz készült OData könyvtár legenerálja a végpont eléréséhez szükséges osztályokat. Az eBay által kínált minden objektumtípushoz (pl. **Bidders**, **Items**, **Deals**) készül egy-egy osztály, valamint egy **EBayData** nevű kontextusosztály is generálódik. Ez a kontextusosztály rendelkezik majd azokkal a műveletekkel, amelyekkel a kódból megszólíthatjuk az OData szolgáltatást.

Az ismételts kedvéért: ezek a generált objektumok kizárólag a fejlesztő kényelmét szolgálják. Ha akarnánk, nyers HTTP kérésekkel és a visszaérkező XML válaszok feldolgozásával is kommunikálhatnánk a végponttal. A generált osztályok ezeket a feladatokat végzik el helyettünk.

Most készítsük el a szolgáltatás megszólításához szükséges kódot! Ehhez szükségünk lesz egy névtérre, melyet a **MainPage.xaml.cs** kódfájl tetején helyezünk el!

```
using System.Data.Services.Client;
```

A feladat elvégzéséhez két fontos változót kell deklarálnunk. Az **EBayData** típusú változó biztosítja a szolgáltatás megszólításához szükséges metódusokat. A **DataServiceCollection<Ebay.Deal>** nevű változó pedig egy **Deal** típusú objektumokat tartalmazó gyűjteményt képvisel. Ez a gyűjtemény kezdetben üres, programunk futása során ezt fogjuk feltölteni a szerveroldról lekérdezett rekordokkal. A két változót több metódusból is szeretnénk majd használni, így deklarációjukat közvetlenül a **public partial class MainPage : PhoneApplicationPage** sor (és az azt követő nyitó kapcsos zárójel) után helyezzük el:

```
Ebay.EBayData context;  
DataServiceCollection<Ebay.Deal> deals;
```

A következő feladatunk a **deals** gyűjtemény feltöltése az OData végpontról lekérdezett objektumokkal. A feltöltést közvetlenül az alkalmazás indulásakor elkezdjük, így a kódot közvetlenül a **public MainPage()** konstruktorban helyezzük el.

Az OData végpont megszólításához elsőként létrehozunk egy példányt az **EBayData** nevű osztályból. Ennek a példánynak létrejöttkor átadjuk a megcímzendő szolgáltatás URL-jét, majd létrehozunk egy példányt a korábban deklarált **Deal** típusú objektumokat tartalmazó gyűjteményből is. Ennek a példánynak a kontextusra (az **EBayData** típusú változó példányára) adunk át egy referenciát, így tudja majd, hogy honnan kell az adatokat lekérnie.

Ezután létrehozunk egy LINQ lekérdezést, amellyel megadjuk, hogy a **deals** gyűjteménybe mit szeretnénk betölteni a szerveroldról. A LINQ eszközeivel definiálhatunk különféle szűrőfeltételeket, rendezéseket, melyeket az OData könyvtár fordít le az OData által elvárt URL-szintaxisra. Az általunk megadott LINQ lekérdezés nem tartalmaz szűrőfeltételt, az összes **Deal** típusú objektumot lekéri.

A LINQ eszköztára bővebb, mint az OData által támogatott műveletek halmaza, így előfordulhat, hogy egy érvényes LINQ lekérdezés futásidejű hibát dob.

Végül a korábban már látott módon feliratkozunk a lekérdezés befejezését jelző eseményre, majd kiadjuk az utasítást a **deals** gyűjtemény feltöltésére az általunk megadott LINQ lekérdezés szerint.

```
public MainPage()  
{  
    InitializeComponent();  
  
    context = new Ebay.EBayData(new Uri("http://ebayodata.cloudapp.net/"));  
    deals = new DataServiceCollection<Ebay.Deal>(context);  
    var query = from deal in context.Deals  
                select deal;
```



```

deals.LoadCompleted += new EventHandler<LoadCompletedEventArgs>(deals_LoadCompleted);
deals.LoadAsync(query);
}

```

Az OData protokollt felkészítették a nagyméretű adathalmazokra. Így ha a lekérdezés sok eredményt adna vissza, programunknak nem kell megvárnia, amíg a teljes adathalmaz megérkezik (ez sok ezer rekordot is jelenthet, ami több megabájtnyi XML-be férne csak bele). Ehelyett a protokoll ilyenkor az első néhány rekordot adja át, majd a válaszban jelzi, hogy még van további része is az eredményhalmaznak.

A **deals\_LoadCompleted** eseménykezelő meghívásakor lekérdezésünk lefut, és a visszaadott objektumok betöltődnek a **deals** gyűjteménybe. A fent említett mechanizmus miatt azonban előfordulhat, hogy még vannak betöltetlen rekordok az adatforrásban. Valamennyi rekordot szeretnénk lekérdezni, ezért ha kódunk azt érzékeli, hogy a lekérdezés még folytatható, akkor a **LoadNextPartialSetAsync()** hívással az eredményhalmaz következő szeletét is letölti. Ehhez nem kell új eseménykezelőt felvennünk, mert ez a metódus is a **deals\_LoadCompleted** eseménykezelőt hívja meg, így lekérdezésünk addig fut, amíg minden objektum meg nem érkezik a szerverről.

Ha úgy találjuk, hogy minden objektum megérkezett, akkor egy üzenetdobozban kiíratjuk az aktuális eBay akciók címeit. Az esetleges hibát pedig természetesen kezeljük.

Hozzuk létre a **deals\_LoadCompleted** metódust az alábbi kóddal:

```

void deals_LoadCompleted(object sender, LoadCompletedEventArgs e)
{
    if (e.Error != null)
    {
        if (deals.Continuation != null)
        {
            deals.LoadNextPartialSetAsync();
        }
        else
        {
            string message = "";
            foreach (var deal in deals) message += deal.Title + Environment.NewLine;
            MessageBox.Show(message);
        }
    }
    else
    {
        MessageBox.Show(e.Error.ToString());
    }
}

```

Ezzel elkészültünk. Teszteljük alkalmazásunkat! Indítsuk el az F5-tel, és kisvártatva megjelennek az aktuális eBay akciók. Az eBay honlapján ellenőrizhetjük is, hogy a megfelelőket töltöttük-e le. A <http://deals.ebay.com/> honlapot megnyitva láthatjuk, hogy valóban az épp elérhető akciókat kérdeztük le (9-7 ábra).

The screenshot shows a mobile application interface. On the left, a dark grey panel lists five items: 'Black & Decker Cordless Flex Vac w/ Accessories', 'Samsung Nexus S Unlocked GSM Android Smart Phone', 'Kodak EasyShare M22 14MP Digital Camera', 'Traveler's Choice Rome 29in Hardshell Suitcase', and 'Fisher Price TrackMaster Motorized Railway Playset'. Below this list is a white button with the text 'ok'. On the right, a light blue panel displays a 'DEAL BLAST' timer at 02:10:41. Below the timer, five product cards are shown, each with an image, the product name, and the discounted price. The products and their prices are: Black & Decker Cordless Flex Vac w/... (\$45.99, 50% off), Samsung Nexus S Unlocked GSM Android Smart... (\$299.99, 40% off), Kodak EasyShare M22 14MP Digital Camera (\$76.99, 23% off), Traveler's Choice Rome 29in Hardshell... (\$59.99, 80% off), and Fisher Price TrackMaster Motorized Railway... (\$99.99, 17% off).

Product	Price	Discount
Black & Decker Cordless Flex Vac w/ Accessories	\$45.99	50% off
Samsung Nexus S Unlocked GSM Android Smart Phone	\$299.99	40% off
Kodak EasyShare M22 14MP Digital Camera	\$76.99	23% off
Traveler's Choice Rome 29in Hardshell Suitcase	\$59.99	80% off
Fisher Price TrackMaster Motorized Railway Playset	\$99.99	17% off

9-7 ábra: Az OData-n keresztül letöltött objektumok és az eBay honlapja

Ezzel megismertük az OData használatának alapjait. Az itt megtanultak alapján már tudunk egy OData-fogyasztásra képes alkalmazást készíteni. A webszolgáltatásokhoz hasonlóan az OData is rendkívül széles témakör. Aki szeretné jobban megismerni az OData protokollt, az látogasson el a <http://www.odata.org> weboldalra, ahol megtalálható az OData-szolgáltatók és a különféle platformokra elérhető OData kliensek folyamatosan bővülő listája, valamint a protokollal kapcsolatos részletes információk és a fórum. Érdemes megismerkedni a protokollal, mert rengeteg Microsoft-termék és számos külső gyártó már most támogatja! Ilyen például az SQL Azure, a SharePoint, az eBay, a Netflix és még sokan mások.

## Adattárolás Windows Azure-ban

Internetre kapcsolt telefonalkalmazásunknak gyakran szüksége lehet saját fájlok és egyéb adatok tárolására. Ezeket az igényeket természetesen megoldhatjuk egy saját szerverrel, melyre alkalmazásunk egy webszolgáltatáson keresztül tölti fel az anyagokat. Ennél azonban sokkal egyszerűbb és olcsóbb megoldást is alkalmazhatunk: a Windows Azure tárhelyszolgáltatását kihasználva néhány száz forintért tárolhatunk gigabájtokat, ráadásul egyszerű objektumorientált API-n keresztül. Ebben a részben a Windows Azure tárhelyszolgáltatásának elemeit és ezek használatát ismerjük meg – egy Windows Phone alkalmazásból.

Az Azure Storage háromféle szolgáltatást nyújt: tárolhatunk benne fájlokat (Blob Storage), létrehozhatunk benne aszinkron kommunikációra alkalmas várakozási sorokat (Queue Service), illetve feltölthetünk egyszerű táblajellegű információkat, mint pl. egy telefonkönyv (Table Service). Ez utóbbi nem váltja ki az SQL szerveret, amint később azt látni fogjuk.

Mindhárom szolgáltatásra igaz, hogy az Azure a feltöltött adatokat három példányban tárolja, ráadásul két, egymástól több száz kilométerre lévő adatközpontba is szétmásolja, így a véletlen adatvesztés gyakorlatilag kizárt. Az Azure emellett komoly kötelezettségeket vállal a szolgáltatás rendelkezésre állására is. Így elmondhatjuk, hogy mindhárom tárhelyszolgáltatás alkalmas komoly, professzionális alkalmazások kiszolgálására is.

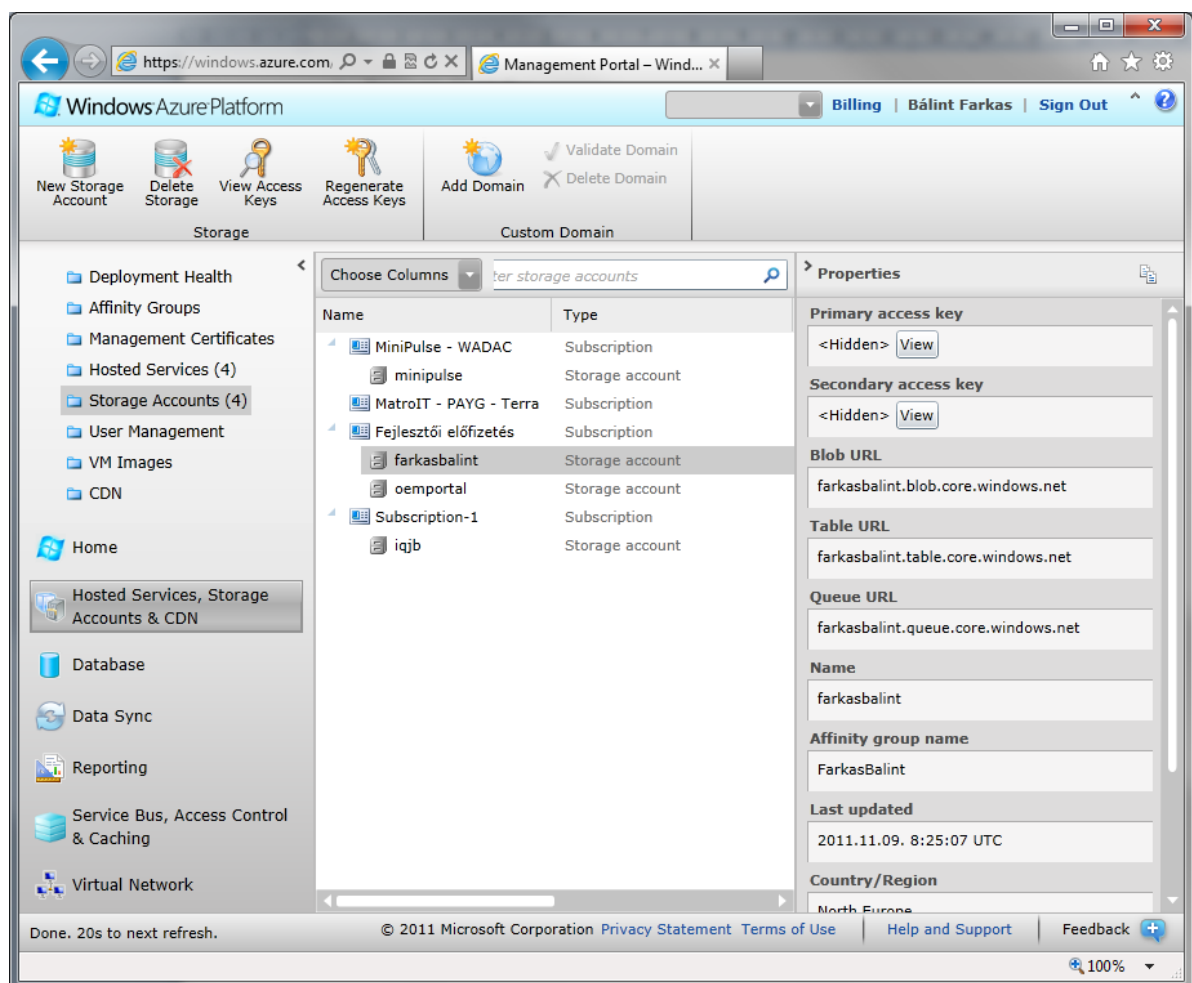
Ismerjük meg ezeket részletesebben!

## Fájlok (blob-ok)

A Blob Storage gyakorlatilag egy fájlserver a felhőben. Azure előfizetésünkhöz úgynevezett konténereket hozhatunk létre. Mindegyik konténerre úgy gondolhatunk, mint egy merevlemez egy partíciójára. A konténerekbe pedig tetszőleges mennyiségű fájlt feltölthetünk, akár mappákba rendezve is. A fájlok méretének felső korlátja egy terabájt (1 TB). A feltöltött fájlokat később URL-jük alapján a telefonról vagy az internetről is elérhetjük.

A fejezetben megszokott módon egy mintaalkalmazáson keresztül ismerjük meg az Azure Storage használatát. Az Azure kínál ugyan egy emulátort, de ez az emulátor csak PC-n futtatható, így telefonalkalmazásunkból egy valódi, a felhőben lévő tárhelyet kell megcímeznünk. Ehhez szükség van egy Azure előfizetésre. Ilyen előfizetéshez a <http://www.microsoft.com/windowsazure/offers/> oldalon juthatunk hozzá – gyakran van ingyenes kipróbálási lehetőség is.

Az előfizetésünkön belül hozzunk létre egy Azure Storage tárhelyet! Ehhez látogassunk el az Azure portálra (<http://windows.azure.com>), majd jelentkezünk be az előfizetéskor megadott Live ID azonosítónkkal! A bal alsó sarokban kattintsunk a Hosted Services, Storage Accounts & CDN fülre, majd válasszuk a Storage Accounts listaelemet (9-8 ábra)! Ekkor megjelennek az előfizetéshez tartozó Azure Storage tárhelyek. A New Storage Account gomb segítségével hozzuk létre tárhelyünket! Ennek során kell megadnunk a tárhely nevét. Miután létrejött, a jobb felső sarokban találjuk a Primary access key sort, és mellette a View gombot. Erre kattintva tekinthetjük meg a tárhelyhez tartozó két automatikusan generált jelszót.



9-8 ábra: Az Azure menedzsment portál, benne tárhelyeinkkel

A tárhely nevét és egyik jelszavát tároljuk el; ezekre lesz majd szükségünk a tárhely programozásához. Most készítsük el mintaalkalmazásunkat!

Az Azure tárhely rendelkezik egy REST-es menedzsment felülettel. Ez azt jelenti, hogy egyszerű HTTP kérésekkel parancsokat tudunk kiadni. Ez azonban kényelmetlen, így készült hozzá egy osztálykönyvtár. Ez az osztálykönyvtár a Windows Azure Toolkit for Windows Phone része. Töltsük ezt le a CodePlex-ről, a <http://watwp.codeplex.com/> URL-en érhető el! Telepítése során a fájlokat egy általunk meghatározott mappába helyezi. Ennek a mappának a helyét szintén jegyezzük meg, mert később szükségünk lesz majd rá!

A szerveroldali programozáshoz készült Windows Azure SDK természetesen szintén tartalmaz kész osztályokat az Azure Storage eléréséhez. Az itt tárgyalt Windows Phone alapú osztályok hasonlítanak ezekre, ám némileg más logikát alkalmaznak.

A Toolkit telepítése után készítsünk egy új Windows Phone Application projektet **AzureStorageSample** néven, Windows Phone OS 7.1 vagy újabb operációs rendszer verziót használva! A Toolkit osztályainak használatához vegyünk fel egy referenciát a **WindowsPhoneCloud.StorageClient.dll** könyvtárra! Ezt a Toolkit mappája alatt a **Binaries** mappában találjuk. A szükséges névterek importálásához helyezzük el a következő sorokat a **MainPage.xaml.cs** kódfájl legtetetejére:

```
using Microsoft.Samples.WindowsPhoneCloud.StorageClient;
using Microsoft.Samples.WindowsPhoneCloud.StorageClient.Credentials;
using System.IO;
using System.Text;
```

Miután ezzel elkészültünk, alkalmazásunk felületére vegyünk fel egy gombot, **BlobButton** néven, „Blob feltöltése” felirattal! Erre a gombra kattintva alkalmazásunk feltölt majd a felhőbe egy példafájlt.

Az Azure osztályok – a fejezet korábbi részeiben tárgyalt okokból – aszinkron működésűek, így a gombra kattintva a kérés csak elindul, befejezni csak egy másik metódusból fogjuk. Ezért a használt változóinkat nem a metóduson belül deklaráljuk, hanem a program törzsében. Helyezzük a következő sorokat a **public partial class MainPage : PhoneApplicationPage** sor (és az azt követő kapcsos zárójel) alá:

```
StorageCredentialsAccountAndKey credentials;
CloudBlobClient blobClient;
CloudBlobContainer container;
```

Az első, **credentials** nevű változóban tároljuk el a tárhely megcímzéséhez használt nevet és jelszót. A két másik változó pedig a Blob storage használatához kell.

Most adjunk értéket a **credentials** változónak! Ezt célszerű a program indulásakor, egyszer megtenni, mert később még többször szükségünk lesz rá. Az értékadáshoz írjuk az alábbiakat a **public MainPage()** konstruktor **InitializeComponent()** sora alá:

```
credentials = new StorageCredentialsAccountAndKey("TÁRHELYNÉV", "JELSZÓ");
```

A **TÁRHELYNÉV** helyére írjuk a tárhely nevét (amit létrehozáskor megadtunk), a **JELSZÓ** helyére pedig a tárhely jelszavát (amit lementettünk az Azure menedzsment portálról).

Amint korábban volt róla szó, a Blob Storage használatának előfeltétele, hogy létrehozzunk benne fájljaink számára egy konténert. Ilyen konténerből tetszőleges számúval rendelkezhetünk, így különböző alkalmazásaink, vagy alkalmazásunkon belül különböző felhasználóink is kaphatnak külön-külön konténert. Alkalmazásunk első lépése, hogy létrehozzon magának egy konténert (ha ezt korábban még nem tette meg).

Kattintsunk duplán a **BlobButton** gombra, hogy belépjünk **Click** eseménykezelőjébe! A korábban említett REST-es API-t egy sor osztály fedi le; nekünk csak ezek metódusait kell hívni, a háttérben zajló HTTP kéréseket az osztályok eltakarják előlünk. Így itt először is példányosítunk egy **CloudBlobContainer** típusú osztályt. Ez szolgál majd a konténerek létrehozására.

A menedzsment API hívásához osztályainknak rendelkezniük kell a szükséges hitelesítési adatokkal. Ezeket az adatokat már eltároltuk a **credentials** nevű osztályba, így a most létrehozott **CloudBlobContainer** példánynak egyszerűen csak ezt kell átadnunk.

A **CloudBlobContainer** példány létrehozása után egy metódus segítségével megindítjuk az alkalmazásunk által használni kívánt konténer létrehozását. A használt metódus intelligens, így megvizsgálja, hogy létezik-e már az adott konténer, és csak akkor próbálkozik meg létrehozásával, ha még nincs ott. Paraméterként át kell adnunk a kívánt konténer URL-jét. Ennek egyrészt tartalmaznia kell tárhelyünk nevét, másrészt egy kötelező elemet (**blob.core.windows.net**), harmadrészt a konténer elvárt nevét. Tárhelyünk nevét helyettesítsük be, a konténer neve pedig legyen **phonetest1**!

```
container = new CloudBlobContainer(credentials);
container.CreateContainerIfNotExist(new
Uri("https://TÁRHELYNÉV.blob.core.windows.net/phonetest1"), CloudBlobContainerCreated);
```

Aszinkron metódust hívtunk, így befejeződése után valahogy folytatnunk kell kódunk futtatását. Ezúttal nem egy eseményre iratkozunk fel, hanem a metódushívásban átadunk paraméterként egy callback függvényt, melyet a metódus majd meghív befejezésekor. Ez a callback függvény a fent látható **CloudBlobContainerCreated**.

Következő feladatunk a callback függvény megírása. A függvényben megvizsgáljuk, hogy történt-e hiba a végrehajtás során. Ha igen, akkor erről értesítjük a felhasználót. Ha nem, akkor folytatjuk feladatunkat: feltöltünk egy blob-ot. Ehhez egy **CloudBlobClient** nevű objektumot használunk. Ennek létrehozásakor megadjuk, hogy az melyik konténerrel dolgozzon (a fenti URL-t kell itt is használnunk), valamint átadjuk neki a hitelesítési információkat. Utána pedig egy metódushívással elvégezzük a feltöltést. A feltöltendő adatot egy **Stream** típusú változóban várja az osztály, ezért létrehozunk egy **MemoryStream**et, amelybe beleírunk néhány betűt. A **Stream** egy általános célú változótípus, így éles alkalmazásban egy képet, a felhasználó által az internetről letöltött állományt vagy bármi mást is fel tudunk tölteni Blob Storage-ba. Írjuk az alábbi kódot a gomb **Click** eseménykezelője alá:

```
private void CloudBlobContainerCreated(CloudOperationResponse<bool> returnValue)
{
    this.Dispatcher.BeginInvoke(() =>
    {
        if (returnValue.Exception == null)
        {
            MemoryStream memoryStream = new MemoryStream();
            StreamWriter streamWriter = new StreamWriter(memoryStream);
            streamWriter.WriteLine("Hello!");
            streamWriter.Flush();
            memoryStream.Seek(0, SeekOrigin.Begin);

            blobClient = new CloudBlobClient(
                "https://TÁRHELYNÉV.blob.core.windows.net/phonetest1", credentials);
            blobClient.Upload("test.txt", memoryStream, BlobUploadCallback);
        }
        else
        {
            MessageBox.Show(returnValue.Exception.ToString());
        }
    });
}
```

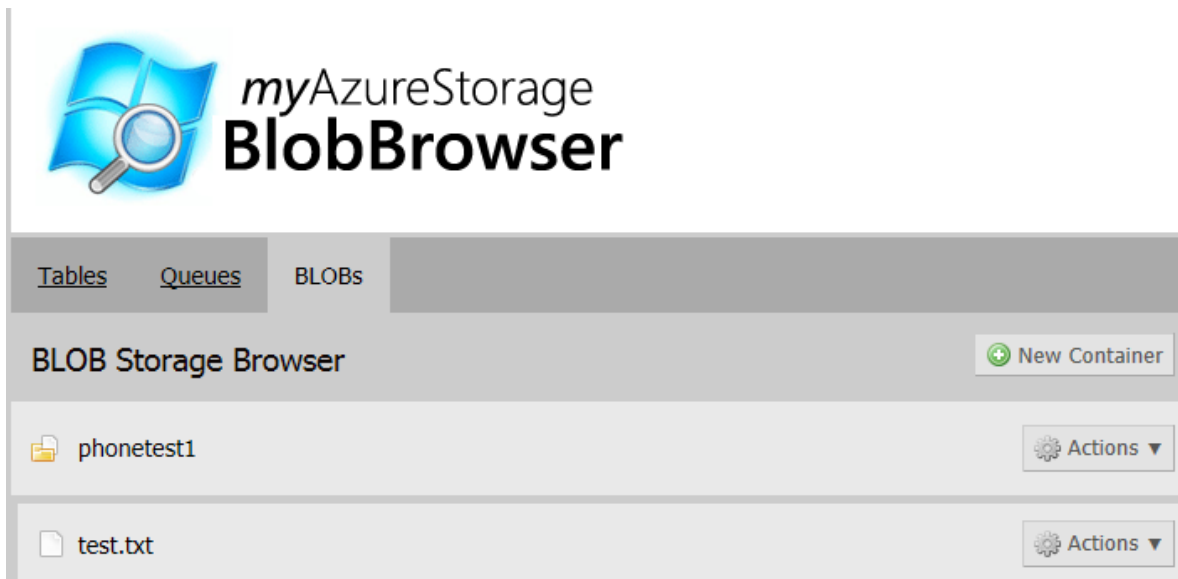
A blob feltöltése is egy aszinkron művelet, így egy második callback függvényt is készítenünk kell. Ennek már mindössze annyi feladata van, hogy tájékoztassa a felhasználót a feltöltés kimenetéről (sikeres vagy sikertelen).

```
private void BlobUploadedCallback(CloudOperationResponse<bool> returnValue)
{
    this.Dispatcher.BeginInvoke(() =>
    {
        if (returnValue.Exception == null)
        {
            MessageBox.Show("Blob feltöltve!");
        }
        else
        {
            MessageBox.Show(returnValue.Exception.ToString());
        }
    });
}
```

Teszteljük alkalmazásunkat! Az F5 gombbal futtassuk, majd nyomjuk meg a „Blob feltöltése” gombot! Alkalmazásunk a háttérben létrehozza a konténert és feltölti a tesztfájlt. Ellenőrizzük ennek megtörténtét! Ehhez valamilyen eszközzel rá kell csatlakoznunk a Blob Storage-ra. Talán a legegyszerűbb a <http://www.myazurestorage.com> honlap használata. Itt meg kell adnunk tárhelyünk nevét és jelszavát, majd egy webes felületen böngészhetjük a Blob Storage tartalmát.

A myazurestorage.com weboldalt a Microsoft fejlesztette, így megbízhatunk benne, nem kell jelszavunk ellopásától tartanunk.

A BLOBs fülre kattintva megtaláljuk a phonetest1 konténert, amit kibontva ellenőrizhetjük, hogy test.txt fájlunk valóban létrejött-e (9-9 ábra).



9-9 ábra: A létrehozott konténer és blob

Ezzel megismertük a Blob Storage használatának alapjait. Természetesen a szolgáltatás képességeinek és a használható API-nak csak egy nagyon kis részét láttuk, de ennek alapján már el tudunk készíteni egyszerű alkalmazásokat. A Blob Storage nyújtotta lehetőségekről és a Windows Phone által támogatott API-ról bővebben olvashatunk az Azure és a Windows Azure Toolkit for Windows Phone honlapjain.

### Várakozási sorok (queue-k)

Következő Storage elemünk a várakozási sor, vagyis a Queue Service. Az Azure várakozási sor kisméretű (maximum 64 kilobájtos) üzenetek megbízható kezelésére alkalmas. Előnye, hogy tetszőleges számú egyszerre működő termelővel és fogyasztóval megbirkózik, kapacitása pedig gyakorlatilag korlátlan.

Egy ügyes mechanizmussal gondoskodik arról, hogy a sorba kerülő üzenetek addig ne törölődjenek, amíg azok sikeresen feldolgozásra nem kerültek. A sorban lévő üzeneteket ugyanis először csak *kivesszük*. A kivett üzenetek bizonyos időre virtuálisan eltűnnek a sorból (alapértelmezésként 10 percre, de ez az időtartam feljebb is vehető). Véglegesen törölni csak egy külön utasítással lehet. Ha ügyelünk arra, hogy az üzeneteket csak sikeres feldolgozásuk után *töröljük*, akkor egy hibára futó feldolgozóegységnél az üzenet nem vész el. Mivel a hibás egység sosem jutott el odáig, hogy törölje az üzenetet, ezért az kis idő elteltével újra megjelenik, és egy másik egység újra megpróbálkozhat a feldolgozásával.

Az üzenetek méretére vonatkozó korlát szándékos: a várakozási sorban csak a feladatok leírását kell tárolnunk (pl. „videót kell konvertálni”). A feladatok végrehajtásához szükséges, nagyméretű fájlokat és egyéb adatokat máshová célszerű helyezni (pl. Blob Storage-ba), ezekre az üzeneten belülről egy URL-lel hivatkozhatunk.

A Queue szolgáltatás felépítése hasonlít a Blob Storage felépítéséhez: Azure tárhelyünk alá tetszőleges számú, egyedileg elnevezett sort hozhatunk létre, és mindegyik sorba külön-külön pakolhatunk üzeneteket.

A Blob Storage-hoz készült példaalkalmazásunkat kiegészítjük egy újabb gombbal, mellyel egy sort hozunk létre, majd elhelyezünk benne egy üzenetet. Ehhez szükségünk lesz néhány osztályszintű változóra. Helyezzük az alábbi sorokat a MainPage osztály tetejére:

```
CloudQueueClient queueClient;
CloudQueue queue;
```

Most hozzuk létre a gombot! Annak neve legyen **QueueButton**, felirata pedig „Queue üzenet létrehozása”. Dupla kattintással lépünk be **Click** eseménykezelőjébe! Első dolgunk a sor létrehozása lesz. Ehhez egy **CloudQueueClient** objektumot készítünk. Ennek átadjuk a Queue szolgáltatás URL-jét, valamint a korábban eltárolt hitelesítési adatainkat. Utána ettől az objektumpéldánytól kérünk egy referenciát egy **phonetest1** nevű sorra, amit egy metódushívással létrehozunk, ha még nem létezik. A meghívott metódus aszinkron, így kódunk egy callback függvényben folytatódik majd.

```
private void QueueButton_Click(object sender, RoutedEventArgs e)
{
    queueClient = new CloudQueueClient("https://TÁRHELYNÉV.queue.core.windows.net",
    credentials);
    queue = (CloudQueue)queueClient.GetQueueReference("phonetest1");
    queue.CreateIfNotExist(CloudQueueCreatedCallback);
}
```

A callback függvényben nincs más dolgunk, mint ellenőrizni, hogy történt-e hiba, és ha nem, akkor beírni egy üzenetet a sorba. Az üzenetek tartalma tetszőleges formátumú lehet, akár szöveges, illetve akár egy bájtömb is. A Windows Phone-hoz készült API csak bájtömb beírását teszi lehetővé, ezért létrehozunk egy szöveget, amint bájtömbbé konvertálva helyezzük el egy üzenetben:

```
private void CloudQueueCreatedCallback(CloudOperationResponse<bool> returnValue)
{
    this.Dispatcher.BeginInvoke(() =>
    {
        if (returnValue.Exception == null)
        {
            CloudQueueMessage message = new CloudQueueMessage()
            { AsBytes = Encoding.UTF8.GetBytes("Hello") };
            queue.AddMessage(message, CloudQueueMessageAddedCallback);
        }
        else
            MessageBox.Show(returnValue.Exception.ToString());
    });
}
```

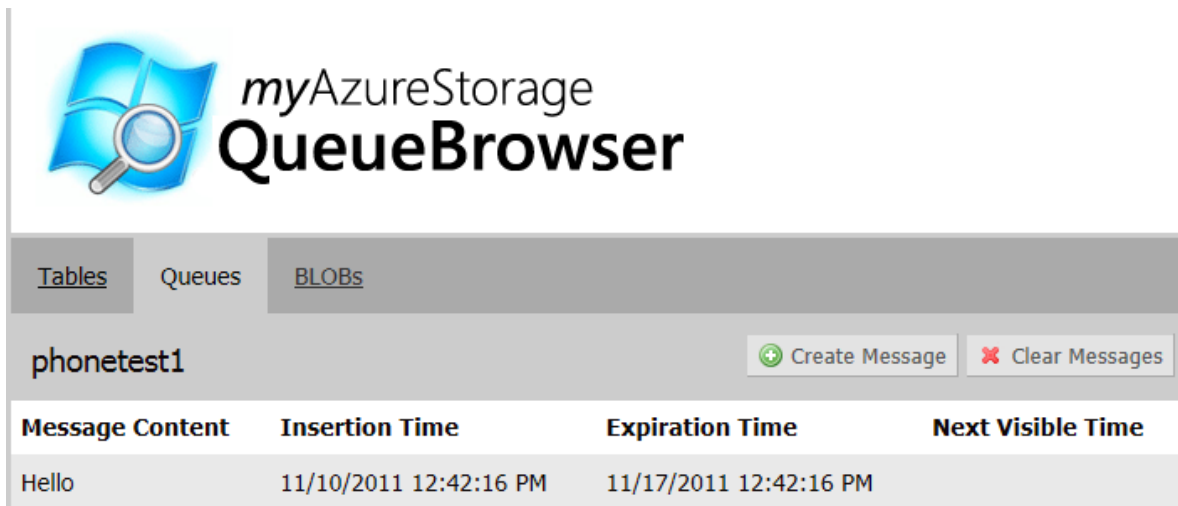


Az üzenet létrehozása természetesen szintén aszinkron, ezért feladatunkat egy újabb callback függvényben fejezzük be. Itt mindössze értesítjük a felhasználót a művelet eredményéről.

```
private void CloudQueueMessageAddedCallback(CloudOperationResponse<bool> returnValue)
{
    this.Dispatcher.BeginInvoke(() =>
    {
        if (returnValue.Exception == null)
        {
            MessageBox.Show("Üzenet létrehozva!");
        }
        else
        {
            MessageBox.Show(returnValue.Exception.ToString());
        }
    });
}
```

Teszteljük alkalmazásunkat: indítsuk el, és nyomjuk meg a **QueueButton** gombot! A Blob Storage példához hasonlóan most is a **myazurestorage.com** segítségével ellenőrizhetjük, hogy üzenetünk valóban létrejött-e. Ismét látogassunk el az oldalra! A Queues fülön megtaláljuk a **phonetest1** várakozási sort és benne az imént létrehozott üzenetünket (9-10 ábra).

Ne lepődjünk meg, ha az üzenet időbélyegzője néhány órával korábbi vagy későbbi, mint a magyar idő – a weboldal az Azure adatközpont helyi idejét jeleníti meg.



Message Content	Insertion Time	Expiration Time	Next Visible Time
Hello	11/10/2011 12:42:16 PM	11/17/2011 12:42:16 PM	

9-10 ábra: A létrehozott várakozási sor és üzenet

### Táblák (table-k)

Az Azure Storage harmadik eleme az Azure Table Service. Itt strukturált, táblaszerű információt, vagyis több mezőből álló adatsorokat tárolhatunk. Tárhelyünk alatt tetszőleges számú, egyedileg elnevezett táblát hozhatunk létre. A táblákhoz nem kell sémát rendelnünk – azaz nem kell őket megterveznünk. Objektumokat helyezhetünk beléjük, és minden tábla felveszi a belerakott objektumok sémáját. Vagyis ha egy **Dog** típusú objektumot helyezünk egy táblába, és a **Dog** objektumnak **Name** és **Age** tulajdonsága van, akkor a táblában automatikusan létrejön egy **Name** és egy **Age** típusú oszlop. Egy táblába többféle objektumot is beletehetünk, de a táblában minden objektum minden tulajdonságához létrejön egy-egy oszlop, ami érdekes ütközéseket és hibajelenségeket okozhat, így ezt inkább kerüljük!

A táblába helyezett objektumoknak tartalmazniuk kell egy **PartitionKey** és egy **RowKey** nevű szöveges mezőt és egy **Timestamp** nevű, dátum típusú mezőt. Ezeket nem kell kézzel beraknunk azokba az



osztályokba, amiket táblában szeretnénk tárolni – elég leszármaztatni osztályainkat a **TableServiceEntity** ősosztályból. A táblába helyezett objektumokat *entitásoknak* nevezzük.

A **Timestamp** mezővel semmi dolgunk, azt az Azure kezeli. A **PartitionKey** és a **RowKey** azonban fontos! A táblákban mindössze ezen a két mezőn van index (a többire nem is rakhatsz). A kettőnek együttesen egyedinek kell lennie. A **PartitionKey** értékét a skálázásban használja az Azure, az azonos **PartitionKey** értékkel rendelkező entitasokat fizikailag egymáshoz közel tárolja, így a logikailag összetartozó adatokat érdemes azonos **PartitionKey** értékkel ellátni. Ha ezzel nem akarunk bajlódni, minden objektumnak lehet különböző **PartitionKey** értéke is. A **RowKey** nem hordoz ilyen jelentést.

Egy népszámlálás adatsorainál a **PartitionKey** érték lehet például az illető városa, a **RowKey** értéke pedig az illető személyi száma. Így a kettő együtt garantáltan egyedi (sőt a **RowKey** akár önmagában is), és ha város alapján keresünk vissza entitasokat, akkor nagyon gyorsan tudunk haladni.

A táblába rakott entitasoknak a három kötelező mezőn kívül maximum 252 általunk megadott mezője lehet. Ezek típusai alapvetően a .NET alaptípusok lehetnek (pl. **Int32**, **Double**, **Boolean**, **Guid** vagy **String**). Komplex vagy saját típusokat (azaz listákat, általunk definiált osztályokat) nem használhatunk, és a .NET alaptípusok közül sem mindet (pl. a **Float** adattípust nem). Egy entitás teljes mérete 1 megabájt lehet.

A fenti korlátozások jelen sorok írásakor aktuálisak voltak, ám az Azure többi szolgáltatásához hasonlóan a Table Service is folyamatosan fejlődik. A jellemzők legfrissebb listája mindig az MSDN-en olvasható.

A Table Service előnye, hogy óriási táblaméreteket támogat, táblánként több millió adatsort is eltárolhatunk. Másik előnye, hogy rendkívül olcsó. A későbbiekben leírt árazásból láthatjuk majd, hogy néhány száz forintból hatalmas adatbázisokat tarthatunk fenn! Hátránya viszont, hogy nem relációs, azaz tábláinkra nem vehetünk fel plusz indexeket, idegen kulcsokat, tárolt eljárásokat, és így tovább, azaz képességei lényegesen szerényebbek, mint egy SQL szerveré. A Table Service és a SQL Azure tehát nem versenytársak, másféle feladatkörökre érdemes alkalmazni őket. Ha programunknak egy High Score listát kell nyilvántartania, vagy éppenséggel sok millió, de egyszerű adatsort kell tárolnia, akkor ezt a feladatot néhány forintért ellátja a Table Service. Egy bonyolult adatstruktúrát alkalmazó programnál, mint pl. egy CMS rendszernél vagy egy banki alkalmazásnál viszont egyértelműen a SQL Azure a megfelelő választás.

Egészítsük ki példaprogramunkat Azure Table Service támogatással!

Térjünk vissza a Visual Studióba és hozzunk létre egy új gombot az alkalmazás felületén! A neve legyen **TableButton**, felirata pedig „Table sor hozzáadása”! A Table Service használatához szükségünk van továbbá még egy DLL fájlra. Vegyünk fel egy referenciát a **System.Data.Services.Client.dll** fájlra, de ne az Add Reference dialógus .NET füléről válasszuk ki, hanem a Browse fül segítségével tallózzuk ki a Toolkit mappája alatt található Binaries mappából! Ez azért szükséges, mert nekünk a DLL 4.0-s verziója kell, míg a Browse fülön a 2.0 verziót találnánk.

Amint korábban láttuk, az Azure Table Service-be osztályok példányait tudjuk feltölteni. Hozzunk létre egy erre szánt osztályt! Jobb gombbal kattintsunk a projekt nevére a Solution Explorer ablakban, és az Add almenüből válasszuk a Class menüpontot! Új osztályunkat nevezzük **SampleData**-nak! Miután létrejött, le kell származtatnunk azt a **TableServiceEntity** ősosztályból, majd fel kell vennünk bele néhány tulajdonságot.

Az osztály tetejére írjuk a következő sort:

```
using Microsoft.Samples.WindowsPhoneCloud.StorageClient;
```

Az osztály tartalma pedig legyen a következő:

```
public class SampleData : TableServiceEntity
{
    public string Name { get; set; }
    public int Age { get; set; }
}
```

Most térjünk vissza a **MainPage.xaml.cs** fájlhoz! Szükségünk lesz néhány újabb osztályszintű változóra, ezeket vegyük fel a kódfájl tetejére, a többi változódeklaráció alá!

```
CloudTableClient tableClient;
ITableServiceContext tableContext;
```

Ezután dupla kattintással lépünk be a **TableButton** gomb **Click** eseménykezelőjébe! Az előző mintapéldákban látottakhoz hasonlóan először létrehozunk egy **TableServiceContext** osztályt a megfelelő hitelesítési adatokkal, majd ezzel aszinkron módon létrehozatunk egy táblát.

```
private void TableButton_Click(object sender, RoutedEventArgs e)
{
    tableContext = new TableServiceContext("https://TÁRHELYNÉV.table.core.windows.net",
        credentials);
    tableClient = new CloudTableClient(tableContext);
    tableClient.CreateTableIfNotExist("phonetest1", TableCreatedCallback);
}
```

A callback metódusban létrehozunk egy példányt a **SampleData** osztályunkból. Ügyelünk arra, hogy a **PartitionKey** és a **RowKey** értékek egyediek legyenek. Majd (az Entity Frameworkhöz és a LINQ to SQL-hez hasonló módon) felvesszük az entitást a táblába, és elmentjük a változásokat. Ekkor történik meg a tényleges kommunikáció a szerverrel.

```
private void TableCreatedCallback(CloudOperationResponse<bool> returnValue)
{
    this.Dispatcher.BeginInvoke(() =>
    {
        if (returnValue.Exception == null)
        {
            SampleData sampleData = new SampleData();
            sampleData.PartitionKey = Guid.NewGuid().ToString();
            sampleData.RowKey = Guid.NewGuid().ToString();
            sampleData.Name = "Demo";
            sampleData.Age = 10;

            tableContext.AddObject("phonetest1", sampleData);
            tableContext.BeginSaveChanges(TableContextSaved, null);
        }
        else
        {
            MessageBox.Show(returnValue.Exception.ToString());
        }
    });
}
```

A táblamentés természetesen aszinkron történik. A művelet befejezése után értesítjük a felhasználót:

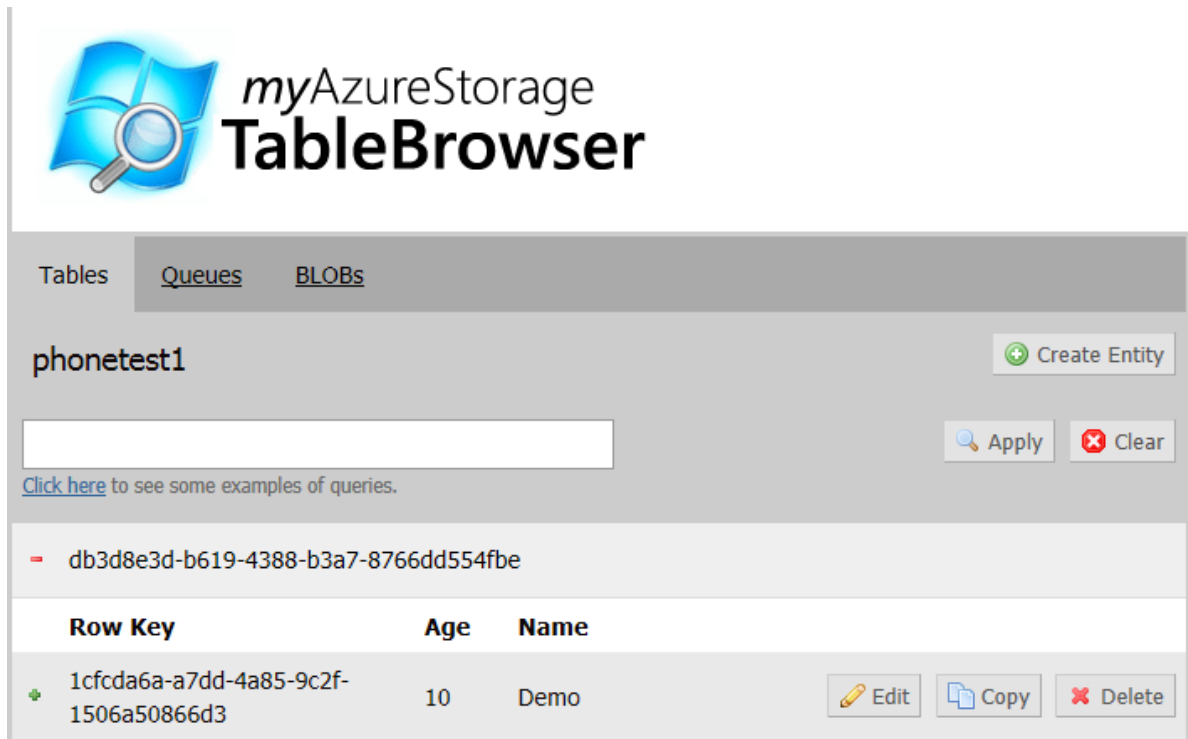
```
private void TableContextSaved(IAsyncResult result)
{
    try
    {
        var response = tableContext.EndSaveChanges(result);
        this.Dispatcher.BeginInvoke(() => MessageBox.Show("Táblasor hozzáadva!"));
    }
}
```

```

    }
    catch (Exception ex)
    {
        this.Dispatcher.BeginInvoke(() => MessageBox.Show(ex.ToString()));
    }
}

```

Teszteljük az alkalmazást! Indítsuk el, kattintsunk rá a „Table sor hozzáadása” gombra, majd a már látott módon nyissuk meg a **myazurestorage.com** oldalt! A Tables fülön megtaláljuk majd a **phonetest1** táblát, benne egy partíciót (amit egyetlen feltöltött entitásunk **PartitionKey** értékéből számolt az oldal), és azon belül az elmentett entitásunkat (9-11 ábra).



The screenshot shows the 'myAzureStorage TableBrowser' application. It has tabs for 'Tables', 'Queues', and 'BLOBs'. The 'Tables' tab is selected, showing a table named 'phonetest1'. There is a 'Create Entity' button and a search bar with 'Apply' and 'Clear' buttons. Below the search bar, there is a link to 'Click here to see some examples of queries.' The table data is as follows:

Row Key	Age	Name
1cfcda6a-a7dd-4a85-9c2f-1506a50866d3	10	Demo

Each row has an 'Edit', 'Copy', and 'Delete' button.

**9-11 ábra: Entitásunk a Table Service-ben**

Ezzel megismertük az Azure Storage harmadik szolgáltatását is. A Table Service segítségével költséghatékony, rendkívül skálázható módon tárolhatunk táblajellegű információkat. A Queue és Blob szolgáltatáshoz hasonlóan a Table Service képességeinek is csak egy töredékét fedtük le, a teljes API az Azure és a Toolkit honlapjain ismerhető meg.

## Árazás

A Windows Azure platform fontos jellemzője, hogy szolgáltatásait önállóan is igénybe vehetjük. Így ha használni szeretnénk valamelyik tárhelyszolgáltatást, semmiféle belépési költséggel vagy járulékos összeggel nem kell számolnunk – kizárólag azt kell fizetnünk, amit használunk.

Az Azure árazása folyamatosan változik, és jellemzően egyre olcsóbb lesz. Jelen sorok írásakor a tárhelyszolgáltatás árai a következők:

- A feltöltés ingyenes.
- Egy gigabájt adat tárolása (legyen az sor, tábla vagy fájl) egy hónapig: 14 dollárcent.
- Egy gigabájt adat letöltése: 15 dollárcent.
- 10 ezer elemi írási/olvasási művelet: 1 dollárcent.

A fenti adatok kizárólag tájékoztató jellegűek! Az Azure aktuális árazását a következő honlapon találjuk: <http://www.microsoft.com/windowsazure/pricing/>.

Amint láthattuk, néhány száz forintért sok gigabájt adatot tárolhatunk a felhőben, garantált rendelkezésre állás és adatbiztonság mellett. Vegyük észre, hogy az Azure táblák és várakozási sorok árazása megegyezik a fájlok árazásával – ebből a havi pár száz forintból egy több gigabájtos adatbázist vagy várakozási sort is fenntarthatunk a Table Service és Queue Service használatával.

### Felhasználó-hitelesítés szerveroldalról

Ha alkalmazásunk internetes erőforrásokat is használ, természetes igényként jelentkezik a felhasználók kezelése, hitelesítése. A klasszikus megoldás erre egy saját hitelesítő mechanizmus használata. Ha kicsit óvatosabbak vagyunk, akkor ezt nem mi magunk írjuk meg, hanem külső gyártótól származó, biztonsági szempontból alaposan átgondolt és tesztelt hitelesítési réteget vetünk be (ilyen például az ASP.NET Membership). Még itt is igaz azonban, hogy az adatbázist nekünk kell tárolni, nekünk kell kezelni a felhasználók problémáit (például jelszó-változtatás). És ami a legrosszabb: szolgáltatásunk használata előtt felhasználónknak át kell esnie a regisztrációs procedúrán. Lehet, hogy ez csak néhány mező kitöltését és egy gyors e-mailes hitelesítést igényel, de így is elvesz pár percet a felhasználó idejéből. Ez pedig gyakran túl sok: talán az Olvasóval is előfordult már, hogy egy regisztrációs űrlapot meglátva rögtön visszafordult és másik szolgáltatást keresett.

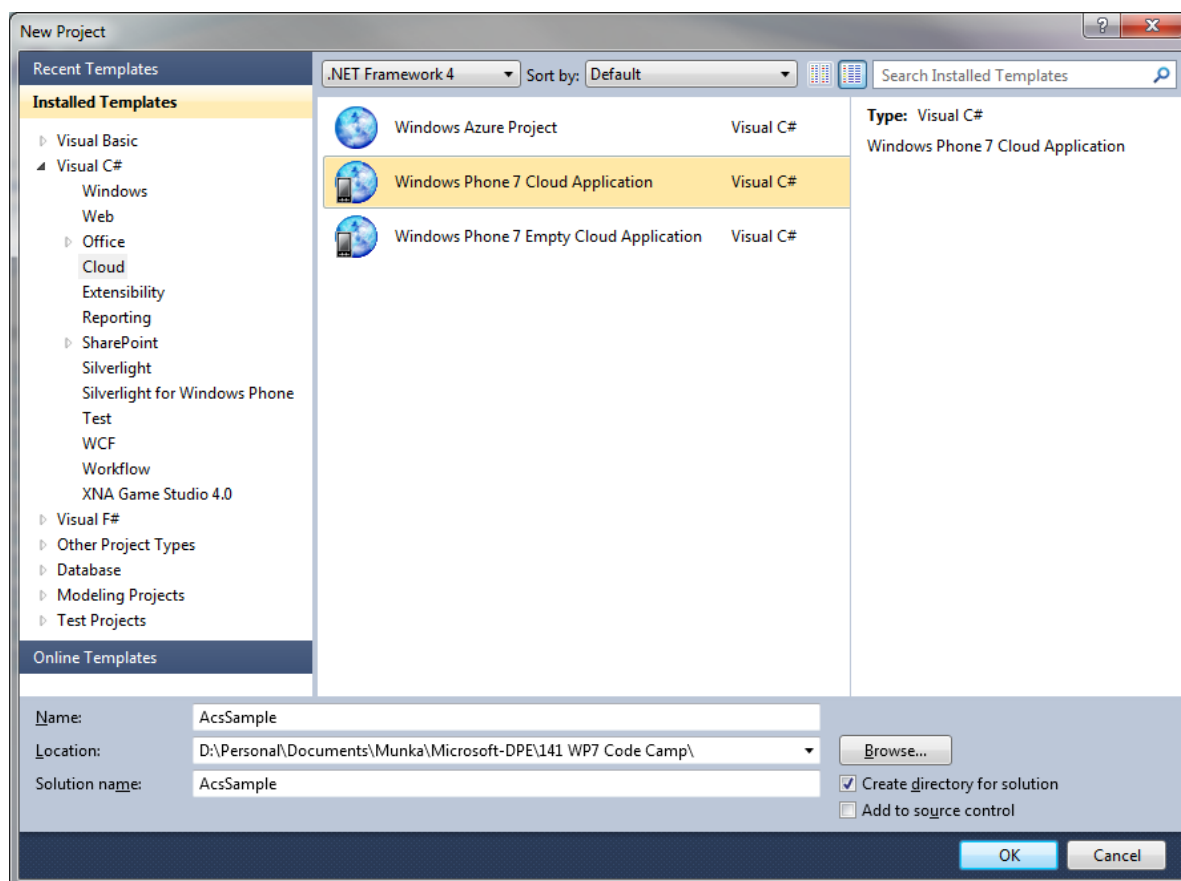
Szerencsére van arra módunk, hogy ezt elkerüljük, de mégis hitelesítsük a felhasználót! Egyszerűen használjuk valamelyik már meglévő regisztrációját! Az internet polgárainak túlnyomó többsége már tagja a Facebooknak, vagy rendelkezik Live ID azonosítóval (például van Hotmail fiókja, használ Messengert vagy van XBOX Live előfizetése), esetleg Gmail-es e-mail címe van. Windows Phone alkalmazásunk közvetlenül e szolgáltatások valamelyikébe küldheti el a felhasználót bejelentkezni. Emberünknek csak egy már létező nevet és jelszót kell beírnia. Alkalmazásunk értesül a bejelentkezés tényéről, és kap egy egyedi azonosítót a felhasználóhoz, így a hitelesítési munkát teljes egészében megspóroltuk, mindkét fél számára előnyös módon.

Ezt a bejelentkezési módszert hívjuk *federált hitelesítésnek*. Megtehetnénk, hogy telefonalkalmazásunkról közvetlenül szólítjuk meg a hitelesítés-szolgáltatókat, de erre gyakran nem adnak közvetlen támogatást, így fejlesztői szempontból nehéz a feladat. A Microsoft Windows Azure felhőplatformja azonban kínál egy szolgáltatást, amely köztes félként működik. A szolgáltatás neve Access Control. Előnye, hogy telefonalkalmazásunkból nagyon könnyen használható, és továbbítja kérésünket a megfelelő hitelesítés-szolgáltató felé. Így elfedi előlünk az egyes szolgáltatók meghívásának bonyolultságát.

Ebben a fejezet részben a Windows Azure Toolkit for Windows Phone kihasználásával készítünk egy Access Control alapú hitelesítésre alkalmas alkalmazást. Amint látni fogjuk, az itt ismertetett lépésektől kis eltéréssel az ASP.NET Membershipet is használhatnánk.

A lenti lépések végrehajtásához szükségünk lesz a Toolkitre és egy élő Azure előfizetésre. Az előző, Azure Storage használatáról szóló fejezet rész után már mindkettővel rendelkezünk.

Indítsunk egy új Visual Studio projektet! A megszokottól eltérően ne a Windows Phone Application típust, hanem a Windows Phone 7 Cloud Application típust válasszuk, amelynek a Cloud kategóriában kell lennie (9-12 ábra). Ha nem találjuk, akkor ellenőrizzük, hogy a Windows Azure Toolkit for Windows Phone-t jól raktuk-e fel!



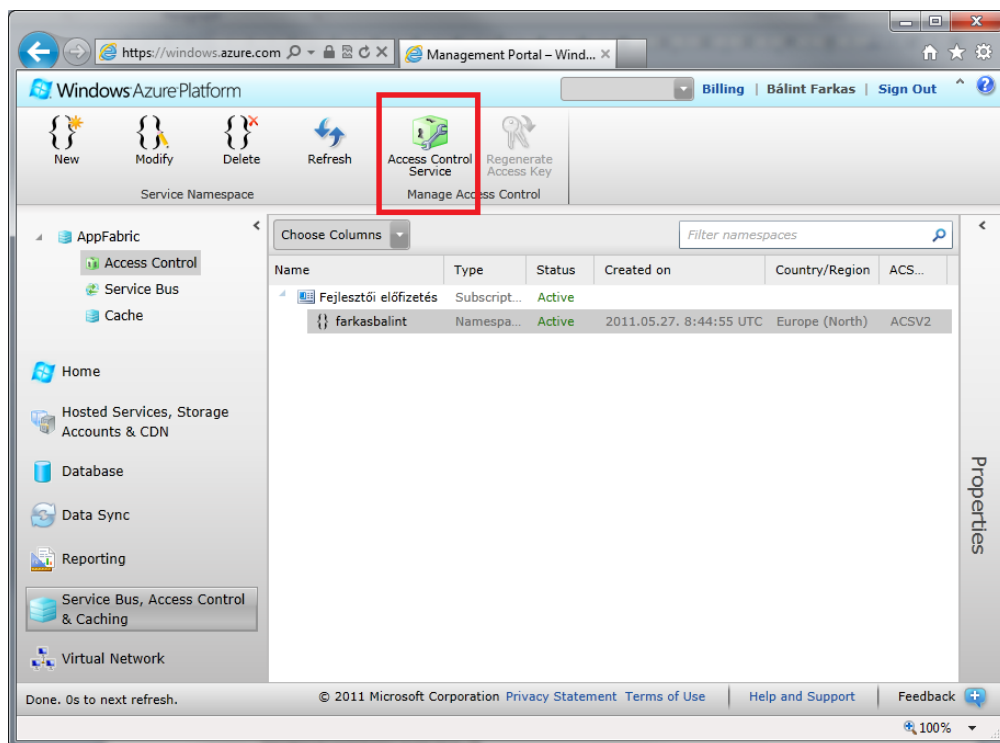
**9-12 ábra: A Windows Phone 7 Cloud Application projekt típus**

Ez a projekt típus nem egy üres alkalmazást, hanem egy kész sablont generál. Ehhez bizonyos alapinformációkra van szüksége, amiket be is kérdez tőlünk. Az első képernyőn azt kell megadnunk, hogy hol akarjuk tárolni fájljainkat (azaz egy Azure Storage account-ot), valamint hogy milyen Push Notification típusokat szeretnénk használni. Ezúttal dolgozhatunk az Azure Storage emulátorral is, így pipáljuk be az Use Storage Emulator jelölőnégyzetet, majd kattintsunk a Next gombra!

A következő párbeszédablakban kell megadnunk a használni kívánt hitelesítési mechanizmust. Itt bejelölhetnénk az ASP.NET Membership-et, de a példa kedvéért használjuk az Access Control Service-t. Válasszuk tehát az Use the Windows Azure Access Control Service lehetőséget! Megjelenik két szövegmező, ahová az ACS hitelesítési adatainkat kell beírunk. Ezeket az Azure menedzsment portálról tudjuk leszedni, úgyhogy nyissuk meg a portált a <http://windows.azure.com> címen!

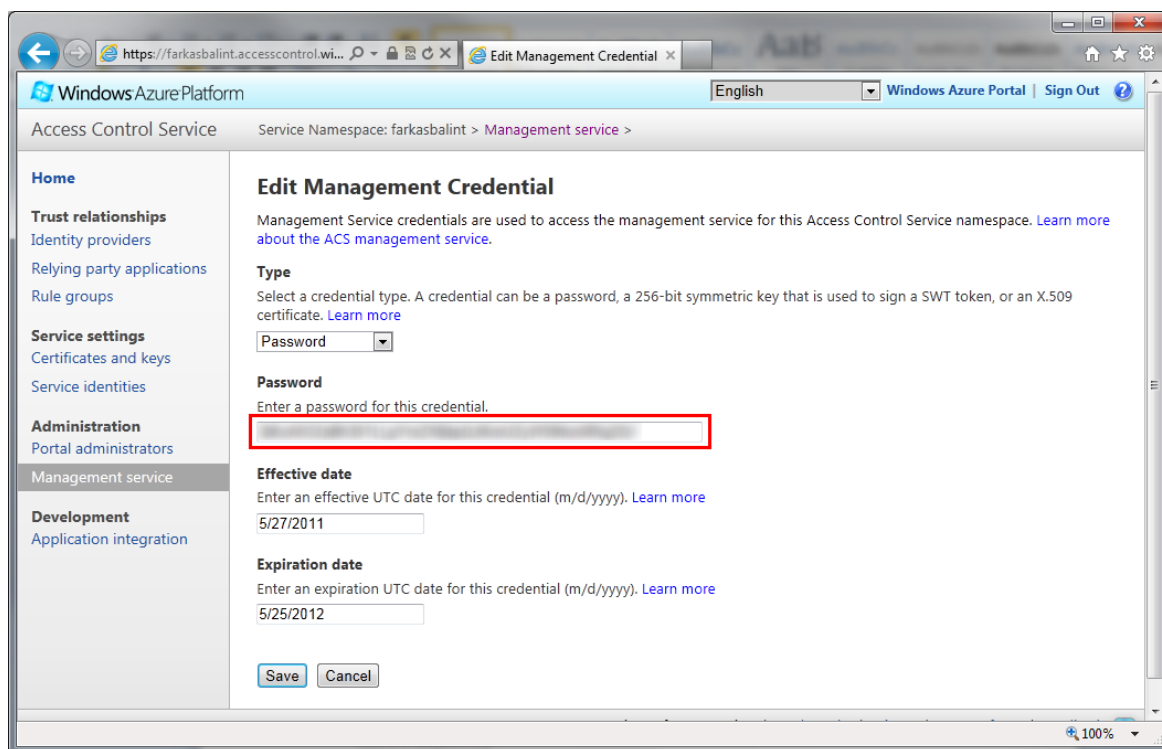
A portál bal alsó sarkában kattintsunk a Service Bus, Access Control & Caching kategóriánévre! Megnyílik a kategória. A bal felső sarokban válasszuk az Access Control elemet! Megjelennek a már létrehozott Access Control névtérek. Ha még nincs egy sem, akkor a New gomb segítségével hozzunk egyet létre, és várjuk meg, amíg státusza Activating-ról Active-ra vált! Ez néhány percig is eltarthat!

Miután a fentiekkel elkészültünk, jelöljük ki névterünket, és a szalagon kattintsunk rá az Access Control Service gombra (9-13 ábra)! Megnyílik egy újabb webes interfész, ahol szolgáltatásunkat konfigurálhatjuk. Ezt nagyrészt elvégzi majd helyettünk a varázsló. Most mindössze annyi dolgunk van, hogy leszedjük a portálról az ACS konfigurálásához szükséges hitelesítési adatokat.



**9-13 ábra: Az Access Control a Windows Azure portálon**

Az Access Control webes interfészének bal oldalán kattintsunk a Management service elemre, majd válasszuk a Management Client elemet a Management Service Accounts listából! Megjelennek az Access Control szolgáltatásunk menedzseléséhez használható hitelesítési adatok. Kattintsunk a Password elemre, és másoljuk ki az ott megjelenített értéket (9-14 ábra)!



**9-14 ábra: Access Control szolgáltatásunk menedzsmment-jelszava**

Access Control szolgáltatásunk nevét és az imént kimásolt jelszót illesszük be a Visual Studio párbeszédpanelbe, majd kattintsunk az OK gombra! A varázsló létrehozza az alkalmazássablont. Ez egy komplex művelet, mert a kódgeneráláson felül Access Control szolgáltatásunk konfigurációja is megtörténik a háttérben, így egy ideig eltarthat.

A kód létrehozása után kaphatunk egy Visual Studio figyelmeztetést, hogy projektünket újra be kell tölteni. Ha ez megtörténik, a Reload gombra kattintva hagyjuk jóvá a műveletet!

Alkalmazásunk rögtön kész is a tesztelésre. Indítsuk el az F5 gombbal! Ennek hatására beindul a szerveroldali komponens az Azure emulátorban. Mellette külön indítsuk el a Windows Phone alkalmazást is: jobb gombbal kattintsunk az **AcsSample.Phone** projekt nevére, és a Debug almenüből válasszuk a Start new instance menüpontot!

A telefonalkalmazás megnyílik, és bejelentkezhetünk az általunk választott hitelesítés-szolgáltató segítségével. Az alkalmazás teljes forráskódját pedig megtekinthetjük a Visual Studióban.

Az eddig megismert témakörökhöz hasonlóan az Access Control szolgáltatásból is csak a legszükségesebbeket láttuk. Érdemes utánaolvasni a Toolkit honlapján és az Azure dokumentumaiban. Vegyük figyelembe, hogy az ACS (az Azure többi szolgáltatásához hasonlóan) önállóan is igénybe vehető, ráadásul használata meglehetősen olcsó, így érdemes elgondolkodnunk alkalmazásán!

## Összefoglalás

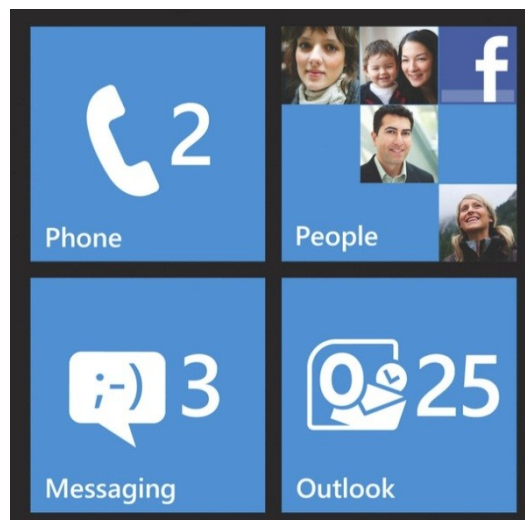
A fejezetben megismertük, hogy alkalmazásfejlesztés közben milyen lehetőségeink vannak az interneten keresztül történő kommunikációra. Egy hálózati képességekkel felruházott alkalmazás túllép a telefon korlátain, és közösségi szolgáltatásokat nyújthat, vagy a készülék képességeinél sokkal nagyobb erőforrásigényű feladatokat is elvégezhet. Érdemes élni a platform által nyújtott lehetőségekkel!





# 10. Lapkák és értesítések

A Windows Phone egyedi lehetőséget biztosít a felhasználóval való kapcsolattartásra – ez pedig a *live tile* (magyarul élő lapkának fordíthatjuk). A telefon kezdőlapja nem élettelen ikonokból áll, amelyeket esetleg egy szám megjelenítésével dinamikussá lehet tenni, hanem nagyméretű lapkákból, melyek tartalmát a fejlesztő szabadon változtathatja – ahogyan azt a 10-1 ábra mutatja.



**10-1 ábra: Windows Phone lapkák**

A lapkákon képeket, szöveget vagy számot jeleníthetünk meg. Frissíthetjük őket az alkalmazás futtatása közben, de akár szerveroldalról, az alkalmazás futása nélkül is. Alkalmazásunkhoz akár több lapka is tartozhat, például minden folyamatban lévő játszmahoz egy-egy.

A lapkákhoz –mint értesítési felülethez – szorosan kapcsolódnak a Windows Phone által kínált további értesítési szolgáltatások. A lapkák a telefon kezdőképernyőjén kapnak helyet, így ha más módon nem tudnánk üzeni a felhasználónak, fontos eseményekről maradna le. Ezért lehetőségünk van felugró üzenetet (*toast notification*) is küldeni – ilyen például az SMS érkezőkor látható sáv. Ezek az üzenetek a felhasználó aktuális tevékenységétől függetlenül mindig megjelennek, ahogyan azt a 10-2 ábra illusztrálja.



**10-2 ábra: Felugró üzenet**

Végül az is lehet, hogy egy külső eseményről nem a felhasználót szeretnénk tájékoztatni, hanem a telefonon futó alkalmazásunknak akarunk üzeni. A platform eszközeivel erre is lehetőségünk van.

A fejezetben részletesen megismerkedünk a lapkák és a további értesítési szolgáltatások képességeivel, programozásával. Ha ezeket jól tudjuk használni, akkor sokkal gazdagabb élményt kínálhatunk, mintha

alkalmazásunk csak bezárásáig működjön. A felhasználót már a kezdőképernyőről folyamatos információkkal láthatjuk el, és valós időben értesíthetjük a számára fontos eseményekről, így sokkal valószínűbb, hogy újra és újra visszatér majd alkalmazásunkhoz.

### Néhány példa a lapkák és értesítések használatára

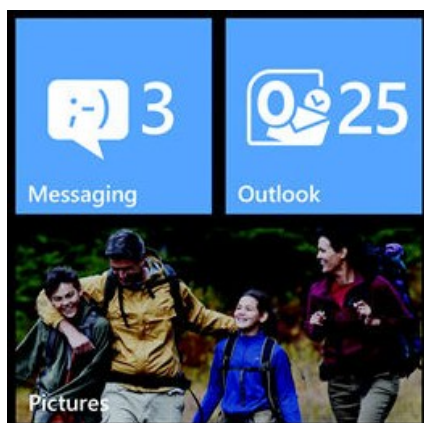
- Közösségi oldalunkhoz tartozó alkalmazásunk számmal jelenítheti meg a felhasználó olvasatlan üzeneteinek számát, szöveggel pedig kiírhatja a legújabb üzenetet. Így a felhasználó egyrészt nyomon követheti a történéseket, másrészt pedig késztetést érez majd az alkalmazás megnyitására.
- Időjárás-előrejelző alkalmazásunkban a lapkát rendszeresen frissíthetjük a várható időjárással. Ízléses képek használatával szinte biztosra vehetjük, hogy a felhasználó kitűzi majd kezdőlapjára alkalmazásunkat. Hasonlóképp érdemes lapkákat használni minden olyan alkalmazásban, ami gyakran változó, viszonylag kevés információt közöl a felhasználóval – legyen az névnap-előrejelző, részvényárfolyam-letöltő vagy focipontszám-nyilvántartó.
- Sakk-alkalmazásunkban minden egyes folyamatban lévő játszmahoz tartozhat egy lapka. Ha az ellenfél megtette lépését, ezt a lapkán keresztül közölhetjük a felhasználóval; ő pedig arra nyomva egyből a megfelelő játszmában találhatja magát.
- Egy légitársaság alkalmazása felugró üzenettel értesítheti a felhasználót, ha járatával bármi történik – így a felhasználó a weboldal folyamatos nézegetése nélkül is állandóan naprakész lehet.
- Ahelyett, hogy rendszeres időközönként kérdezzetnénk (és így feleslegesen terheljünk) a szervert, sportfogadó-alkalmazásunkban a szerver üzenhet az alkalmazásnak, ha bármi változás történt – így az alkalmazás mindig a legfrissebb információ birtokában van, felesleges adatforgalom nélkül.

### A lapkák tulajdonságai

A lapkák egyszerű, egyszínű négyzeteknek tűnnek, ám valójában rengeteg jellemzővel és képességgel ruházták fel őket a Windows Phone fejlesztői. A következőkben megismerkedünk ezekkel a jellemzőkkel, hogy később hatékonyan tudjuk őket felhasználni.

#### Méret

A Windows Phone kétféle méretű lapkát támogat: négyzet alakút, melyekből kettő fér egymás mellé, és széleset, mely nagyjából két négyzet szélességének megfelelő. A 10-3 ábra felső részében a négyzet alakú, alsó részében pedig a szélesebb lapkát láthatjuk.



10-3 ábra: A kétféle méretű Windows Phone lapka

Széles lapkája mindössze néhány beépített Windows Phone alkalmazásnak van – ilyenek pl. a **Calendar** és a **Pictures**. Mi, fejlesztők *kizárólag négyzet alakú lapkával rendelkező alkalmazásokat tudunk fejleszteni*, a széles lapkák számunkra nem használhatók.

A pontosság kedvéért: a Windows Phone teljes kijelzője 480 pixel széles és 800 pixel magas; a négyzet alakú lapkák 173 pixel szélesek és magasak.

## Kiszögezés és elrendezés

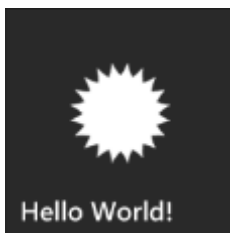
A lapkák fontos jellemzője, hogy egy telefon felhasználójaként tetszésünknek megfelelően rendezhetjük el őket – megváltoztathatjuk sorrendjüket, és törölhetjük is őket a kezdőképernyőről. A frissen feltelepített alkalmazások alapértelmezésként nem kerülnek ki a kezdőképernyőre, és a lapka törlésével a hozzá tartozó alkalmazás nem kerül eltávolításra.

Mivel a kezdőképernyőt a felhasználó a telefon minden bekapcsolásakor látja, ezért nagyon jó, ha alkalmazásunk lapkája is megjelenik ott. Azonban *lapka kitűzését kódból még kezdeményezni sem tudjuk, arra kizárólag a felhasználónak van joga*.

Ez csak az alkalmazás elsődleges lapkájára érvényes; a részleteket lásd később, a másodlagos lapkák tárgyalásánál.

## Statikus és dinamikus lapkák

Ha egy alkalmazást fejlesztünk, az külön programozói erőfeszítés nélkül is kitűzhető lesz a kezdőképernyőre, azaz rendelkezik majd lapkával. Ám alapértelmezésként ezek a lapkák mindössze az alkalmazás ikonjából és nevéből állnak, megnyomáskor az alkalmazás kezdőlapjára visznek – vagyis ezek a *statikus* lapkák, ahogyan azt a 10-4 ábra illusztrálja.



10-4 ábra: Statikus lapka az alkalmazás nevével és ikonjával

A később megismert technikákkal a lapka tartalmát frissíthetjük, animálhatjuk. Az ilyen képességekkel felruházott lapkákat nevezzük *dinamikusnak*. Egy ilyen dinamikus lapkára mutat példát a 10-5 ábra.



10-5 ábra: Dinamikus lapka, a tartalmát a program frissíti

## A lapkák felépítése

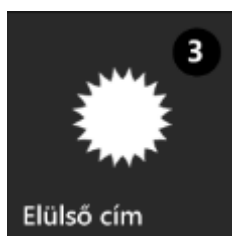
A lapkák megjelenítését egy sor előre definiált tulajdonsággal szabályozhatjuk (ezek kódból történő felhasználását később tárgyaljuk). Minden tulajdonság pontosan meghatározott módon változtatja meg a

lapka megjelenését. Például a cím mindig a bal alsó sarokban, ugyanazzal a betűtípussal jelenik meg. Nem kötelező minden tulajdonságot beállítanunk.

Minden lapka két felülettel rendelkezik: egy elülsővel és egy hátsóval. Alapértelmezésként csak az elülső felület látható; ha a hátsónak is adunk valamilyen tartalmat, akkor a lapka időnként megfordul a tengelye körül, hogy azt is mutassa.

Az elülső felület lehetséges tulajdonságai:

- Háttérkép – a teljes lapkát kitöltő kép (emlékeztetőül: a lapkák 173 pixel szélesek és magasak).
- Cím – kis betűkkel, a bal alsó sorban megjelenő felirat.
- Számláló – a jobb felső sarokban, kör közepére írt szám. Értéke minimum 1, maximum 99 lehet (0 esetén nem jelenik meg), ahogyan azt a 10-6 ábra mutatja.



**10-6 ábra: Lapka címmel és számlálóval**

A hátsó felület lehetséges tulajdonságai:

- Háttérkép – a teljes lapkát kitöltő kép, és ez más lehet, mint az elülső felület képe.
- Cím – kis betűkkel, a bal alsó sorban megjelenő felirat, amely szintén különbözhet az elülső felület címétől.
- Tartalom – nagyobb betűkkel, a lapka felső részében megjelenő felirat, amint azt a 10-7 ábra mutatja.



**10-7 ábra: Egy lapka hátsó felülete címmel és tartalommal**

Ezeket a tulajdonságokat használva a többi Windows Phone alkalmazással konzisztens lapkákat hozhatunk létre. Ha valamiért teljesen egyedi formatervet szeretnénk, akkor csak a háttérképet használjuk, amire aztán tetszésünk szerint bármit rajzolhatunk.

### Másodlagos lapkák

Amint korábban láttuk, minden Windows Phone alkalmazás rendelkezik legalább egy lapkával. Ennek angol neve: *application tile*. Fejlesztőként lehetőségünk van további, másodlagos lapkák (angolul: *secondary tile*) létrehozására is. Például ha egy játékszoftvert írunk, akkor minden egyes folyamatban lévő játékhoz tartozhat egy másodlagos lapka.

A másodlagos lapkák létrehozására kicsit eltérő szabályok vonatkoznak, mint az elsődlegesekére. Másodlagos lapkát már létrehozhatunk kódból, ám ehhez az alábbi feltételeknek kell teljesülniük:

- A másodlagos lapka kitűzését végző kód csak egy egyértelműen erre a célra szolgáló vezérlőelem hatására futhat le. Azaz rendelkezünk kell egy „Lapka hozzáadása” gombbal vagy jelölőnégyzettel. Ezt a korlátozást az alkalmazás piactérbe (*Marketplace*) való feltöltésekor ellenőrzik le – ha nem tettünk neki eleget, alkalmazásunkat visszautasítják.
- Figyelembe kell vennünk, hogy a másodlagos lapka kitűzését követően programunkat leállítja a telefon, és a kezdőképernyőre visszaugorva megmutatja a felhasználónak a frissen kitűzött lapkát.

Ezek a szabályok a felhasználó biztonságát szolgálják. Senki sem szeretné, hogy egy véletlenül feltelepített rosszindulatú (vagy csak hanyagul megírt) alkalmazás teleszemetelje gondosan kialakított kezdőképernyőjét.

A szabályokat betartva tetszőleges számú másodlagos lapkát hozhatunk létre – ez nem jelent biztonsági problémát, hiszen minden egyes másodlagos lapka létrehozását a felhasználónak kell kezdeményeznie. Ha a felhasználó törli az alkalmazás elsődleges lapkáját, a másodlagos lapkái még megmaradnak.

## Deep Linking

A másodlagos lapkák nagy előnye, hogy megadhatjuk, azok pontosan hogyan is indítsák el alkalmazásunkat. A másodlagos lapkák létrehozásakor ugyanis be kell állítanunk egy alkalmazásunkra mutató URL-t is. Ez egyrészt tartalmazza, hogy alkalmazásunk mely képernyőjét (XAML fájlját) nyissa meg a lapka, másrészt ennek a képernyőnek (XAML fájlnak) paramétereket is át tudunk adni.

Egy ilyen URL lehet például az alábbi:

```
/MainPage.xaml?id=3
```

Webprogramozóknak ez a felépítés ismerős lehet – az interneten is ugyanígy tudunk paramétereket átadni egy webalkalmazásnak.

Fontos, hogy minden másodlagos lapkának különböző linkkel kell rendelkeznie. Ha egy már létező linket megpróbálunk újra felhasználni, futásidejű hibát (*InvalidOperationException*) kapunk.

## Lapkák létrehozása és frissítése alkalmazásunkból

### A ShellTile API

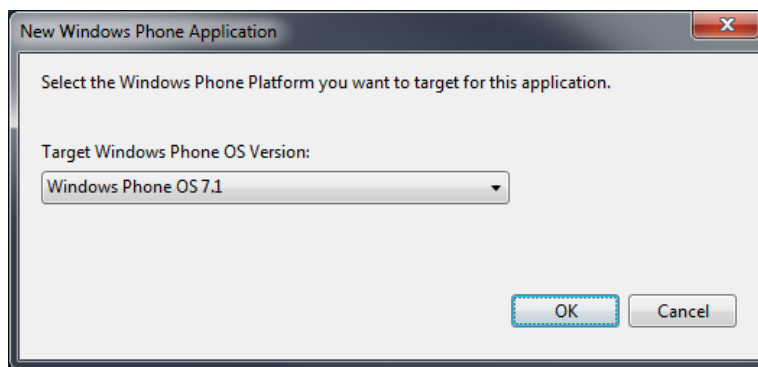
A Windows Phone első, 7.0-ás kiadásában a lapkákat kizárólag egy külső szerver igénybevételével lehetett frissíteni. Ez esetenként jelentős pluszmunkát és plusz költséget jelenthetett. A 7.5-ös, Mango kiadásban megjelent a *ShellTile API*, amivel már alkalmazásunk kódjából is tudjuk manipulálni lapkáinkat.

A *ShellTile API* egyaránt alkalmas másodlagos lapkák létrehozására, és a már létező lapkák tulajdonságainak frissítésére. Segítségével az alkalmazás futása alatt tudunk műveleteket végezni. Ha az alkalmazás futásán kívül, például valamilyen időzítés szerint vagy külső esemény hatására szeretnénk lapkáinkat frissíteni, akkor erre a később ismertetett lehetőségeket használhatjuk.

### Lapka frissítése

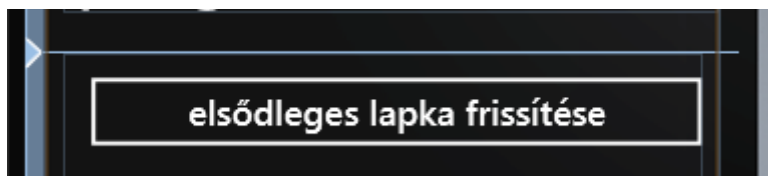
Elsőként nézzük, hogyan lehet egy már létező lapka adatait frissíteni! A módszer ugyanaz elsődleges lapkák és másodlagos lapkák esetén is. A kód nagyon egyszerű, azt egy példaalkalmazáson keresztül ismerhetjük meg:

Hozzunk létre egy új Visual Studio projektet! A projekt típusa legyen az alapértelmezett Windows Phone Application, neve **LocalTileApiSample**! Fontos, hogy a projekt létrehozása után megjelenő párbeszédpanelen Windows Phone OS 7.1 (vagy újabb) verziójú projektet hozzunk létre, amint azt a 10-8 ábra mutatja.



**10-8 ábra: Windows Phone OS 7.1 változat használata**

Ha ez a kérdés nem jelenik meg, akkor régi SDK-t használunk – frissítsunk a legújabb verzióra! A projekt létrejötte után helyezzünk el az alkalmazás felületén egy új gombot, „elsődleges lapka frissítése” felirattal! Nevezzük el ezt **UpdateAppTileButton**-nak! A nyomógomb a szerkesztőfelületen úgy jelenik meg, ahogyan azt a 10-9 ábra mutatja.



**10-9 ábra: Az új nyomógomb a szerkesztőfelületen**

A szerkesztőfelületen kattintsunk duplán a gombra, ezzel belépünk a gombhoz tartozó eseménykezelő kódba.

Szerezzünk egy referenciát az alkalmazásunkhoz tartozó elsődleges lapkára! Ehhez írjuk be az alábbi sort:

```
var appTile = ShellTile.ActiveTiles.First();
```

A **ShellTile** objektumot alapértelmezésként nem ismeri fel a Visual Studio. Így begépelése után írjuk a következő sort a kódfájl legtetéjére:

```
using Microsoft.Phone.Shell;
```

Az első kódsorból látható, hogy az **ActiveTiles** gyűjtemény legfelső elemét kérjük el a platformtól. Ez az elem minden esetben az alkalmazás elsődleges lapkáját tartalmazza, és akkor is létezik (azaz nem **null**), ha a lapka még nincs kitűzve a telefon kezdőképernyőjére.

A lapka frissítéséhez létre kell hoznunk egy adatszerkezetet, amely a megadni kívánt új adatokat tartalmazza.

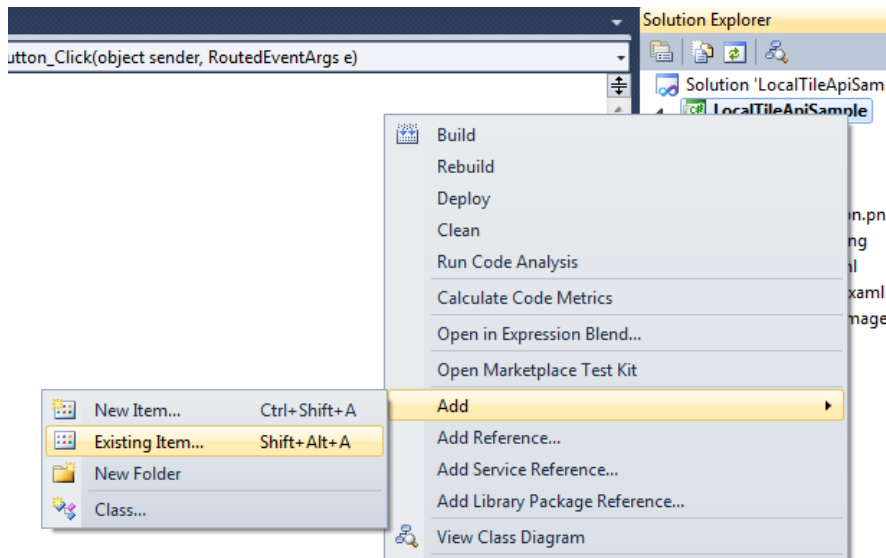
```
StandardTileData tileData = new StandardTileData();
```

Most állítsuk be a lapka tulajdonságainak új értékeit! Amint korábban láttuk, egyiket sem kötelező használni – saját alkalmazásunkban majd csak azokat állítsuk be, amelyekre szükségünk van! Kezdjük a lapka első oldalán található címmel és számlálóval!

```
tileData.Title = "Elülső cím"; //Cím az elülső oldalon  
tileData.Count = 15; //Számláló az elülső oldalon
```

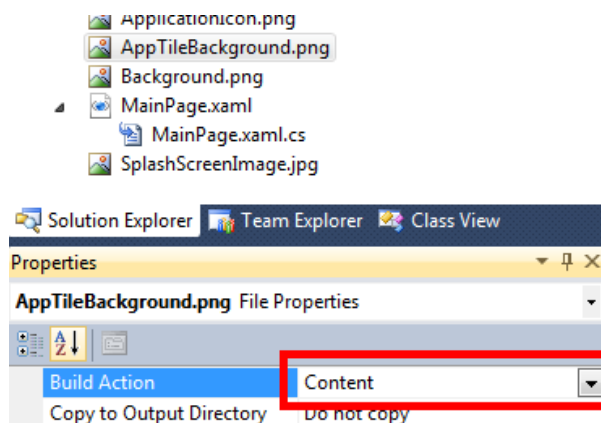
Adjunk meg egy háttérképet is! Egy lapkára előre elkészített, statikus képet is helyezhetünk, vagy akár egy képfájltra mutató internetes URL-t is megadhatunk. Kezdjük most egy előre gyártott képpel!

Kedvenc képszerkesztő alkalmazásunkban hozzunk létre egy képet! Érdekes 173×173 pixelesre gyártani, de ha eltérő méretűt hozunk létre, akkor a telefon azt átméretezi majd nekünk. Mentsük el a képet PNG vagy JPG formátumban, **AppTileBackground** néven! Térjünk vissza a Visual Studióba, és a Solution Explorer ablakban kattintsunk jobb gombbal a projekt nevére (**LocalTileApiSample**), majd az Add menüből válasszuk az Existing Item... menüpontot (10-10 ábra)! Tallózzuk ki az imént létrehozott képet, és szűrjük be alkalmazásunkba!



10-10 ábra: Az Add Existing Item funkció kiválasztása

Most közölnünk kell a Visual Studióval, hogy a frissen megadott fájlt tartalomként akarjuk használni – ezután tudunk majd kódból hozzáférti. A Solution Explorer-ben jelöljük ki a frissen beszúrt fájlt, majd a Properties ablakban állítsuk Build Action tulajdonságát Content értékűre!



10-11 ábra: Képfájl felvétele az alkalmazásba tartalomként

Miután ezzel elkészültünk, már nincs más dolgunk, mint beírni a **tileData** adatstruktúrába az így létrehozott kép URL-jét.

```
tileData.BackgroundImage = new Uri("/AppTileBackground.png", UriKind.Relative);
```

A sor végén található **UriKind.Relative** paraméter nagyon fontos. Ez adja meg, hogy az URL relatívként értelmezendő. Ha ezt nem írjuk be, a program hibára fut.

Ezzel elkészültünk a lapka első oldalának beállításával. Alkalmazzuk az idáig elkészült beállításokat, és nézzük meg, mit alkottunk! Az elkészült beállítások életbeléptetéséhez az alábbi sort kell beírunk.

```
appTile.Update(tileData);
```

Futtassuk programunkat az F5 megnyomásával! Az emulátorban tűzzük ki a programot a telefon kezdőképernyőjére, majd lépünk be abba, és nyomjuk meg az „Elsődleges lapka frissítése” gombot! Ha ezután visszatérünk a kezdőképernyőre, látni fogjuk, hogy a lapka tulajdonságai tényleg megváltoztak, amint azt a 10-12 ábra is mutatja.



**10-12 ábra: A lapka megjelenése a kezdőképernyőn**

Térjünk vissza a Visual Studióhoz, és állítsuk be a lapka hátsó felületének tulajdonságait is! A következő kódrészleteket az `appTile.Update(tileData);` sor fölé írjuk be!

A hátoldalon szintén van **Title** tulajdonság; számláló viszont nincs, helyette **Content** van, amely egy szöveges értéket vár.

```
tileData.BackTitle = "Hátulsó cím";  
tileData.BackContent = "Tartalom";
```

Háttérképnek most ne egy helyi, hanem egy internetes képet válasszunk! Kedvenc képkereső szolgáltatásunkkal keressünk egy szimpatikus, kisméretű ikont, majd vágjuk ki ennek URL-jét a böngészőből! (A kép ne legyen túl nagy, mert a nagyon elhúzódnak a letöltéseket a telefon megszakítja!)

```
tileData.BackBackgroundImage = new Uri("http://IKON_AZ_INTERNETROL");
```

Újra futtassuk le az alkalmazásunkat! Nyomjuk meg az „Elsődleges lapka frissítése” gombot, majd a telefon Windows gombjával lépünk ki a kezdőképernyőre! Néhány másodperc elteltével (amit kizárólag a telefon határoz meg, nekünk nincs beleszólásunk) a lapka megfordul majd, és láthatjuk a beállításaink eredményét, amint azt a 10-13 ábra mutatja:



**10-13 ábra: A lapka hátoldala**

Ezzel a lapkák valamennyi tulajdonságának beállítását és használatát láttuk. A megírt teljes metódus törzse az alábbi:



```
private void UpdateAppTileButton_Click(object sender, RoutedEventArgs e)
{
    var appTile = ShellTile.ActiveTiles.First();

    StandardTileData tileData = new StandardTileData();
    tileData.Title = "Elülső cím"; //Cím az elülső oldalon
    tileData.Count = 15; //Számláló az elülső oldalon
    tileData.BackgroundImage = new Uri("/AppTileBackground.png", UriKind.Relative);

    tileData.BackTitle = "Hátulsó cím";
    tileData.BackContent = "Tartalom";
    tileData.BackBackgroundImage = new Uri("http://IKON_AZ_INTERNETROL");

    appTile.Update(tileData);
}
```

## Másodlagos lapka létrehozása

Amint korábban már láttuk, alkalmazásunkhoz tartozhatnak másodlagos lapkák is. Ezeket (a felhasználó kérésére) kódból mi magunk hozhatjuk létre. Készítésükkor pedig paramétereket építhetünk beléjük, amiket megnyomásukkor megkap az alkalmazás, így ezeket feldolgozva egyből programunk „belsejébe” (pl. a megfelelő játszma, megfelelő részvényhez stb.) vihetjük a felhasználót.

Hozzunk létre programunk felületén két gombot, egyiket „Kék”, a másikat „Piros” felirattal! A gombok megnyomására alkalmazásunk egy-egy másodlagos lapkát hoz majd létre. A „Kék” gomb hatására létrehozott másodlagos lapka kék háttérszínnel, a másik pedig piros háttérszínnel indítja majd programunkat.

A gombok nevei legyenek **BlueButton** és **RedButton**!

Létrehozásuk után kattintsunk duplán a „Kék” feliratú gombra, hogy eljussunk a kódnézetbe! Másodlagos lapka létrehozásához ugyanarra a **StandardTileData** adatszerkezetre lesz szükségünk, mint amit az előző fejezetrészben is használtunk. Kihasználva, hogy nem kell minden tulajdonságot megadnunk, a leendő másodlagos lapka csak címet kap:

```
StandardTileData tileData = new StandardTileData();
tileData.Title = "Kék";
```

Az így elkészült másodlagos lapkát pedig egyszerűen felvesszük az alkalmazás lapkái közé:

```
ShellTile.Create(new Uri("/MainPage.xaml?color=blue", UriKind.Relative), tileData);
```

Nézzük meg figyelmesen a lapka beágyazott URI-ját! Láthatjuk, hogy két részből áll: egyrészt megadjuk a megnyitandó Silverlight nézetet (**MainPage.xaml**), másrészt egy paramétert is átadunk (neve: **color**, értéke: **blue**). Alkalmazásunk e paraméter alapján tudja ellenőrizni majd, hogy a felhasználó melyik lapka segítségével indította el.

Miután ezzel megvagyunk, az itt beírt három sort másoljuk át a „Piros” gomb **Click** eseményének kezelőjébe is (ügyelve arra, hogy a gomb feliratát és paraméterét átírjuk „Piros”-ra és **red**-re)!

Most állítsuk be, hogy a lapkákra kattintva alkalmazásunk a megfelelő háttérszínben pompázva induljon – azaz használjuk a Deep Linking lehetőséget!

A háttérszín beállító kódot akkor kell futtatnunk, amikor alkalmazásunk betöltődik. Használjuk erre az **OnNavigatedTo** eseményt! Álljunk a kurzorral egy üres kódsorhoz (mondjuk közvetlenül a **private void UpdateAppTileButton\_Click** sor fölé), és gépeljük be:

```
override onnavigatedto
```

A Visual Studio erre egy listából felkínálja számunkra azt az eseményt, amelynek kódját felül szeretnénk írni. Nincs más dolgunk, mint a Tab gomb megnyomásával létrehozni ezt az eseménykezelőt.

Az eseménykezelőn belül a **NavigationContext** objektum **QueryString** tulajdonságán keresztül kérhetjük el alkalmazásunk indítási paramétereit. Elsőként megnézzük, hogy egyáltalán kaptunk-e **color** nevű paramétert, majd leellenőrizzük, hogy ennek értéke **blue**-e. Ha egyezést találunk, akkor alkalmazásunk hátterét kékre állítjuk.

```
if (NavigationContext.QueryString.ContainsKey("color") &&
    NavigationContext.QueryString["color"] == "blue")
{
    LayoutRoot.Background = new SolidColorBrush(Colors.Blue);
}
```

Ezt a kódrészletet még egyszer – értelemszerűen – felhasználva tudjuk kezelni a piros szín esetét is.

Futtassuk az F5 gombbal alkalmazásunkat! Nyomjuk meg előbb a Kék, majd a Piros gombot! Látni fogjuk, hogy a telefon mindkét gomb megnyomására bezárja az alkalmazásunkat, és létrehoz nekünk egy-egy új másodlagos lapkát. A létrehozott lapkákat megnyomva pedig alkalmazásunk újra elindul – a kiválasztott háttérszínnel. Ezt a mechanizmust kihasználva a háttérszín-váltásnál sokkal komplexebb belső navigációt is kivitelezhetünk.

Fontos tudni, hogy egy adott URL-ű másodlagos lapkából egyszerre csak egy lehet. Vagyis ha a Kék gombot még egyszer megnyomjuk, akkor alkalmazásunk hibára fut. Egyszerűen védekezhetünk ez ellen a hibalehetőség ellen, ha a lapka kirakása előtt ellenőrizzük, hogy a **ShellTile.ActiveTiles** gyűjtemény tartalmaz-e már a létrehozni kívánt URL-lel egy lapkát.

### Másodlagos lapka törlése

Előfordulhat, hogy a kirakott másodlagos lapkákat törölni szeretnénk. Erre a felhasználónak az alkalmazásunktól függetlenül is van lehetősége, de az aktuális játszma, repülőút, stb. lezárulta után célszerű automatikusan törölni a hozzá tartozó, korábban létrehozott lapkát.

Hozzunk létre egy újabb gombot felületünkön, „Kék törlése” felirattal, **DeleteBlueButton** névvel, majd dupla kattintással lépünk be az eseménykezelőjébe! A lapka törléséhez egyszerűen ki kell azt keresnünk a lapkák gyűjteményéből, majd meghívni a **Delete()** metódusát. A kikereséshez használhatjuk a lapka URL tulajdonságát, amely garantáltan egyedi – ahogyan azt korábban már bemutattam.

Az alábbi kódrészlet (a C#-ba épített LINQ lekérdezőnyelv segítségével) ellenőrzi, hogy létezik-e a törölni kívánt lapka a gyűjteményben, és ha létezik, letörli:

```
if (ShellTile.ActiveTiles.Count(i => i.NavigationUri.ToString() == "/MainPage.xaml?color=blue")
    > 0)
{
    ShellTile.ActiveTiles.Single(i => i.NavigationUri.ToString() ==
        "/MainPage.xaml?color=blue").Delete();
}
```

Vegyük észre, hogy a lekérdezéshez használt módszert használhatjuk a lapka létrehozása előtt is, annak ellenőrzésére, hogy az nem létezik-e már!

### Lapkák frissítése Background Agent-ek segítségével

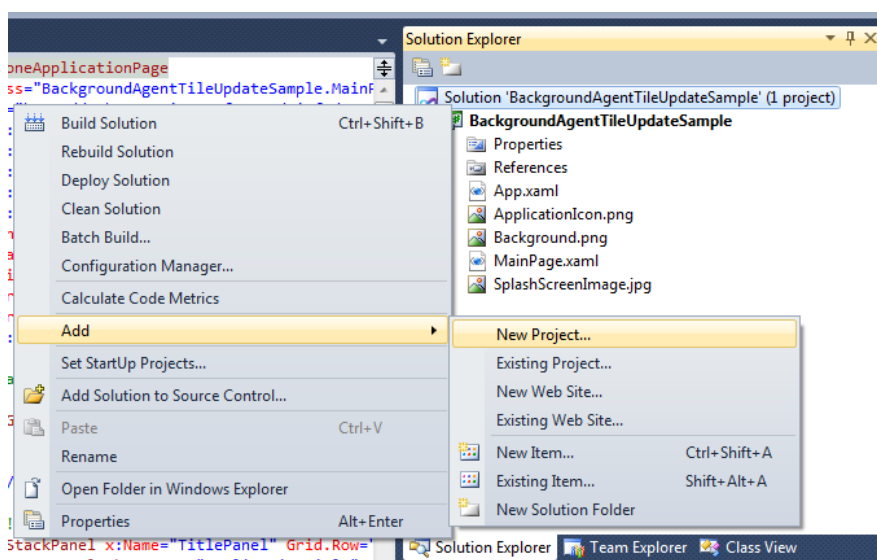
A korábban bemutatott ShellTile API segítségével alkalmazásunk futása alatt teljes körűen tudjuk frissíteni, bővíteni és törölni lapkáinkat. A lapkák igazi haszna azonban éppen abban rejlik, hogy azok képesek alkalmazásunk futása nélkül is frissülni! A következő fejezet részben a lapkák Background Agent-ből való frissítését ismerjük meg. Ez a módszer nem igényel külső szervert, és időzített frissítést használ, azaz alkalmazásunk leállása után is működik.

A Background Agent a Windows Phone 7.5-ben debütált, és nem kizárólag a lapkák frissítésére való – épp ellenkezőleg, számos egyéb rendszeresen végrehajtandó vagy erőforrás-igényes feladat elvégzésére használható. Részletes ismertetésével a könyv 5. fejezete foglalkozik.

A következő oldalakon egy példaprogramot készítünk, amely egy Background Agentet regisztrál be. Ez a háttérben futva rendszeres időközönként frissíti majd az alkalmazás lapkáját. A példaprogram elkészítése során csak a Background Agent funkcionalitásának szükséges részét használjuk ki. A szolgáltatás teljes leírása megtalálható az 5. fejezetben.

Készítsünk egy új Visual Studio projektet; legyen ez egy Windows Phone Application! Projektünk neve legyen **BackgroundAgentTileUpdateSample**! Az előző fejezetrészhez hasonlóan itt is Windows Phone OS 7.1 verziót (vagy újabbat) válasszunk!

A Background Agent kódját külön projektbe kell helyezni. A Windows Phone Application projekt létrejötte után kattintsunk jobb gombbal a Solution Explorer ablak legtetetjén található *Solution* 'BackgroundAgentTileUpdateSample' sorra, és az Add menüből válasszuk a New Project lehetőséget, amint azt a 10-14 ábra mutatja!



10-14 ábra: A Background Agent projekt hozzáadása

Itt ismét találkozunk a projektek listájával. Hozzunk létre egy Windows Phone Scheduled Task Agent típusú projektet! Ennek neve maradhat az alapértelmezett **ScheduledTaskAgent1**.

Miután mindkét projektünket létrehoztuk, tudatnunk kell a telefonalkalmazással, hogy hol találja a neki rendelt Background Agent-et. Ehhez a **BackgroundAgentTileUpdateSample** projektből fel kell vennünk egy referenciát a **ScheduledTaskAgent1** projektre. Kattintsunk jobb gombbal a **BackgroundAgentTileUpdateSample** projekt References mappájára, válasszuk az Add Reference lehetőséget, és a megjelenő ablak Projects fülén válasszuk ki a **ScheduledTaskAgent1** elemet!

A projektfájlok létrejötte után a Visual Studio automatikusan megnyitja számunkra a **ScheduledAgent.cs** állományt. A Background Agent futásakor az itt található **OnInvoke()** metódust hívja meg a futtatókörnyezet, így ide kell elhelyeznünk a lapkát frissítő kódunkat. A metódus végén már megtalálható a **NotifyComplete()** hívás. Ez jelzi az operációs rendszer számára, hogy a Background Agent elvégezte a rábízott feladatokat, és futása leállítható. Ezért ezt a sort ne töröljük ki, és hagyjuk mindig a saját kódunk után!

A lapka frissítésére a ShellTile API-t vehetjük igénybe, így az előző fejezetrészben megismerteket hasznosíthatjuk újra. Példaalkalmazásunk az aktuális időt írja majd ki a lapkára. Egy éles alkalmazás ennél természetesen sokkal bonyolultabb logikát is meghívhat a Background Agent-ből.

A kód megírásához először is szükségünk van egy sor névtérre. Írjuk be az alábbiakat a kódfájl (**ScheduledAgent.cs**) legtetetjére!

```
using System;
using System.Linq;
using Microsoft.Phone.Shell;
```

Miután ezzel megvagyunk, gépeljük be az alábbi kódot az **OnInvoke()** metódusba (a **NotifyComplete()** fölé)! Ez semmi újdonságot nem tartalmaz, a lapka címét frissíti a ShellTile API segítségével:

```
StandardTileData tileData = new StandardTileData();
tileData.Title = DateTime.Now.ToShortTimeString();
ShellTile.ActiveTiles.First().Update(tileData);
```

A Background Agent-et ezzel felprogramoztuk.

Magától természetesen nem fog lefutni, ehhez telefonalkalmazásunknak előbb időzítenie kell azt. Minden Windows Phone alkalmazáshoz legfeljebb egy Background Agent tartozhat. Ezt kétféle módon időzíthetjük:

- **PeriodicTask** időzíteni mód: A Background Agent rövid ideig (25 másodpercig) kap futási jogot kb. 30 percenként.
- **ResourceIntensiveTask** időzíteni mód: bizonyos külső feltételek (pl. külső áramellátás) megléte esetén kap futási jogot, 10 percig.

A lapka frissítése jellemzően nem erőforrás-igényes, de rendszeresen szeretnénk elvégezni, így a **PeriodicTask** időzíteni módot használjuk.

Térjünk vissza a **BackgroundAgentTileUpdateSample** projektbe! A Background Agent időzítését egy gomb végzi majd. Fontos tudni, hogy időzítése után két héttel a Background Agent automatikusan törlődik a rendszerből, így felhasználónknak legalább ilyen gyakorisággal újra kell időzíttetnie – célszerű, ha egy éles alkalmazásban erre valamilyen módon emlékeztetjük.

Húzzunk ki a kezelőfelületre egy gombot; felirata legyen „Background Agent időzítése”, neve pedig **ScheduleBackgroundAgentButton**! Miután kihúztuk, dupla kattintással lépünk be a gomb **Click** eseménykezelőjébe!

A kódfájl tetejére helyezzük el az alábbi névteret, ami a Background Agent-ek időzítéséhez szükséges eszközöket tartalmazza!

```
using Microsoft.Phone.Scheduler;
```

Az időzítő az egyes Background Agent-eket nevük alapján azonosítja. Első feladatunk, hogy ellenőrizzük, létezik-e már az általunk kívánt néven Background Agent. Ha igen, letöröljük. (Ez könnyen előfordulhat, ha a felhasználó a kéthetes automatikus törlési időszak alatt újra kéri a Background Agent időzítését.) Ennek elvégzéséhez helyezzük az alábbi sorokat a gomb eseménykezelőjébe:

```
string periodicTaskName = "TileUpdaterSample";

PeriodicTask periodicTask = ScheduledActionService.Find(periodicTaskName) as PeriodicTask;
if (periodicTask != null) ScheduledActionService.Remove(periodicTaskName);
```

Ezután utasítjuk a futtatókörnyezetet a Background Agent időzítésére. Ehhez először egy objektum formájában létrehozzuk és beállítjuk a leírását (ezt látja majd a felhasználó az operációs rendszer beállításai között, az épp futó Background Agent-ek listájában). Ezután következik a tényleges időzítés. Ezt mindenképp tegyük egy **try...catch** blokk belsejébe, ugyanis a felhasználó a telefon beállításai között letilthatja a Background Agent-eket, melyről egy kivétel formájában értesülünk.

```
periodicTask = new PeriodicTask(periodicTaskName);
periodicTask.Description = "This is the Tile Updater sample.";

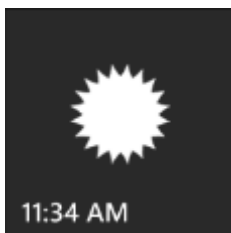
try
{
    ScheduledActionService.Add(periodicTask);
}
catch (Exception ex)
{
    MessageBox.Show(ex.Message);
}
```

Észrevehető, hogy a fenti kódban semmilyen módon nem adtuk meg, *melyik* osztályt kívánjuk az időzített feladat futtatása során használni. Ezt az operációs rendszer onnan tudja, hogy korábban felvettünk egy referenciát a **ScheduledTaskAgent1** projektre, és ezt Background Agent-nek ismerte fel.

Ezzel tulajdonképpen elkészültünk, ám a Background Agent-ek kb. 30 percenként futnak, ami nagyon megnehezíti a tesztelésüket. A futtatókörnyezet biztosít egy hívást, amellyel a Background Agent lefuttatása néhány másodperc várakozás után kikényszeríthető. A próbafuttatáshoz érdemes beírni a következő programsort is, *de az alkalmazás publikálása előtt feltétlenül töröljük ki*, mert a Marketplace ellenőrzésen valószínűleg meg fog bukni!

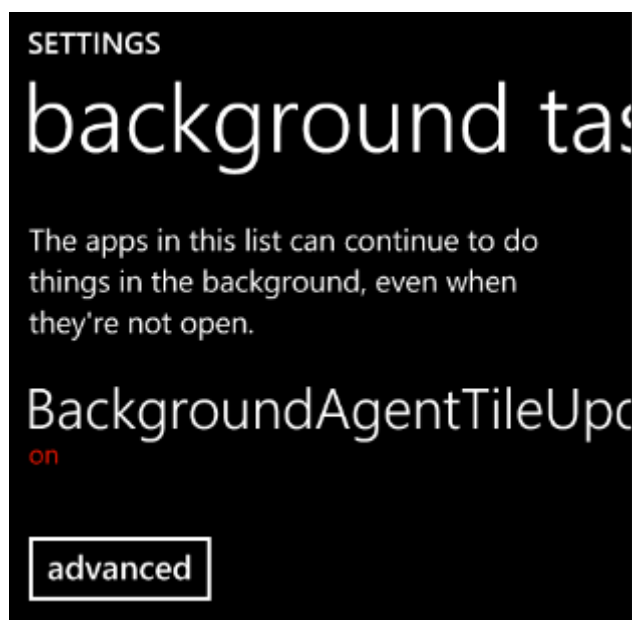
```
ScheduledActionService.LaunchForTest(periodicTaskName, TimeSpan.FromSeconds(30));
```

Az F5 megnyomásával futtassuk alkalmazásunkat! Nyomjuk meg a „Background Agent időzítése” gombot, majd az emulátor Windows gombjával lépünk ki programunkból, hogy az Agent lefuthasson! Tűzzük ki alkalmazásunkat a kezdőképernyőre! Kisvártatva tanúi leszünk a lapka frissülésének, amint azt a 10-15 ábra is mutatja.



**10-15 ábra: A Background Agent által frissített lapka**

Ha szeretnénk, a Background Agent működését megszemlélhetjük a telefon beállításai között is. Ehhez lépünk be a telefon beállításai közé (Settings), lapozunk az Applications fülre, és nyomjuk meg a Background Tasks lehetőséget (10-16 ábra)!



10-16 ábra: Background Agent-ünk a telefon listájában

A Background Agent segítségével alkalmazásunk a leállása után is naprakészen tarthatja lapkáját. Egy folyamatosan frissülő lapka pedig jó ok arra, hogy a felhasználó a kezdőképernyőre tűzve tartsa alkalmazásunkat. A Background Agent nagy előnye, hogy ehhez még külső szerverre sincs szüksége.

## Lapkák frissítése ShellTileSchedule segítségével

Az előző két fejezetrészen megismerkedtünk a ShellTile API-val és a Background Agent-ekkel. Mindkét módszer segítségével a telefonon futó kódból, imperatív módon tudtuk frissíteni lapkáinkat, szerver igénybevétele nélkül. Bizonyos esetekben azonban kényelmesebb lehet, ha a szükséges tartalmat nem a készüléken, hanem egy kiszolgálón állítjuk elő. A *ShellTileSchedule* arra ad lehetőséget, hogy a megjelenítendő információt szerveroldalon készítsük elő, a telefon pedig egyszerűen csak letöltse azt.

### Időzítés beállítása az elsődleges lapkára

Korábban megismerkedtünk a lapkák tulajdonságaival. A **ShellTileSchedule** ezek közül kizárólag a lapka előlő felületén található háttérképet tudja frissíteni, a többi tulajdonság ezen az API-n keresztül nem módosítható.

Használatához (a Background Agent-hez hasonlóan) alkalmazásunk futása közben időzítenünk kell a frissítéseket, amelyek utána a háttérben futnak majd, alkalmazásunk leállítását követően is. Az időzítés beállításához az alábbi paramétereket adhatjuk meg:

A frissíteni kívánt lapka (lehet elsődleges vagy valamelyik másodlagos lapka is)

- Hányszor történjen meg a frissítés (egyszer, fix alkalommal vagy határozatlan ideig)?
- Milyen gyakran történjen meg a frissítés (óránként, naponta, hetente vagy havonta)?
- Milyen URL-ről töltődjön le a kép?

A letöltésre kiválasztott kép kizárólag internetes URL-ről származhat, azaz nem használhatunk a telefonon lévő helyi erőforrásokat. A képfájl maximum 80 kilobájtos lehet, és letöltése nem tarthat 30 másodpercnél tovább. Ha az időzített művelet végrehajtása 3 egymást követő alkalommal sikertelen, akkor az időzítést törli a futtatókörnyezet.

Írjunk egy rövid példaprogramot, ami bemutatja a **ShellTileSchedule** használatát!

Az eddigiekhez hasonlóan hozzunk létre egy új Windows Phone Application típusú projektet **ShellTileScheduleSample** néven, Windows Phone OS 7.1 vagy újabb verziót célozva! A **ShellTileSchedule** beállítását már rögtön az alkalmazás indulásakor elvégezzük. Ehhez kódunkat helyezzük a **MainPage()** konstruktorába, közvetlenül az **InitializeComponent()** metódushívás után! Lapkáinkat határozatlan ideig, óránként szeretnénk frissíteni.

Az időzítés beállításához egy **ShellTileSchedule** típusú objektumot használunk. Beállítjuk a kívánt paramétereket, majd a **Start()** metódus hívásával érvénybe léptetjük az időzítést.

```
ShellTileSchedule schedule = new ShellTileSchedule();
schedule.Recurrence = UpdateRecurrence.Interval;
schedule.Interval = UpdateInterval.EveryHour;
schedule.MaxUpdateCount = 0;
schedule.RemoteImageUri = new
Uri("http://www.weather.gov/forecasts/graphical/images/conus/MaxT1_conus.png");
schedule.StartTime = DateTime.Now;
schedule.Start();
```

Nézzük meg az egyes paraméterek jelentését!

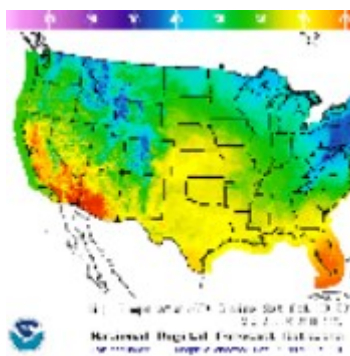
- A **Recurrence** tulajdonság értékétől függ, hogy időzítésünk hányszor fut le. Az **UpdateRecurrence.Interval** érték esetén többször is, az **UpdateRecurrence.Onetime** érték esetén pedig csak egyszer.
- Az **Interval** tulajdonság adja meg, hogy lapkánk milyen gyakorisággal frissüljön. Lehetséges értékei: **EveryHour**, **EveryDay**, **EveryWeek** és **EveryMonth**. A tulajdonság értékét csak akkor veszi számításba a telefon, ha a lapka nem csak egyszer frissítendő (lásd **Recurrence** tulajdonság).
- A **MaxUpdateCount** segítségével különböztethetjük meg a határozatlan ideig futó és a bizonyos időre szóló időzítéseket. Ha a tulajdonság értéke nagyobb, mint 0, akkor a lapka a megadott számú alkalommal lesz frissítve. Ha 0 vagy annál kisebb, akkor nincs korlát a frissítések számára. (Ha a **Recurrence** tulajdonságnál **UpdateRecurrence.Onetime** értéket adtunk meg, akkor ennek a beállításnak nincs értelme.)
- A **RemoteImageUri** tulajdonsággal állíthatjuk be a letölteni kívánt kép címét. Fontos, hogy ez csak az internetről származhat (azaz helyből nem). Egy éles alkalmazásnál ennek kiszolgálására egy saját szervert használhatunk, amely egy adott URL-en egy dinamikusan frissülő képet tesz elérhetővé. Ez ASP.NET Web Forms esetén például képeket visszaadó Handler-ek (ASHX fájlok) segítségével oldható meg, de a további webes technológiák is biztosítanak erre lehetőséget.
- A **StartTime** tulajdonsággal határozhatjuk meg az időzítés kezdő időpontját.

Végül a **Start()** metódushívás kiadja a parancsot az időzítés tényleges elvégzésére. Ezt követően a telefon megvárja a **StartTime** időpontot, és onnantól kezdve a megadott beállítások szerint végzi a lapka frissítését.

A fenti tulajdonságok használatával készíthetünk egyszeri alkalommal, határozott alkalommal vagy akár határozatlan ideig futó időzítéseket is. Ne feledjük azonban a korábban látott korlátozásokat! Három egymást követő sikertelen frissítési kísérlet után (azaz ha a képfájl nagyobb, mint 80 KB, vagy letöltése 30 másodpercnél tovább tart) a telefon törli az időzítést. Ezért célszerű, ha alkalmazásunk minden egyes indításakor újra és újra elvégzi a beállítást.

A kód megírása után indítsuk el alkalmazásunkat, majd lépünk ki belőle és tűzzük ki a kezdőképernyőre. Némi idő elteltével megtörténik a frissítés; eredménye a 10-17 ábrához hasonlít majd. Lehet, hogy erre sokat kell várnunk (akár egy órát is), mert az energiatakarékosság érdekében az esedékes frissítéseket összegyűjti és egyszerre hajtja végre a telefon.





10-17 ábra: A *ShellTileSchedule* által frissített lapka

### Időzítés beállítása egy másodlagos lapkára

Hogy állíthatunk be **ShellTileSchedule** időzítést egy másodlagos lapkára? Az objektum paraméterezése nem változik, csak létrehozásakor át kell neki adnunk a frissíteni kívánt másodlagos lapkát. (Emlékezzünk vissza, hogy a másodlagos lapkákra mutató referenciákat a **ShellTile.ActiveTiles** gyűjteményben találjuk, ahol az első elem az elsődleges lapka, az összes többi elem pedig az épp aktuális másodlagos lapka.)

Így ha egy másodlagos lapkára szeretnénk időzítést létrehozni, fenti példakódunkat így kellene módosítanunk:

```
ShellTileSchedule schedule = new ShellTileSchedule(tile);
```

A **tile** változóba egy **ShellTile** típusú objektumot vár a telefon.

### Időzítés törlése

Előfordulhat, hogy a létrehozott **ShellTileSchedule** értelmét veszti, így szeretnénk azt letörölni. Egy időzítés törléséhez először referenciát kell rá szereznünk, majd meg kell hívnunk a **Stop()** metódusát. Előfordulhat, hogy az időzítést a programunk egy korábbi futásakor állítottuk be, így az eredeti **ShellTileSchedule** objektum természetesen már nem elérhető. Ilyenkor egyszerűen újra létrehozzuk és elindítjuk az időzítést, majd rögtön le is állítjuk. Mivel egyszerre csak egy **ShellTileSchedule** lehet aktív lapkánként, ezzel felülírjuk a korábbi időzítést, a frissen létrehozott példányt pedig leállítjuk, azaz nem lesz aktív időzítés.

```
ShellTileSchedule schedule = new ShellTileSchedule();
schedule.Recurrence = UpdateRecurrence.Interval;
schedule.Interval = UpdateInterval.EveryHour;
schedule.MaxUpdateCount = 0;
schedule.RemoteImageUri = new
Uri("http://www.weather.gov/forecasts/graphical/images/conus/MaxT1_conus.png");
schedule.StartTime = DateTime.Now;
schedule.Start();

schedule.Stop();
```

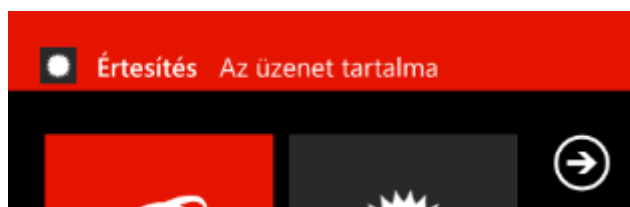
## A felugró értesítések

A fejezetben idáig alaposan megismerkedtünk a lapkák és különböző frissítési lehetőségeikkel. Amint a bevezetőben arról már szó volt, a lapkákon kívül a felugró értesítések is fontos módjai a felhasználóval való kapcsolattartásnak. A lapkák passzívak, használatukhoz a felhasználónak ki kell tűznie alkalmazásunkat a kezdőképernyőre, és a lapkákban található információ csak akkor jut el hozzá, ha később rájuk is néz. A felugró értesítések (angolul: *toast notification*) ezzel szemben aktívak! Bármilyen alkalmazást is futtat a felhasználó, az elküldött értesítés mindenképpen megjelenik a telefon kijelzőjének



felső részén, rányomva pedig egyből a hozzá tartozó alkalmazás indul be. Természetesen óvatosan kell bánnunk az értesítésekkel – ha túl agresszívan, felesleges üzenetekkel bombázzuk a felhasználót, lehetősége van letiltani alkalmazásunk értesítéseit. Mértékkel használva viszont hatékonyan hívhatjuk fel alkalmazásunkra a figyelmet.

A megjelenő értesítések bal oldalán alkalmazásunk ikonjának kicsinyített változata található. Az alkalmazás ikonja természetesen változtatható (erről lásd a korábbi fejezeteket), de az értesítésekhez nem állítható be külön ikon. Az ikon melletti félkörvén szöveg az értesítés címe, majd ezt követi az értesítés tartalma. Ez a két szöveg már állítható, ám összesen kb. 40-50 karakter jelenik meg, a többi kilóg a képernyőről és nem látható. Végül a lapkákhoz hasonlóan az értesítéseknél is használhatjuk a Deep Linking technikát, így egy URI-t is beállíthatunk az értesítés mögötti tartalom eléréséhez (10-18 ábra).



10-18 ábra: Egy felugró értesítés

Az értesítések fontos megköötése, hogy csak alkalmazásunkon *kívül* jelennek meg. Azaz ha fut az alkalmazásunk, nem tudunk értesítést küldeni. Ennek megfelelően csak Background Agent-ből (illetve a fejezet későbbi részében ismertetett Push Notification szolgáltatás segítségével) van lehetőségünk megjeleníteni egyet!

Az értesítések kezeléséhez szükséges kód igen egyszerű. A lapkáknál már részletesen megismertük a Background Agent-ek készítését. Végezzük el az ott leírt lépéseket (azaz hozzunk létre egy Windows Phone Application projektet, vegyünk fel mellé egy Windows Phone Scheduled Task Agent projektet, adjunk hozzá erre egy referenciát az első projektből stb.), majd a Scheduled Task Agent projektben található **ScheduledAgent.cs** fájl **OnInvoke()** metódusában helyezzük el a következő kódot:

```
ShellToast toast = new ShellToast();
toast.Title = "Értesítés";
toast.Content = "Az üzenet tartalma";
toast.Show();
```

A **toast** objektumnak van még egy **NavigationUri** tulajdonsága is. Ha ennek értékét nem adjuk meg, akkor az értesítésre nyomva a felhasználó alkalmazásunk kezdőképernyőjét látja majd megjeleníteni. Ha viszont szeretnénk használni a Deep Linking lehetőséget, akkor a lapkáknál leírtakkal azonos módon itt is megtehetjük.

Ne feledjük el befejezni a lapkáknál megismert műveleteket, azaz írjuk meg a Background Agent-et beregisztráló kódot is!

Miután ezzel elkészültünk, indítsuk el az alkalmazásunkat, regisztráljuk be a Background Agent-et, majd a telefon Windows gombjának segítségével lépünk ki az alkalmazásból (ez fontos, különben az értesítés nem lesz látható)! Néhány másodperc elteltével megjelenik az értesítés.

## A Push Notification szolgáltatás

A fejezet eddigi részében megismertük a Background Agent szolgáltatást és a **ShellTileSchedule** osztályt. A Background Agent szolgáltatással tetszőleges kódot futtathatunk a háttérben. Így kommunikálhatunk szerverrel, frissíthetünk lapkákat, és megjeleníthetünk értesítéseket. A **ShellTileSchedule** osztállyal pedig telefonunk időzítetten kérhet adatokat egy szervertől egy lapka frissítéséhez. Azaz két módszert is láttunk már a szerveroldali kommunikációra.

Viszont mindkét módszer csak kérni tud adatokat (azaz *pull* irányban működik). Így alkalmazásaink – és a felhasználók is – a különféle eseményekről óhatatlanul késéssel értesülnek, hiszen csak bizonyos

időközönként kapnak frissítést. Ráadásul, ha sokáig nem történik semmi, akkor felesleges kérésekkel fogyasztják a telefon akkumulátorát.

A *Push Notification* mechanizmus segítségével az alkalmazásunkat futtató telefont összekapcsolhatjuk egy általunk fenntartott szerveroldali alkalmazással. Az összekapcsolás után kiszolgálónk bármikor küldhet a telefonnak üzeneteket. Ez *push* irányú forgalom lesz, azaz nem alkalmazásunknak kell kérésekkel bombáznia a szervert, hanem a szerver üzenhet valamilyen esemény megtörténtekeor.

Ebben a fejezetrészen a Push Notification szolgáltatás tulajdonságaival, használatával ismerkedünk meg.

### ***A Push Notification szolgáltatás működése***

A Push Notification szolgáltatás architektúrája 3 elemből áll: az általunk írt telefonalkalmazásból, a szintén általunk írt szerveroldali alkalmazásból és a központi Microsoft Push Notification Service-ből (MPNS). Használata a következőképpen néz ki:

1. Alkalmazásunk jelzi a telefon felé, hogy szeretne push üzeneteket fogadni.
2. A telefon operációs rendszerének részét alkotó Push kliens az interneten át közli ezt a szándékot az MPNS-szel. Az MPNS egy publikus URL-t állít elő, és vállalja, hogy az ide érkező üzeneteket továbbítja a telefonnak. Ezt az URL-t pedig visszaküldi a Push kliensnek, aki visszaadja az alkalmazásunknak.
3. Alkalmazásunknak ezt az URL-t közölnie kell a szerverünkkel. Erre tetszőleges kommunikációs forma használható. Érdeemes webszolgáltatásokkal megoldani, amikről a 9. fejezetben bővebben esik szó.
4. Ha szerverünk ezután bármikor üzeni szeretne a telefonnak, akkor erre az URL-re küldi el üzenetét. Ezt az MPNS kapja meg, és továbbítja a készülék felé.

A szerveroldali komponens platformfüggetlen elemeket használ (XML formátumú üzeneteket kell küldenie), így tetszőleges szerveroldali technológiával megírható. Ebben a fejezetrészen ASP.NET nyelven programozzuk majd.

A Microsoft Push Notification Service ingyenes mind a fejlesztők, mind az alkalmazás-felhasználók számára. Használata azonban az alábbi feltételekhez kötött:

- A Push Notification-öket a felhasználónak kimondottan kérnie kell az alkalmazás felületén található valamilyen vezérlőelemmel (*opt-in* rendszer). Ezenkívül lehetőséget kell neki biztosítani az értesítések kikapcsolására is. (Ha ezeknek a feltételeknek nem teszünk eleget, az alkalmazást visszautasíthatják a Marketplace-ből.)
- A fentebb leírt regisztrációs lépések egy csatornát hoznak létre az MPNS és a telefon között. Telefononként maximum 30 ilyen csatorna lehet, azaz fel kell rá készülnünk, hogy a korábban telepített alkalmazások már mindet lefoglalták, és a mi alkalmazásunknak nem jut hely.
- Alkalmazásonként maximum 1 csatorna létezhet. Ez a korlátozás szerencsére nem súlyos, mert az alkalmazásunk előző futásaikor létrehozott csatornát lehetőségünk van újrahasznosítani.
- Szerverünk és az MPNS között alapértelmezésként HTTP (titkosítatlan) csatornán folyik a kommunikáció. Ezt egyszerű használni, de telefononként (csatornánként) és naponta maximum 500 üzenetet küldhetünk ki. Ha ennél többre van szükségünk, akkor egy HTTPS tanúsítványt kell vásárolnunk, és hitelesített csatornát kell létrehoznunk. Ez a korlátozás megint csak nem súlyos: napi 500 üzenet azt jelenti, hogy akár 3 percenként küldhetünk egy üzenetet a nap minden órájában.

A hitelesített csatornák létrehozására és a hozzájuk kapcsolódó „callback” üzenetküldésre ebben a fejezetben nem térünk ki. A szolgáltatás leírása megtalálható az internetes MSDN könyvtárban.

A Push Notification szolgáltatáson keresztül háromféle üzenetet küldhetünk a telefonnak:

- **Lapkafrissítés:** a fejezetben már megismertük a lapkák különféle tulajdonságait – előlő és hátsó felület, háttérkép, cím, stb. Egy Push üzenetből a lapka valamennyi tulajdonságát frissíthetjük. A frissítéshez nem kell futnia az alkalmazásunknak; az üzenetet a telefon operációs rendszerében lévő Push kliens fogadja, és ő végzi el a módosítást.
- **Felugró értesítés:** a Push Notification-ök segítségével felugró értesítéseket is megjeleníthetünk a felhasználó számára. A korábban megismert tulajdonságok itt is használhatók: beállíthatjuk az értesítés címét, tartalmát, és még egy Deep Linking-re alkalmas URL-t is átadhatunk. Ha alkalmazásunk nem fut, akkor megjelenik az értesítés, ha pedig fut, akkor kódból elkaphatjuk az üzenetet, és tetszésünk szerint feldolgozhatjuk.
- **„Raw” (nyers) üzenet:** Minden más esetre a *Raw Notification* használható. Nevéhez hűen ebben tetszőleges adatokat (szöveget vagy akár egy bájtömböt) is elküldhetünk a telefonnak. Ha alkalmazásunk fut, akkor megkapja. Ha nem fut, akkor viszont az üzenet elvész.

A fenti üzenettípusok mindegyikének rögzített formátuma van. A fejezet hátralévő részében egy példán keresztül megismerjük, hogy miként tudjuk regisztrálni alkalmazásunkat az MPNS-ben, majd hogyan küldhetjük ki a különféle üzenettípusokat.

## Lapkák frissítése Push Notification-ök segítségével

Ismét egy példaalkalmazást készítünk, amit később továbbfejlesztünk majd az értesítések és a raw üzenetek küldésére és fogadására. Az alkalmazásnak két szükséges komponense van: egy kliens, ami a telefonon fut majd, és egy szerver, amit éles környezetben egy webszerveren vagy a felhőben tartunk, ebben a példában azonban a helyi fejlesztői gépünkről szolgáljuk ki.

A webszerver elkészítéséhez erre alkalmas Visual Studio változatra van szükségünk. Ha a telepített Visual Studio nem teszi lehetővé a webfejlesztést, akkor töltsük le és telepítsük az ingyenes Visual Web Developer Express verziót!

Hozzunk létre egy új Windows Phone Application projektet **PushNotificationClientSample** néven! Továbbra is a Windows Phone OS 7.1 vagy újabb verziót válasszuk! Első lépésként megírjuk azt a kódot, ami beregisztrálja alkalmazásunkat az MPNS felé. Az egyszerűség kedvéért ezt már közvetlenül az alkalmazás indulásakor lefuttatjuk; éles alkalmazásban ne feledjük ezt valamilyen vezérlőelemhez (például egy gombhoz) kötni!

Elsőként írjuk a következő sort a fájl legtetetejére:

```
using Microsoft.Phone.Notification;
```

Mivel alkalmazásonként maximum 1 csatornánk lehet az MPNS felé, ezért feltétlenül meg kell vizsgálnunk, hogy alkalmazásunk korábbi futtatásakor regisztrált-e már be csatornát. Ehhez a **HttpNotificationChannel** objektumot és annak metódusait használjuk majd.

Ha azt találjuk, hogy a csatorna még nincs beregisztrálva, akkor ezt megteesszük. A beregisztrált csatorna egyrészt különféle eseményeket indíthat (például: hiba történt, vagy megváltozott az MPNS által biztosított URL), amelyekre fel kell iratkoznunk; másrészt közölnünk kell a telefonnal, hogy a csatorna jogosult a lapkák frissítésére! A mintakódban láthatóak lesznek az esemény-feliratkozások, és a csatorna lapkafrissítésre való regisztrációja is (**BindToShellTile()** hívás).

Ha viszont korábban már megnyitottuk a csatornát, akkor nincs más dolgunk, mint újra feliratkozni annak eseményeire. Az eredeti eseményfeliratkozások az alkalmazáson belül történtek, így természetesen az alkalmazás leállításakor azok elvesznek.

A fentiek elvégzéséhez helyezzük az alábbi kódot a **MainPage.xaml.cs** kódfájl **MainPage()** konstruktorába:

```
HttpNotificationChannel pushChannel;
string channelName = "SampleChannel";

//Megnézzük, hogy létezik-e már a csatorna.
pushChannel = HttpNotificationChannel.Find(channelName);

//Még nem létezik - létrehozuk, és feliratkozunk az eseményeire.
if (pushChannel == null)
{
    pushChannel = new HttpNotificationChannel(channelName);
    pushChannel.ChannelUriUpdated += new
        EventHandler<NotificationChannelUriEventArgs>(pushChannel_ChannelUriUpdated);
    pushChannel.ErrorOccurred += new
        EventHandler<NotificationChannelErrorEventArgs>(pushChannel_ErrorOccurred);
    pushChannel.Open();

    pushChannel.BindToShellTile();
}
//Már létezik - ismét feliratkozunk az eseményeire.
else
{
    pushChannel.ChannelUriUpdated += new
        EventHandler<NotificationChannelUriEventArgs>(pushChannel_ChannelUriUpdated);
    pushChannel.ErrorOccurred += new
        EventHandler<NotificationChannelErrorEventArgs>(pushChannel_ErrorOccurred);

    System.Diagnostics.Debug.WriteLine(pushChannel.ChannelUri.ToString());
}
```

A fenti kód két eseménykezelőre is hivatkozik. Az első (**pushChannel\_ChannelUriUpdated**) akkor kerül meghívásra, ha az MPNS megváltoztatja a csatorna URL-jét. Éles alkalmazásban ezt célszerű közölnünk a webszerverünkkel; itt mindössze egy erre vonatkozó üzenetet íratunk ki a Visual Studióval. A második (**pushChannel\_ErrorOccurred**) pedig hibakezelésre való; ebben a kódban a hibákról üzenetet küldünk a felhasználónak.

```
void pushChannel_ChannelUriUpdated(object sender, NotificationChannelUriEventArgs e)
{
    Dispatcher.BeginInvoke(() =>
    {
        System.Diagnostics.Debug.WriteLine(e.ChannelUri.ToString());
    });
}

void pushChannel_ErrorOccurred(object sender, NotificationChannelErrorEventArgs e)
{
    Dispatcher.BeginInvoke(() =>
    {
        MessageBox.Show("Hiba történt: " + e.Message);
    });
}
```

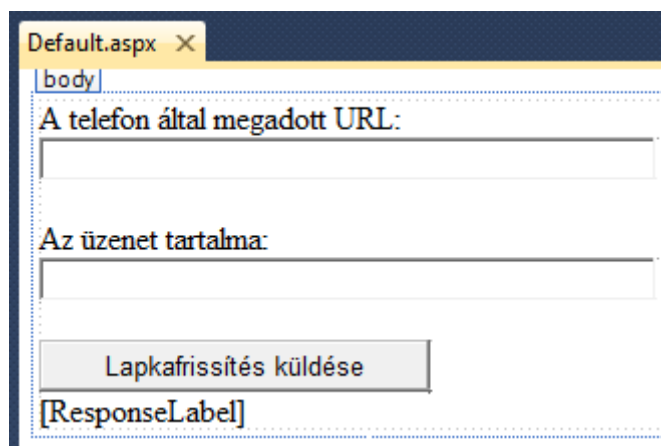
Ezzel a kliensoldali kódot meg is írtuk – az alkalmazás vagy új csatornát hoz létre, vagy újrahazsnosítja a régit, és lekezezi annak különféle eseményeit. Látszólag hiányzik viszont egy fontos elem: nem küldjük el az URL-t a szerveroldalra! Egy éles alkalmazásban valóban így történne, például egy webszolgáltatáson keresztül, amiről a 9. fejezetben részletesen olvashatunk. Most az egyszerűség kedvéért mindössze kiíratjuk az URL-t, és a vágólapon keresztül másoljuk át a szerverre.

Írjuk most meg az alkalmazás szerveroldali komponensét! Ehhez nyissunk egy új Visual Studio projektet!

Ezt létrehozhatjuk a már megnyitott Visual Studio példányban is, de ha a Visual Studio Express verziót használjuk, akkor új példányt kell indítanunk, mert a telefonos fejlesztésre használt Visual C# Express nem támogatja a webfejlesztést, ehhez a Visual Web Developer Express-re van szükségünk.

A projekt típusa legyen ASP.NET Empty Web Application, neve **PushNotificationServerSample**!

A projekt létrejötte után adjunk ahhoz egy új Web Form típusú elemet, **Default.aspx** néven! A Visual Studio automatikusan megnyitja az űrlap HTML nézetét. Hozzunk létre az űrlapon két szövegmezőt **UrlTextBox** és **MessageTextBox** néven, egy gombot **SendTileButton** néven és „Lapkafrissítés küldése” felirattal, valamint egy címkét **ResponseLabel** néven! Az első szövegdobozba kerül majd az MPNS által visszaadott URL, ahová üzenetünket küldjük majd. A második szövegdobozba írjuk üzenetünket, amit megjelenítünk majd a lapkán! A gombbal kezdeményezzük a küldést, a címkébe pedig a visszatérési értéket helyezzük majd el. A fentiek elvégzése után az űrlapnak a 10-19 ábrához kell majd hasonlítania.



**10-19 ábra: A Default.aspx oldal a vezérlőelemekkel**

Következik a szerveroldali kód megírása. A telefontól megszokott módon kattintsunk duplán a Lapkafrissítés küldése gombra! A megnyíló kódfájl tetejére helyezzük el a következő néhány sort:

```
using System.IO;
using System.Net;
using System.Text;
```

Ezután írjuk meg a gomb **SendTileButton\_Click** eseménykezelőjének kódját! Mindhárom értesítési típusnak egy kötött struktúrájú XML üzenetet kell küldenie egy HTTP kérésen keresztül, kódunk ezért először felépíti ezt az XML struktúrát. Amint a példakódban levő XML-ből látjuk, a lapkák korábban megismert valamennyi tulajdonságát frissíthetjük. Ha valamelyik XML tulajdonságot üresen hagyjuk, akkor az nem változik a lapkán. A **ShellTileSchedule** osztállyal ellentétben itt háttérképeket helyi erőforrásból is beállíthatunk – ehhez használjunk relatív URL-t (amire már láttunk példát a fejezet korábbi részeiben)!

A példakód az egyszerűség kedvéért kizárólag a lapka címét frissíti a megadott üzenetre, a többi tulajdonságot változatlanul hagyja.

```
//Előállítjuk az üzenetet.
string tileMessage =
"<?xml version=\"1.0\" encoding=\"utf-8\"?>" +
"<wp:Notification xmlns:wp=\"WPNotification\">" +
"  <wp:Tile>" +
"    <wp:BackgroundImage></wp:BackgroundImage>" +
"    <wp:Count></wp:Count>" +
"    <wp:Title>" + MessageTextBox.Text + "</wp:Title>" +
"    <wp:BackBackgroundImage></wp:BackBackgroundImage>" +
```

```
"<wp:BackTitle></wp:BackTitle>" +  
"<wp:BackContent></wp:BackContent>" +  
"</wp:Tile> " +  
"</wp:Notification>";
```

A fenti kód Windows Phone 7.5 (Mango) vagy újabb operációs rendszer esetén működőképes. Bizonyos tulajdonságok még nem léteztek a Windows Phone 7.0 operációs rendszerben. Ha ehhez írunk programot, akkor ezeket hagyjuk ki! Ezekről lásd: [http://msdn.microsoft.com/en-us/library/ff402558\(VS.92\).aspx](http://msdn.microsoft.com/en-us/library/ff402558(VS.92).aspx)

Az előállított XML fájlt egy HTTP kérésen keresztül tudjuk elküldeni az MPNS-nek. Erre a **HttpRequest** osztályt használjuk. Az üzenetet bájttömb formájában kell majd átküldeni a hálózaton, így a korábban létrehozott XML formátumú szöveget bájttömbbé alakítjuk.

A **HttpRequest** paraméterezésére is ügyelnünk kell! Egyrészt, a kérést POST üzenet formájában kell elküldeni. Másrészt, a HTTP fejrészében be kell állítanunk, hogy az üzenet milyen célt szolgál, és mennyire sürgős. Az **X-WindowsPhone-Target** tulajdonság értéke adja meg, hogy lapkafrissítésről, értesítésről vagy raw üzenetről van szó. A **X-NotificationClass** pedig az üzenet sürgősségét határozza meg – ha 1, akkor azonnal, ha 11, akkor legkésőbb 450 másodpercen belül, ha 21, akkor pedig legkésőbb 900 másodpercen belül továbbküldi a telefon felé az MPNS.

```
//Létrehozuk a HTTP kérést.  
HttpRequest sendNotificationRequest = (HttpRequest)WebRequest.Create(UriTextBox.Text);  
byte[] notificationMessage = Encoding.Default.GetBytes(tileMessage);  
sendNotificationRequest.Method = "POST";  
sendNotificationRequest.ContentLength = notificationMessage.Length;  
sendNotificationRequest.ContentType = "text/xml";  
sendNotificationRequest.Headers.Add("X-WindowsPhone-Target", "token");  
sendNotificationRequest.Headers.Add("X-NotificationClass", "1");
```

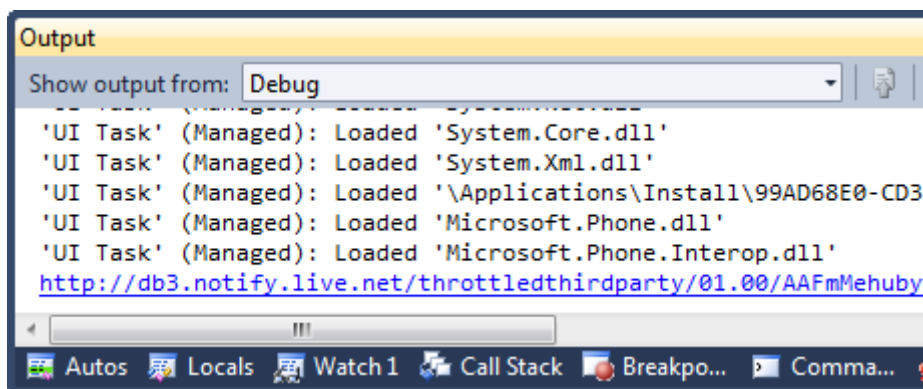
Ezek után már nincs más hátra, mint az üzenet tényleges elküldése. Az MPNS visszajelez, miután elküldte az üzenetet; a különféle státusz-üzenetek a válasz HTTP fejlécében érkeznek. Ezt rögtön ki is íratjuk az erre a célra létrehozott **ResponseLabel** címkébe.

```
//Elküldjük az üzenetet és kiolvassuk a választ.  
using (Stream requestStream = sendNotificationRequest.GetRequestStream())  
{  
    requestStream.Write(notificationMessage, 0, notificationMessage.Length);  
}  
HttpWebResponse response = (HttpWebResponse)sendNotificationRequest.GetResponse();  
string notificationStatus = response.Headers["X-NotificationStatus"];  
string notificationChannelStatus = response.Headers["X-SubscriptionStatus"];  
string deviceConnectionStatus = response.Headers["X-DeviceConnectionStatus"];  
ResponseLabel.Text = notificationStatus + " | " + deviceConnectionStatus + " | " +  
notificationChannelStatus;
```

Mivel hálózati kódról van szó, ezért érdemes a fentieket körbevenni egy **try..catch** blokkal.

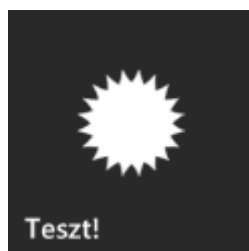
Ezzel elkészültünk. Teszteljük a programot! Indítsuk el mindkét projektet! A Windows Phone alkalmazás látszólag semmit nem csinál, de valójában elvégzi az MPNS regisztrációt, és az eredményül kapott URL-t kiírja a Visual Studio Debug üzenetei közé. Ezeket a fejlesztő felület Output ablakában nézhetjük meg. Ha ez nem látszik, akkor a View menü Output menüpontjával tegyük láthatóvá (lásd 10-20 ábra)!





**10-20 ábra: Az Output ablak és az MPNS-től visszakapott URL**

A megjelenő URL-t másoljuk ki, és illesszük be a weboldalunk felső szövegdobozába! Az alsóba írunk valamilyen rövid üzenetet! A telefonon lépünk ki az alkalmazásból, és tűzzük ki a kezdőképernyőre. Végül térjünk vissza a webalkalmazáshoz, és kattintsunk a Lapkafrissítés küldése gombra! A webalkalmazás összeállítja az XML-t, elküldi az MPNS-nek, ami rögtön továbbítja azt a telefon felé. Ennek hatására a frissítés gyakorlatilag azonnal megjelenik, amint az a 10-21 ábrán látható.



**10-21 ábra: A Push Notification-nel frissített lapka**

Jó tudni, hogy míg a ShellTile API-n keresztül elvégzett változtatásokat akkor is elmentette a telefon, ha az alkalmazás lapkája nem volt kitűzve, a Push Notification-öknél ez nincs így – ha az alkalmazás épp nincs kitűzve, az üzenet elvész. Szintén fontos, hogy a Push Notification-ök sajnos nem minden karaktert támogatnak. Hibát kaphatunk, ha ékezetes betű van az üzenetben.

## **Értesítések megjelenítése Push Notification-ök segítségével**

Nézzük, hogyan tudunk felugró értesítéseket megjeleníteni Push Notification-ök segítségével! A művelet logikailag szinte teljesen megegyezik a lapkafrissítésekkel – a telefonon beregisztráljuk a csatornát, elküldjük az URL-t a szervernek, a szerverről pedig XML formátumú kéréseket irányítunk a csatornába.

Ennek megfelelően a kliensoldali kódban minimális változtatást kell csak végrehajtanunk. Mindössze azt kell közölnünk a telefonnal, hogy a létrehozott csatornán keresztül értesítések is érkezhettek, és ezeket is kezelje le. Írjuk be a `pushChannel.BindToShellTile();` sor alá a következő egyetlen sort:

```
pushChannel.BindToShellToast();
```

A szerveroldalon szintén nagyon kevés változtatásra van szükségünk. A felugró üzenetet és a lapkafrissítést küldő kód csak az elküldött XML fájl formátumában és néhány paraméter-értékben tér el. Hozzunk létre egy új gombot a webes projektben `SendToastButton` néven és „Értesítés küldése” felirattal! Dupla kattintással lépünk be `Click` eseménykezelőjébe, és egy az egyben másoljuk át a `SendTileButton_Click` eseménykezelő tartalmát ide!

Módosítsuk az XML üzenetet a felugró üzenetek sémájának megfelelően! Láthatjuk, hogy ez az XML is tartalmazza a felugró értesítések valamennyi lehetséges tulajdonságát. A lapkákhöz hasonlóan itt is

ügyeljünk, mert a 7.0-s operációs rendszert futtató telefonok még nem ismerik mindet az itt szereplő tulajdonságok közül!

```
//Előállítjuk az üzenetet.  
string tileMessage =  
"<?xml version=\"1.0\" encoding=\"utf-8\"?>" +  
"<wp:Notification xmlns:wp=\"WPNotification\">" +  
  "<wp:Toast>" +  
    "<wp:Text1>" + MessageTextBox.Text + "</wp:Text1>" +  
    "<wp:Text2></wp:Text2>" +  
    "<wp:Param>/MainPage.xaml</wp:Param>" +  
  "</wp:Toast> " +  
"</wp:Notification>";
```

Ezután módosítsuk a kérés HTTP fejrészét! Meg kell adnunk, hogy ez egy értesítés, azaz „toast” típusú, és át kell írunk a kézbesítés gyorsaságát szabályozó konstanst is (itt 2 jelenti az azonnali kézbesítést, 12 a 450 másodpercen belülit, és 22 a 900 másodpercen belülit).

```
sendNotificationRequest.Headers.Add("X-WindowsPhone-Target", "toast");  
sendNotificationRequest.Headers.Add("X-NotificationClass", "2");
```

Miután ezzel elkészültünk, már tesztelhetjük is az alkalmazást. A teszteléshez előbb el kell távolítani a kliens alkalmazást a telefonról. Erre azért van szükség, mert korábbi tesztjeink során az már megnyitotta a Push Notification csatornát. Így ha most elindítjuk, akkor csak újrahajszosítja a csatornát, és nem fut le a módosított kódunk, ami lapkafrissítéshez is beregisztrálná (a **BindToShellToast()** hívás). Távolítsuk el tehát a programot a telefonról, majd indítsuk el a két projektet! A telefonalkalmazásból lépünk ki (különben az értesítés nem jelenik meg), majd másoljuk át az URL-jét a webalkalmazásba, és kattintsunk az Értesítés küldése gombra! Az eredmény a 10-22 ábrához hasonlít majd.



10-22 ábra: A megjelenő felugró üzenet

Ha az alkalmazás futása közben is szeretnénk kezelni a megérkező értesítéseket, akkor ehhez a **ShellToastNotificationReceived** eseményre kell feliratkoznunk.

### **Raw üzenetek fogadása a Push Notification szolgáltatáson keresztül**

Végül ismerkedjünk meg a Raw üzenetek küldésével! A felugró értesítésekhez hasonlóan nagyon kevés változtatást kell végeznünk már meglévő kódunkon. A raw üzenetek csak akkor fogadhatók, amikor alkalmazásunk fut, így ezúttal egy eseményre kell majd feliratkoznunk. Helyezzük a következő kódot **mindkét pushChannel.ErrorOccurred +=** kezdetű sor után:

```
pushChannel.HttpNotificationReceived += new  
EventHandler<HttpNotificationEventArgs>(pushChannel_HttpNotificationReceived);
```

A **HttpNotificationReceived** eseménykezelő törzsében mindössze kiolvassuk és megjelenítjük a kapott üzenetet – egy éles alkalmazás természetesen ennél sokkal bonyolultabb logikát is végrehajthat:



```

void pushChannel_HttpNotificationReceived(object sender, HttpNotificationEventArgs e)
{
    using (System.IO.StreamReader reader = new System.IO.StreamReader(e.Notification.Body))
    {
        string message = reader.ReadToEnd();

        Dispatcher.BeginInvoke(() =>
        {
            MessageBox.Show(message);
        });
    }
}

```

Most módosítsuk a serveroldalt is! Hozzunk létre egy új gombot **SendRawButton** néven „Raw üzenet küldése” felirattal, lépünk be **Click** eseménykezelőjébe, majd másoljuk át ide a **SendToastButton\_Click** eseménykezelő törzsét! Ezt ismét csak két helyen kell módosítanunk.

Elsőként írjuk be az elküldendő üzenetet! Ez raw üzenet esetén bármi lehet – XML, szöveg vagy akár egy tetszőleges bájtömb is. Ügyeljünk azonban arra, hogy az MPNS maximum 1 kilobájtnyi fejléccel és 4 kilobájtnyi törzssel rendelkező üzeneteket továbbít!

```

//Előállítjuk az üzenetet.
string tileMessage = MessageTextBox.Text;

```

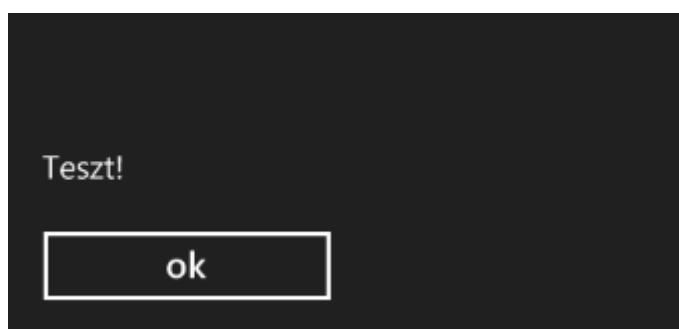
Másodszor, módosítsuk a HTTP fejléceket! Az **X-WindowsPhone-Target** fejlécre ezúttal nincs szükségünk – ezt a lenti kódrészletben kommentálással jeleztük. Az **X-NotificationClass** értéke raw üzenet esetén 3, 13 vagy 23 lehet – a megszokott módon ez azonnali, 450 másodpercen belüli vagy 900 másodpercen belüli kézbesítést jelent.

```

//sendNotificationRequest.Headers.Add("X-WindowsPhone-Target", "toast");
sendNotificationRequest.Headers.Add("X-NotificationClass", "3");

```

Készen állunk az alkalmazás tesztelésére. A felugró üzenetknél látott módon távolítsuk el az alkalmazást a telefonról, majd indítsuk be a két projektet, másoljuk át az URL-t a webalkalmazásba, és küldjünk egy raw üzenetet! A várható eredmény a 10-23 ábrán látható.



10-23 ábra: A kézbesített raw üzenet

## Push Notification-önök és a Windows Azure

Ha szerverre van szükségünk, kézenfekvő, hogy a felhőt használjuk erre a célra. Jó hír, hogy a Windows Azure kifejezetten támogatja a Windows Phone-nal való együttműködést. A Microsoft külön toolkit-et készített Azure Toolkit for Windows Phone néven, ami kész alkalmazás-sablonokat tartalmaz. Ezek képesek az Azure tárhelyek használatára, vagy az Azure-on keresztüli felhasználó-hitelesítésre, és ami a leglényegesebb: a Push Notification-ök használatára is. Architektúráis szempontból ugyanazt valósítják

meg, mint a fentebb leírtak, de kész, érett kód formájában. Ha lehetőségünk van az Azure használatára, érdemes kipróbálni a toolkit-et, amelynek beállítását, egyes funkcióinak használatát a 9. fejezet tárgyalja.

## Összefoglalás

A fejezetben megismertük a lapkák tulajdonságait, és láttuk a frissítésükre használható négy módszert. Ezek célja ugyan azonos, de más-más esetben érdemes őket használni. Az alábbi táblázatban összehasonlítjuk a lapkák frissítésére használható lehetőségeket.

Szolgáltatás	Jellemzők
ShellTile API	Csak a telefon kell hozzá. Alkalmazásunk futása alatt használható.
Background Agent	Csak a telefon kell hozzá. Alkalmazásunknak nem kell futnia, a frissítés időzítetten történik.
ShellTileSchedule	Szervert is igényel. Alkalmazásunknak nem kell futnia, időzített módon, a telefon kezdeményezésével működik.
Push Notification	Szervert is igényel. Alkalmazásunknak nem kell futnia, eseményvezérelt, vagyis a szerver kezdeményezi a frissítést.

Láttuk továbbá a felugró értesítéseket is, melyekkel a felhasználó figyelmét aktívan irányíthatjuk alkalmazásunkra. Megismertük, hogy az értesítések a lapkákhoz hasonlóan Background Agent és Push Notification segítségével is küldhetők.

Ezen eszközökkel alkalmazásunk nemcsak addig „él”, amíg a felhasználó aktívan futtatja, hanem leállítása után is képes friss információkat biztosítani. Érdemes őket használni, mert segítségükkel sokkal professzionálisabb megjelenést, interaktívabb élményt biztosíthatunk.

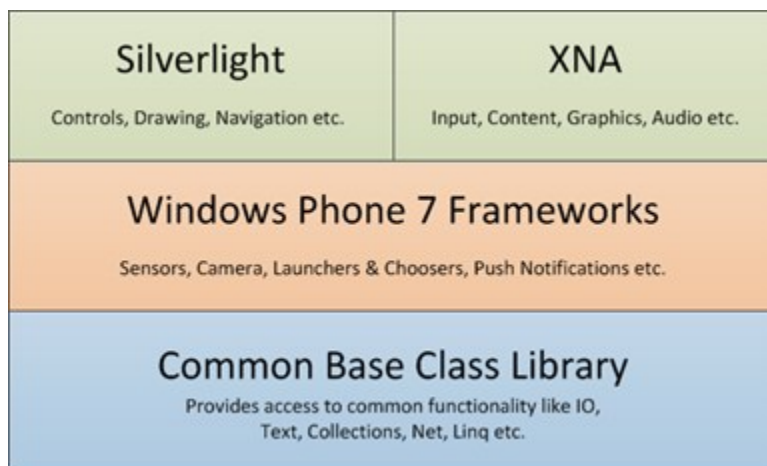
# 11. Játékok a Mango világában

A most soron következő fejezet témáját tekintve merőben eltér a korábbi fejezetektől, azonban úgy gondoltuk, hogy mindenképpen érdemes adni egy rövid ízelítőt a telefonos játékok fejlesztéséből, illetve bemutatni, hogy ezen a téren a Mango frissítés milyen újdonságokat hozott magával. A fejezet első részében megismerkedünk az XNA Framework és a grafikus programozás legfőbb mozzanataival, ezzel elméleti és gyakorlati alapot adva azon Olvasóinknak is, akik eddig még soha nem találkoztak fejlesztői szempontból számítógépes játékokkal vagy grafikus programozással. Ezek után megnézzük, pontosan mit is jelent a Silverlight és az XNA hibrid megjelenítési rendszere, és mikor tudjuk kihasználni az új lehetőségeket.

## Windows Phone – a motorháztető alatt

A Windows Mobile fejlesztése során a Microsoft rendkívül sok tapasztalatot halmozott fel, a fejlesztői modell előnyei és hátrányai jól körvonalazódtak. Amikor a platform utódjának fejlesztésébe kezdtek, egy olyan sokkal átgondoltabb és korszerűbb rendszert szerettek volna alkotni, amely a legújabb technológiákra épít, és az arra való áttérés egy .NET fejlesztő számára sokkal kevésbé megterhelő, mint azelőtt volt. Miután a teljes .NET Framework számos okból nem vihető át egy mobil eszközre, így a platform felépítésekor két fő szempontot kellett betartani: a megszokott alkalmazásfejlesztési modell alkalmazható legyen általános alkalmazásfejlesztéshez, de az okos telefonok piacán egyre nagyobb teret hódító játékok és grafikus alkalmazások se maradjanak támogatás nélkül. A Silverlight mint korszerű alkalmazásépítési platform jó választásnak tűnt, több éve kedvelt és elterjedt technológia, aránylag rövid kiadási ciklusokkal. Azonban hiába a gazdag médiatámogatás, a teljesítménye nem megfelelő sebességorientált magas számítási kapacitást igénylő megoldásokhoz – ide bizony egy grafikus technológiára volt szükség. PC-re írt játékok esetében két elterjedt grafikai API létezik: OpenGL és DirectX (pontosabban a Direct3D, amely a teljes platformnak csak egy darabja). Miután a Microsoft a nyílt OpenGL-t nem használja saját fejlesztéseihez, és a Windows a grafikus alrendszer támogatásán kívül mást nem nyújt, így a választás a Direct3D lehetett volna. Azonban a Windows Phone az ipar számára egy újonnan belépő tagot jelentett, így nemcsak a tapasztalt játékfejlesztőket kellett megcélozni a technológiai támogatásokkal, hanem a játékok fejlesztése iránt érdeklődő hobbi- és kezdő fejlesztőket is. Ők nagy valószínűséggel nem fognak egy komplex grafikus rendszert kitanulni, így sokkal kényelmesebb megoldásra volt szükség, amely magasabb absztrakciójú megoldásokat kínál, könnyebben tanulható és könnyebben kezelhető. A választás az *XNA Framework*re esett. Az XNA a Microsoft játékfejlesztési eszközkészlete és futtatókörnyezete különböző eszközökre, mint Windows, Xbox és most már a Windows Phone is. Ez azt jelenti, hogy a céleszközök között az alkalmazások átvihetők, skálázási eltérésekre és néhány, az eszközök közötti különbségekből származó eltérésre szükséges csak odafigyelni a játék fejlesztése során. Természetesen, ahogy azt megszoktuk, .NET alapú programozási felülettel rendelkezik, és tartalmaz mindent, amire egy játékprogram esetében szükség lehet (11-1. ábra):

- Átgondolt irány-architektúra a játék számára
- Grafikai rendszer
- Bemenetek támogatása magas szinten
- Audiorendszer
- Tartalmakat kezelő alrendszer
- Különböző eszközök



11-1. ábra: A Windows Phone Mango fejlesztői platform

Nem keverendő össze az XNA Framework egy játékmotorral (*game engine*)! Sok szempontból úgy tűnhet, hogy az XNA mindent tartalmaz, amit egy játékmotor, de ez messze nem igaz. Egy játékmotor sokkal specifikusabb, az esetek jelentős részében kötötten egy játékkategórián belül használható jól, illetve az számos olyan komponenst kínál (fizikai számításokért felelős komponens, mesterséges intelligencia, szkript rendszer, stb.), amelyeket az XNA nem. Az XNA egy jól átgondolt alapot ad, aminek támogatásával rövidebb idő alatt, egyszerűbben juthatunk el a kívánt célunkig. Ha nem szeretnénk minden apró részletet saját magunk kifejleszteni a legelejétől fogva, akkor az interneten számos nyílt forráskódú játékmotor fellelhető, amelyek kimondottan XNA-hoz készültek.

Fontos megjegyzés az XNA-val kapcsolatban: sokat egyszerűsít a játékfejlesztési folyamaton, azonban ezt azon az áron teszi, hogy soha nem a legfrissebb natív grafikus platformhoz van igazítva, ami a mi esetünkben a DirectX. Ha a legújabb funkciókat szeretnénk Windows környezetben kihasználni, akkor ahhoz DirectX-re és natív kódra lesz szükségünk, amely az esetek egy jelentős részében – a jól optimalizálhatóság miatt – C++-ban íródik. Windows Phone telefonokon azonban nincs lehetőségünk natív kódot futtatni.

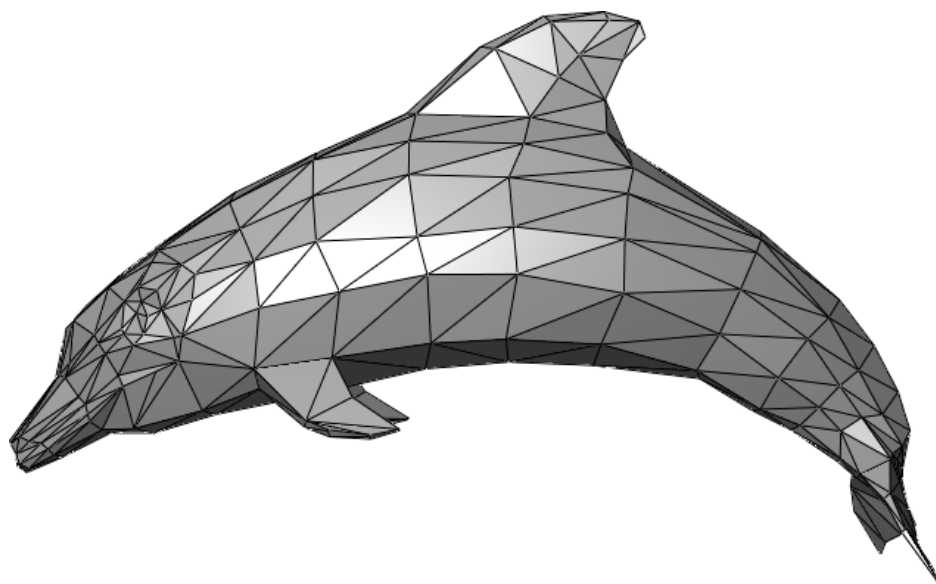
A keretrendszer pillanatnyilag a 4.0-s változatánál tart, amely külön is letölthető a Microsoft letöltő központjából, azonban szerves részét képezi a Windows Phone fejlesztői készletnek is.

## Grafikus programozási és XNA alapok

### A kép kialakítása

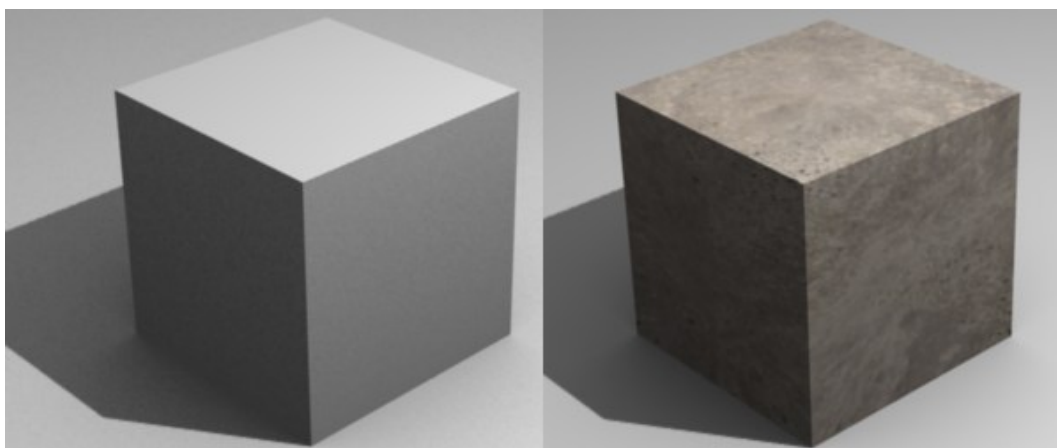
A mai játékok hónapról hónapra egyre szebb grafikai megvalósításokat sorakoztatnak fel, amely elsősorban a fejlődő hardveres erőforrásoknak köszönhető, másodsorban pedig a folyamatosan fejlődő technológiáknak. De milyen komponensekből épül fel egy játék grafikája? Nos, 3 dimenziós térben gondolkozunk, minden, amit meg szeretnénk jeleníteni, térben helyezkedik el. Képek reprezentációja során jól ismerjük a két létező analógiát, miszerint egy kép lehet pixeleivel számítva – amely az esetleges skálázások során pontatlanságot és darabosságot okoz, azonban gyorsan feldolgozható – és lehet vektoros alapon számítva – amely skálázás esetén is mindig pontos lesz, azonban a matematikai számításokat újra és újra végre kell hajtani. Ha ezt térbe szeretnénk levetíteni, akkor már előre megjósolható, hogy valamilyen gyorsító megoldásra itt is szükség lesz, nem fogjuk minden egyes pontját az adott térbeli testnek kiszámítani. Helyette poligonokra (sokszögekre) bontjuk fel az alakzatot, a poligonoknak pedig a csúcspontjait tároljuk le. Megjelenítéskor a csúcspontokat összekötjük, és a területüket kitöltjük, így a poligonok számának függvényében fog változni az adott modell részletessége is. Természetesen minél több poligonnal szeretnénk dolgozni, az annál több számítást fog igényelni. Amikor a grafikus processzor feldolgozza és megjeleníti ezeket a poligonvázakat, akkor először a térbeli információk kerülnek be a megjelenítéshez szükséges elemeket tartalmazó csővezetékbe. A GPU több

lépésen keresztül tovább bontja azokat háromszögekre, a keletkezett háromszögeket pedig raszterizálja, azaz levetíti a képernyő megfelelő pixeleire (11-2. ábra).



**11-2. ábra: Egyszerű poligonváz**

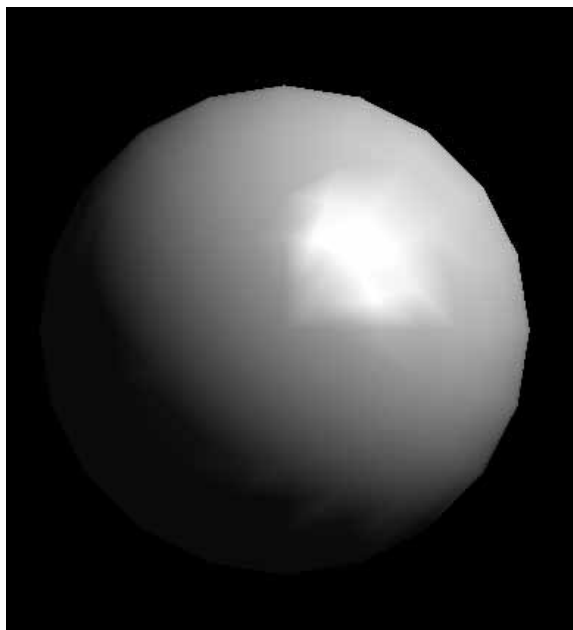
A 11-2. ábrán jól látható, hogy a térben megalkotott delfin alakzat nem igazán valósághű, alakja kezd hasonlítani arra, amit létre szeretnénk hozni, de részletessége még nagyon messze áll ettől. A számítógépes grafika minden területén a való életből merít ötleteket, ezért nem a poligonok számának növelésével és színezésével finomítjuk a modellt tovább, hanem anyagmintákat (textúrákat fogunk) felhasználni. Az anyagminták egyszerű képfájlok, melyeket térben a megfelelő poligonra vetítünk le, így azt a hatást keltve, mintha az adott test felületén helyezkedne el (11-3. ábra).



**11-3. ábra: Poligonváz textúra nélkül és textúrával**

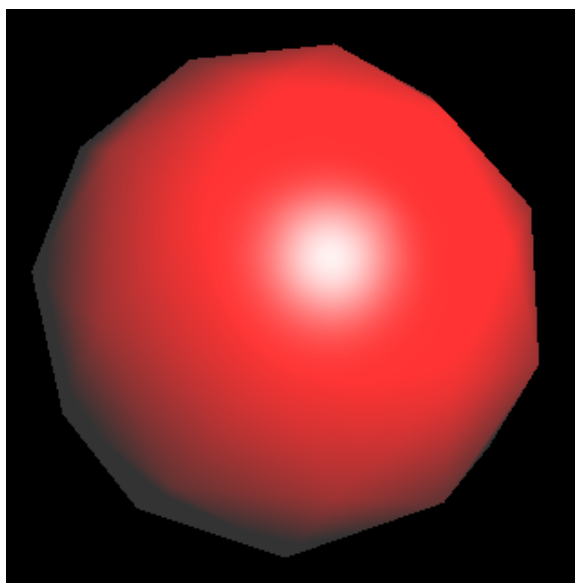
Most már olyan érzetet kelt bennünk, mintha tényleg valamilyen valós objektumot jelenítettünk volna meg egyszerűen a számítógép segítségével, de még mindig nem az igazi. A textúrák nemcsak egyetlen rétegben helyezkedhetnek el a modellen, hanem több különböző réteg több feladatot tud ellátni. Azon túl, hogy az alapmintázatot megadja, egy másik rétegben tárolhatunk olyan információkat is, amik arra vonatkoznak, mennyi fényt tud elnyelni az adott test helyenként, mennyire árnyalt, és így tovább. Elhangzott egy fontos fogalom: a fény. Rendkívül sokat tud a valóságérzet kialakításában segíteni egy jól elkészített fény-árnyék rendszer, azonban (mint minden más is egy játékban) ez matematikai alapon van megfogalmazva, és számításigényes lehet. Mi kell a fények megjelenítéséhez? Valamilyen fényforrás és az adott térbeli alakzat információja arra vonatkozóan, hogy ha egy fénysugár becsapódik, akkor azt hogyan verjük vissza és milyen intenzitással. A visszaverődéshez egy felületi tulajdonságra van szükségünk, ami

nem más, mint az adott poligon normálvektora. A normálvektor és a fényvektor skaláris szorzata meghatározza a visszaverődési paramétert.



**11-4. ábra: Polygononkénti megvilágítás**

A 11-4. ábrán jól megfigyelhető, hogy a polygononként meghatározott visszaverődés csak akkor ad szép eredményt, ha magas poligonszámú vázzal van dolgunk. Ennek lehetséges orvoslása, ha minden egyes pixelre külön határozzuk meg a fények hatását, ehhez azonban pixel alapú manipulációra van szükségünk, amelyet a grafikus technológiák nem segítenek alapértelmezetten. A beépített – és egyben korlátos – megoldások (Fixed Function Pipeline) megkerülésére aránylag hosszú ideje van lehetőség grafikus processzorra írt árnyaló (shader) kódok segítségével. Itt már csak a számítási komplexitás szabhat határt, amit megfogalmazunk, azt közvetlenül a GPU (grafikus processzor) fogja végrehajtani. Egy pixel alapú megvilágítást mutat be a 11-5. ábra.

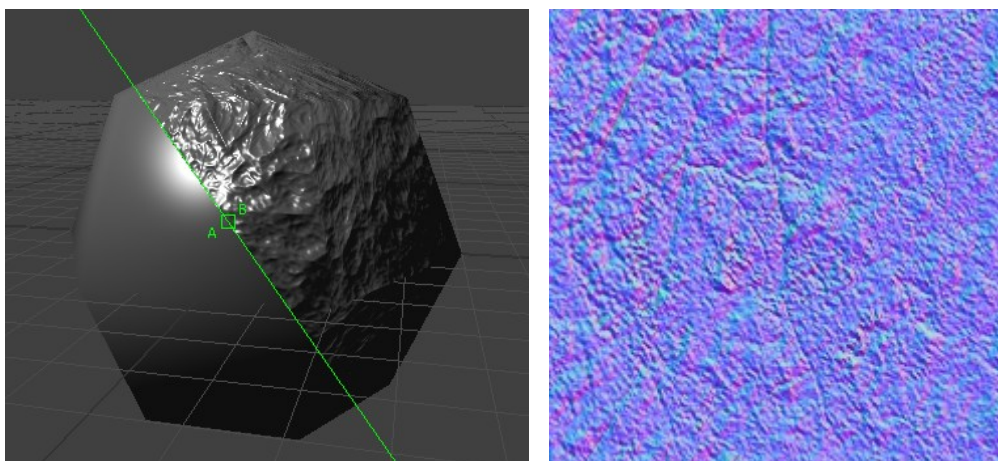


**11-5. ábra: Pixel alapú fény egy kis felbontású modellen**



Érdekesség: a fény-árnyék rendszerek nemcsak a számítógépes játékokban jelentősek, hanem a filmekben használt speciális effektusok nagy része is ezekre épít. Amit a fejezetben eddig tárgyaltunk, az a sebességorientált (inkrementális) képszintézis, azonban a tökéletes tükröződések, törések és megvilágítások reprezentációjához létezik egy sugárkövetés (*ray tracing*) nevű megközelítés is. Ennek a lényege az, hogy a képernyő minden egyes pixeléből sugarakat indítunk – akárcsak a való életben a fotonok - a virtuális térbe, melyeknek az útját követjük, és megfigyeljük, hogy milyen anyagi tényezőket gyűjtöttek össze, ezzel a matematikailag tökéletes világitást megvalósítva. Természetesen vannak kezdeményezések megfelelő sebességű algoritmusok előállítására, amelyek játékokban valós időben tudnak megvalósítani sugárkövetést, azonban a komplex számítási rendszer miatt a komolyabb megoldások még jó ideig csak a filmekben maradnak.

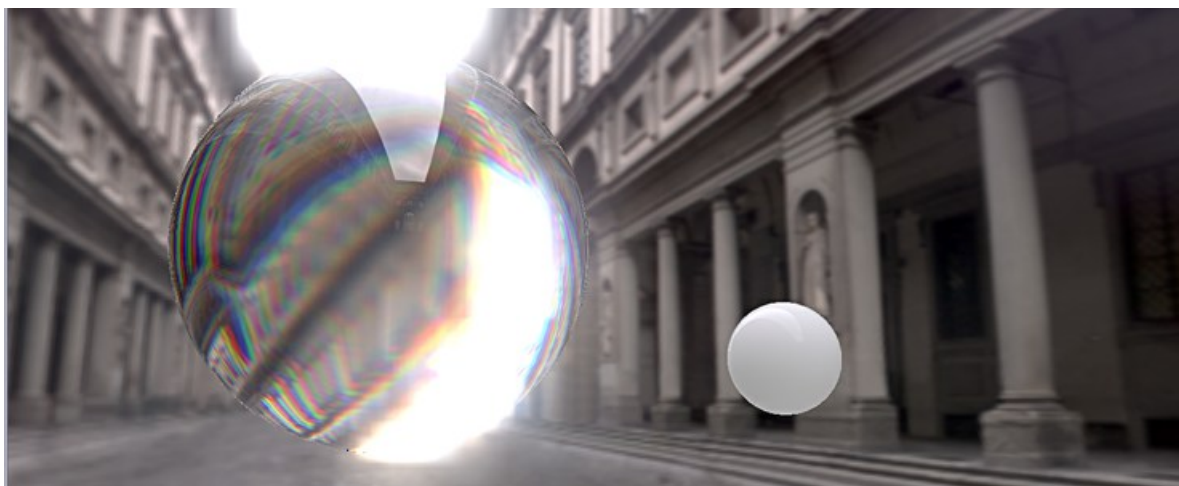
Ahogy haladunk előre a szintér elemeinek megismerésével, egyre finomabbá kezd válni a virtuális térben létrehozott környezetünk, azonban a polygonváz – számítási teljesítmény miatt visszafogott – felbontása még mindig problémát okoz. Erre egy elterjedt megoldást jelent az úgynevezett buckaleképezés (Bump Mapping), amely egy újabb réteg textúrát ad a modellünkhöz, azonban szorosan kapcsolódva az eredeti mintához. Ez a textúra minden pixelében egy normálvektort ír le, melyet szintén pixelenként kiértékelve a fénysugár függvényében egy részletes, térbeli felület hatását kelti. Ha megfelelő szögben nézzük az adott objektumot (se nem túl kis szögben, se nem közvetlen szemből), akkor a felület úgy néz ki, mintha jóval több polygon segítségével modelleztük volna azt. A buckaleképezés elemei megfigyelhetők a 11-6. ábrán, ahol a textúrát egy nagyon alacsony poligontartalmú gömbre feszítettük ki.



**11-6. ábra: Buckaleképezés és a normálvektorokat tartalmazó textúra**

Sajnos a könyv terjedelme (és legfőképpen témakörei) miatt a virtuális objektumok számítógépes grafikai módszereit nincs lehetőségem tovább részletezni, bár minden egyes említett témakör még hosszasan taglalható lenne. Helyette térjünk át a szintér egy másik aspektusára, ami nem más, mint a képernyőtér (a képernyő pixelei, amin „keresztül” a teret látjuk).

Nagyon sok valóságérzetet keltő effektus ezen keresztül valósítható meg, amely megspórolja számunkra azt, hogy a térelem minden egyes pixelét számításba vegyünk, helyette csak a képernyő felbontása szerinti pixelekkel kell számolnunk. Ilyenek például a dinamikus fénykontrasztokat leíró HDR (High Dynamic Range) effektek, melyekkel valósághű vakító és tompított fényviszonyokat tudunk létrehozni. Ide tartoznak a különböző elmosások, amelyekkel a sebesség illetve a térbeli távolság hatását tudjuk hangsúlyozni. De ilyen például a sci-fi filmekre jellemző, drámai hatások elérése is, mint a kásás, zavaros (zajjal torzított) megjelenítés, vagy a filmszakadást imitáló sávosítás is. Ezeket az effektusokat mondatokkal megfogalmazva nehézkes szemléltetni, helyettem inkább beszéljenek a képek (11-7. és 11-8. ábra)!



**11-7. ábra: Fénykontrasztos (HDR) effekt**



**11-8. ábra: Többszörös effektek képernyőterben**

A fenti megoldások segítségével sok minden megvalósítható, azonban hol vannak az olyan természetben előforduló jelenségek, mint az eső, hó, tűz, füst és az ősszel hulló falevelek? Ezeknek a modellezése pixel alapon rendkívül körülményes lenne, ezért makroszkopikusabb elemekkel kell megközelítenünk a problémát. Képzeljünk el egy egyszerű kis textúrát, amely egy esőcsepp mintázatát tartalmazza áttetsző háttérrel! Feszítsük azt ki egy négyzögre, amely mindig a kamera felé néz. Készítsünk belőle sokat, fűzzük fel egy listára, és adjuk át egy olyan metódusnak, amely megmondja, hogy az egyes négyzeteknek milyen irányba, milyen sebességgel és hogyan kell mozogni! A kód gondoskodjon arról is, hogy ha egy elem már nem látható, akkor a helyére pótoljon egy új elemet, amellyel az egész folyamat újra lejátszódik! Ezt a megközelítést részecskerendszernek hívják, és a legtöbb, sok elemből álló és egységes szabály szerint működő jelenség megvalósítható vele. A mai modern játékok a részecskerendszert is a GPU segítségével valósítják meg és fizikai szimulációval egészítik ki, amely még sokkal valóságosabb eredményt ad.

Elég sokat barangoltunk az elméleti síkon, még egy kitérőt teszünk a játékciklus megismerésével, majd utána a gyakorlatban is megnézzük, hogy egy egyszerű kétdimenziós játék megvalósításához milyen elemekre van szükségünk, és hogyan használjuk őket.



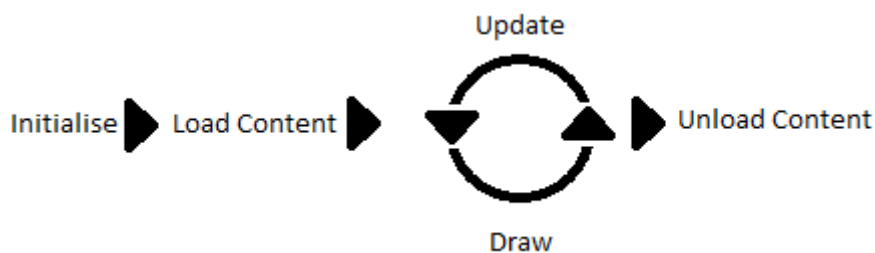
## A játékciklus

Amikor alkalmazásokat fejlesztünk, egy olyan környezethez vagyunk hozzászokva, ahol a központi szerepet az események kapják. Ha valamilyen tennivalója van a programnak, a felhasználó valamilyen interakciót végez a felhasználói felületen vagy a háttérben valamilyen adatszinkronizáció történik, akkor úgy tekintünk a programra, hogy „fut”, majd amikor nincs több feladata, akkor megáll, és egészen addig nem végez újabb feladatot, amíg az előbb felsorolt tevékenységek közül valamelyik be nem következik.

Természetesen ez csak elgondolás szintjén történik így, ez az eseményvezérelt szemlélet sajátossága. Az alkalmazás a háttérben egy pillanatra sem áll meg. Egy játéknak a PC esetében másodpercenként körülbelül 60, mobiltelefon esetében kb. 30-szor kell pillanatfelvételt készítenie, és az adott képkockát megjelenítenie a felhasználó számára, így a folytonosság, a mozgás érzetét keltve. Nos, a felhasználó másodpercenként 30 alkalommal biztosan nem fogja lenyomni az egérgombot, ezzel nem jön létre külső interakció, így magunknak kellene eseményt létrehozni, amely hatására megtörténik a képszintézis (adott képkocka létrehozása – továbbiakban csak renderelésként fogjuk említeni).

Játékok írása során nem ezt a megközelítést szokták használni, hanem az úgynevezett játékciklusos megoldást. Ez azt jelenti, hogy úgy tekintünk a programra, ahogy az valójában működik is, folyamatosan fut, amíg ki nem lépünk. A program indulása során inicializáljuk a szükséges paramétereket, betöltjük a tartalmakat (képek, hangok, modellek, stb.), majd belépünk egy végtelen ciklusba, amelyen belül folyamatosan ismétlődni fog a renderelési lépés, így garantáltan másodpercenként legalább 30-szor létrejön a kívánt kép. Ez eddig tökéletesen működőképes elgondolás, de hogyan kezeljük a bemenetet, a fizikai szimulációt, a mesterséges intelligenciát és még számos más komponensét a játéknak, amelyek nélkül működésképtelen?

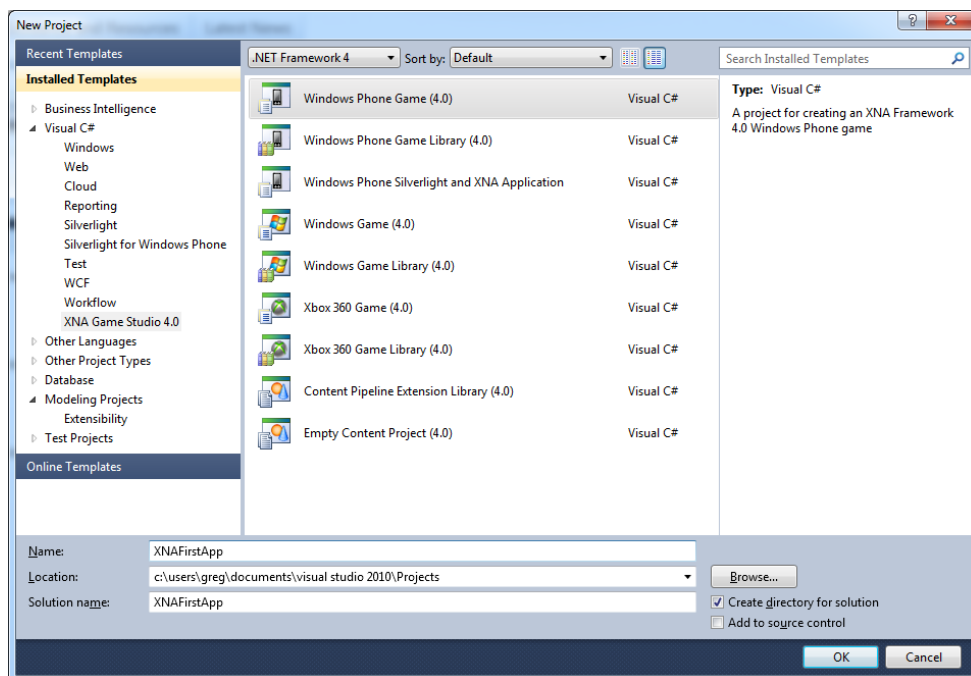
Ezekhez nincs szükségünk eseményekre! Mindössze a ciklus törzsén belüli sorrendre kell odafigyelnünk. Először megnézzük, hogy volt-e valamilyen bemenet a felhasználó oldaláról – Xbox kontrolleren egy gomb lenyomása, Windows Phone telefon képernyőjének megérintése, billentyű illetve egérekattintás. Ha volt, akkor kezeljük azt, majd továbblépünk például a mesterséges intelligencia lépéseinek a számítására. Ha pedig ez is megtörtént, akkor jöhet a képszintézis, azaz a számítások és a bemenetek függvényében a szintér (az a tér, ahol a játék szereplői, dinamikus és statikus tartalmai elhelyezkednek, és amit, mint szemlélő kívülről tekintünk) renderelése. Mindezt másodpercenként 30 alkalommal végezzük el. Ha a felhasználó a menüben a kilépésre kattint (Windows Phone esetében ezt a szerepet a Vissza gomb kapta), akkor a ciklusból kilépünk, elmentjük az állapotot és felszabadítjuk a memóriából a jelentősen erőforrás-igényes tartalmakat. (11-9. ábra)



11-9. ábra: A játékok végrehajtási modellje

## Egy XNA program felépítése

A hosszas elméleti bevezető után nézzük meg, hogyan épül fel egy XNA alkalmazás! Indítsuk el a Visual Studio-t, és hozzunk létre egy új projektet! Itt a C# nyelvű sablonokat tartalmazó szekción belül válasszuk ki az XNA Game Studio 4.0 kategóriát! Ezen belül egy Windows Phone Game (4.0) típusú projektre lesz szükségünk (11-10. ábra).

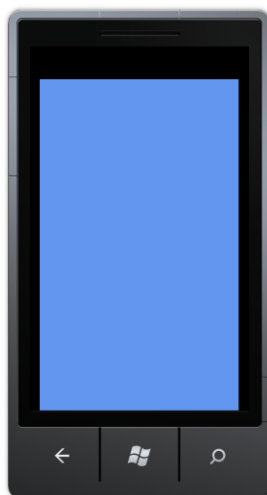


**11-10. ábra: Új XNA projekt létrehozása Visual Studio-ban**

Minden sablont nem tudunk részletesen megvizsgálni ebben a fejezetben, álljon itt egy rövid összefoglalás arról, hogy melyik mire használható:

- **Windows Phone Game:** Ez az alapja a leendő XNA játékunknak. Az inicializáláshoz szükséges részeket, alapbeállításokat, a tartalomkezelőt és a játékciklust tartalmazza.
- **Windows Phone Game Library:** A fordítás során DLL fájl keletkezik belőle, amelyet az alkalmazás elemeinek dekompozíciójához tudunk felhasználni.
- **Windows Phone Silverlight and XNA Application:** A Mangóhoz szorosan kapcsolódó Windows Phone 7.5 újdonság, amely lehetővé teszi, hogy az XNA és Silverlight környezeteket együtt használjuk.
- **Content Pipeline Extension Library:** Ezzel a projektípussal saját fájlformátumunkhoz készíthetünk feldolgozót a tartalomkezelő számára. A tartalomkezelőről még nem beszéltünk, de a fejezet során megnézzük a működését.

Mielőtt részletesen megnéznénk, hogy a projekt milyen részekből épül fel, emulátor segítségével próbáljuk ki az alkalmazást!



**11-11. ábra: Egyszerű XNA sablon emulátoron való futtatása**

A Visual Studio két projektet készített elő számunkra. Az egyik maga a játék, ahol kódunkat fogjuk írni, a másik viszont egy tartalomkezelő (Content Pipeline) projekt, melynek feladata egy helyre összegyűjteni a játékhoz tartozó tartalmakat, mint például a textúrák, a modellek, a hangok, a zenék, és így tovább. Miért van erre szükség? Az XNA alatt a különböző fájlokat nem fájlokként fogjuk használni, hanem a tartalomkezelő rendszer segítségével tudjuk betölteni. Amikor a játék forráskódját Visual Studio-ban lefordítjuk, és abból egy XAP fájl keletkezik (melyet aztán a telefonra tudunk telepíteni), olyankor a tartalomfájlok is átdolgozásra kerülnek, amely azt jelenti, hogy az XNA számára kezelhető formátumba kerülnek átalakításra.

Tekintsünk például egy textúrát, amelynek a neve **xna100.png**! Amikor a tartalomkezelőtől ezt az állományt lekérjük majd, a kiterjesztését az elérési útban már nem fogjuk szerepeltetni, mivel a PNG formátumú képfájlból a Visual Studio egy bináris képfájlt készít, melyet az XNA majd értelmezni tud. És itt felhívom a figyelmet arra, hogy a XAP fájl méretét jelentősen növelni tudják a képek és zenék. Hiába dolgozunk valamilyen tömörítési algoritmust használó médiaformátummal, az átkódolt állomány mérete jelentősen nagyobb lehet, mivel az már egy belső formátumra kerül átalakításra. Ha a tartalomkezelő nem támogat egy fájlformátumot, vagy valamilyen saját, általunk készített formátumhoz szeretnénk értelmezőt készíteni, akkor arra lehetőségünk van a Pipeline Extension projekt típus segítségével. A tartalomkezelő kódból való használatát még a későbbiekben látni fogjuk.

Ha áttérünk a játék kódját tartalmazó projektre, akkor láthatjuk, hogy a program teljes feladatát pillanatnyilag a **Game1** osztály látja el, amely a **Game1.cs** fájlban található. Nézzük meg a tartalmát:

```
public class Game1 : Microsoft.Xna.Framework.Game
{
    GraphicsDeviceManager graphics;
    SpriteBatch spriteBatch;

    public Game1()
    {
        graphics = new GraphicsDeviceManager(this);
        Content.RootDirectory = „Content”;

        TargetElapsedTime = TimeSpan.FromTicks(333333);

        InactiveSleepTime = TimeSpan.FromSeconds(1);
    }

    protected override void Initialize()
    {
        base.Initialize();
    }

    protected override void LoadContent()
    {
        spriteBatch = new SpriteBatch(GraphicsDevice);
    }

    protected override void UnloadContent()
    {
    }

    protected override void Update(GameTime gameTime)
    {
        if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)
            this.Exit();

        base.Update(gameTime);
    }

    protected override void Draw(GameTime gameTime)
    {
        GraphicsDevice.Clear(Color.CornflowerBlue);
    }
}
```

```
        base.Draw(gameTime);  
    }  
}
```

A **GraphicsDeviceManager** objektum segít elérni a program számára használható beállításokat. Itt tudjuk például beállítani a játék felbontását, a státuszsort eltüntetni, és ezzel teljes képernyőssé tenni a programot, és lekérdezni a fejezet korábbi részében tárgyalt profilbeállításokat. Amikor emulátorban lefuttattuk a programot, láthattuk, hogy a felbontása nem megfelelő, és nem is használja a teljes képernyőt. Ahhoz, hogy mi magunk állítsuk be a felbontást a Windows Phone telefonok által pillanatnyilag egységesen támogatott 480×800 pixeles méretűre, helyezzük el a következő kódsorokat az osztály konstruktorán belül:

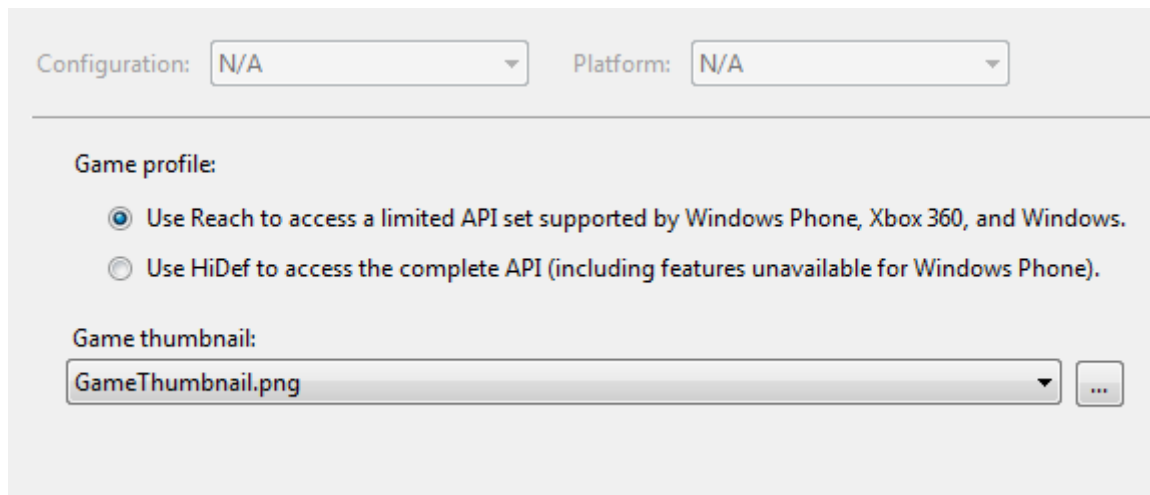
```
graphics = new GraphicsDeviceManager(this);  
graphics.PreferredBackBufferWidth = 480;  
graphics.PreferredBackBufferHeight = 800;  
graphics.IsFullScreen = true;
```

A konstruktor tartalmaz még két fontos beállítást. A másodpercenkénti képkockák számát (*frames per second*) harmincra korlátozza, amely a telefonokon futó játékprogramok számára irányelv, és érdemes is minden esetben betartani. Továbbá definiálja, hol fogjuk a tartalomkezelő segítségével elérni a játékhoz tartozó fájlokat.

Két külön metódusunk van, amely a tartalmak kezelésével foglalkozik. Az első a betöltésért, míg a második a memóriából való felszabadításért felel. A **LoadContent()** metódus elején található egy nagyon fontos sor, amely az osztályhoz tartozó **SpriteBatch** példányt hozza létre. Ennek az objektumnak a feladata közvetlen kapcsolatba lépve a grafikus hardverrel, hogy az a későbbiekben a 2D-s képeinket (ezek az ún. *sprite*-ok) kirajzolja.

A **Draw()** és az **Update()** metódusok a játékciklus részei. Ahogy a nevük is mutatja, az **Update** felel a mögöttes számításokért. Itt érdemes kezelnünk a bemeneteket, a játék logikáját és minden egyéb számítást, amelyre a játéknak szüksége van. Ha az **Update** lefutott, akkor kerül sorra a **Draw** meghívása, amely először törli a rasztértár előző tartalmát, és felrajzolja a színterünket pillanatnyi állapotnak megfelelően.

OpenGL és Direct3D használata esetében a játékmotor feladata az, hogy kiderítse az adott hardver képességeit – sajnos nem rendelkezik minden egyes grafikuskártya ugyanazokkal a képességekkel. Ha valamit nem támogat a hardver, akkor a CPU-nak is átadható a feladat, azonban ennek a sebessége katasztrofális, nem érdemes vele próbálkozni. Így csak az marad, hogy az adott funkciót nem használhatjuk ki, és a játék grafikája már nem lesz olyan szép, mint amilyennek eredetileg terveztük. Az XNA keretrendszer egyszerűsíti ennek a folyamatnak a lépéseit. Kétféle profilt használhatunk. A Reach profil mind a három platformon azonosan működik, azonban nem használja ki a grafikuskártyák adta legkülönbélebb speciális funkciókat. Az előnye, hogy biztosak lehetünk, hogy az alkalmazás futni fog az adott célhardveren. A másik profil a HiDef. Itt nincsenek limitációink, minden, ami az XNA és a hardver által támogatott, azt szabadon használhatjuk. A profilbeállítások az XNA projektre kattintva a Properties szekcióban találhatók (11-12. ábra).



11-12. ábra: XNA alkalmazásprofilok

## Sprite-ok

Ahogy azt már a fejezet korábbi részében leszögeztük, nem foglalkozunk 3D-s tartalmakkal, csak a 2D-s játékok fejlesztéséhez szükséges elméleti és gyakorlati anyagokat vesszük sorra. Egy térbeli játék esetében szükséges lenne, hogy alakzatokkal foglalkozzunk, és azoknak a csúcspontjai alapján hajtsuk végre a renderelést, azonban most sokkal könnyebb dolgunk lesz. Gondoljunk csak bele egy egyszerű 2D-s játék esetében képeket rajzolunk fel, ezek a képek egy négyzög területét töltik ki. Ha valamilyen interakció történik, akkor ezeket a négyzeteket kell mozgatnunk a képernyő dimenziói között. Hogyan tudunk egy képet megjeleníteni XNA-ban? Textúraként. Ebben a feladatban a `Texture2D` objektum fog segíteni, amely a tartalomvezérlőtől le tudja kérni a szükséges képinformációkat. Mivel nincs még képünk, így keressünk egy tetszőleges PNG vagy JPG fájlt, és helyezzük el a Content projekten belül!

Tipp: a tartalomkezelő (Content) projekten belül mindig érdemes rendet tartani. Itt is tudunk létrehozni mappákat és almappákat, amelybe típusuktól függően rendezhetjük a fájlokat, így a betöltés kódból sokkal strukturáltabb lesz, esetleg automatizálható.

A tartalom (jelen esetben ez a textúra) lekérést a `Content.Load<T>()` generikus metódus segítségével tudjuk elvégezni. Itt mindig a tartalom **Content** projekten belüli elérési útjára hivatkozunk és az adott fájl kiterjesztését hagyjuk le, mivel a tartalomkezelő számára ez átalakításra kerül a projekt fordítása során.

Ezek után megkezdhetjük a sprite kirajzolását! Ez mindig képernyő (pontosabban a rasztértár) törlésével kezdődik, amelyet a `GraphicsDevice.Clear()` metódusa végez el, és ennek egyetlen paramétere a törlési szín megadása, a képernyő összes pixele ezt a színt veszi fel.

Ha a képernyőt töröltük, akkor jöhet a tényleges rajzolás. Ez mindig a `SpriteBatch.Begin()` és `SpriteBatch.End()` metódushívások között történik a `Draw` metódus hívásával. A `Draw` több paraméterezéssel is elérhető, ezek közül kettő fontos számunkra:

Az első paraméterezésben szereplő paraméterek sorrendben:

1. A kirajzolandó textúra objektum.
2. Egy négyzög, amely első két paramétere a sprite pozíciója, a második kettő pedig a sprite mérete pixelekből kifejezve.
3. A textúrát lehetőségünk van összemosni egy másik színnel. Ha fehértől eltérőt adunk meg, akkor a textúra pixelei összeszorzásra kerülnek az adott színnel.

A második paraméterezés:

1. A kirajzolandó textúra objektum.
2. Ugyanaz a négyzög, amely az előző paraméterezés esetében is.

3. Egy újabb négyszög, amely segítségével a textúrából egy darabot ki tudunk vágni. Jelentőségét az ún. *sprite sheet*ek létrehozásakor fogjuk látni a fejezet egy későbbi részében.
4. Az összemosási szín.
5. Z-tengely körüli (a képernyő síkjában) való forgatás valós szöggel kifejezve.
6. A sprite origójának megadása. Alapesetben ez a képernyő bal felső sarka (0, 0).
7. A sprite horizontális illetve vertikális tükrözésére van lehetőségünk a **SpriteEffects** osztály használatával.
8. Z-Index: ez egy nagyon fontos paraméter! Ezzel adhatjuk meg [0; 1] közötti valós számok segítségével, hogy az adott sprite a többi előtt vagy mögött helyezkedik-e el, ezzel mélységi sorrendet tudunk kialakítani. Pl. egy egyszerű kétdimenziós játékban nem szeretnénk, ha egy főcsomó vagy bokor lenne a játékosunk előtt, így a mélységi sorrendjüket megváltoztatni szükséges.

A következő kód egy textúrát rajzol ki két különböző módon:

```
private Texture2D xnaLogo;

protected override void LoadContent()
{
    spriteBatch = new SpriteBatch(GraphicsDevice);

    xnaLogo = Content.Load<Texture2D>( @"xna-logo" );
}

protected override void Draw(GameTime gameTime)
{
    GraphicsDevice.Clear(Color.Black);

    spriteBatch.Begin();
    spriteBatch.Draw( xnaLogo, new Rectangle( 200, 400, 230, 114 ), Color.White );
    spriteBatch.Draw( xnaLogo, new Rectangle( 200, 400, 230, 114 ), null,
        Color.CornflowerBlue, 45.0f, Vector2.Zero,
        SpriteEffects.FlipVertically, 0.0f );
    spriteBatch.End();

    base.Draw(gameTime);
}
```

Ha a kódot implementáltuk és a megfelelő képfájl a Content projekt része, akkor nyugodtan próbáljuk ki emulátorban az alkalmazást (ha van rá lehetőségünk, telefonon is kipróbálhatjuk, ugyanazt az eredményt fogjuk kapni). A program futását a 11-13. ábra szemlélteti.

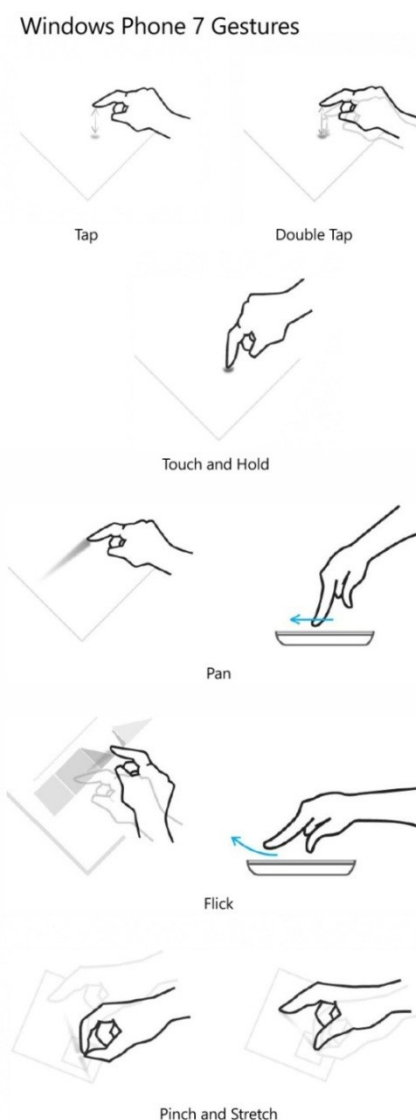


11-13. ábra: Két sprite kirajzolása két különböző módon

## Bemenetek kezelése

Minden egyes XNA által támogatott eszközre külön meg lehetne nézni a bemenetek kezelését, mivel az teljesen más egy Xbox és egy PC esetében is. Ebben a fejezetben mi a Windows Phone Mangóra szeretnénk játékokat írni. A telefonok azonos időben egyszerre több érintést képesek lekezelni a képernyőjükön keresztül, így az alkalmazásainkat egy Multi Touch (több érintést kezelő) környezetre kell kialakítanunk. Az érintések kezelésére többféle lehetőségünk is van. Ha nem kérjük az operációs rendszer segítségét, hanem mi magunk szeretnénk feldolgozni minden bemeneti információt, akkor lehetőségünk van a „nyers” érintésadatok lekérésére, ami azt jelenti, hogy egy tömbben visszkapjuk azokat az (X, Y) koordinátákat, ahol a képernyőhöz ért a felhasználó. Ebből saját algoritmussal kell megállapítanunk, hogy a program milyen választ szolgáltasson a bemenet hatására.

Azonban a Windows Phone magas szintű támogatást nyújt az érintéspontok kezelésére. Az operációs rendszer beépítve tartalmaz mintákat (*gestures*), amelyek bizonyos érintésmódok hatására aktiválódnak, és a programozó számára már események képében kerülnek megvalósításra. Ilyen lehet például egy egyszerű, pár pillanatig tartó érintés (*tap*), vagy egymás után gyorsan végrehajtva két rövid érintés (*double tap*), de ez lehet akár egy folyamatos mozgás is a képernyő felületén (*touch and hold*). Az 11-14. ábra a különböző érintésmintákat mutatja be.



11-14. ábra: Érintésminták Windows Phone Mangón

Az érintésminták feldolgozása az XNA alkalmazásokban nagyon könnyen elvégezhető! Első dolgunk engedélyezni a program számára azokat a gesztusokat, amelyekkel dolgozni szeretnénk. Ezt a **TouchPanel** objektumon keresztül tudjuk megtenni. Az **EnableGestures** tulajdonságát a **GestureType** elemeivel kell feltölteni, ahol felsoroljuk a használni kívánt érintésmintákat. Az egyes elemeket logikai "vagy" művelet segítségével tudjuk felsorolni.

Ha a gesztusok engedélyezésre kerültek, akkor a játékciklusunk **Update** szekciójában folyamatosan figyelniünk kell, hogy volt-e valamilyen bemenet. Ha voltak interakciók –ennek ellenőrzéséhez nem kell más, mint a **TouchPanel.IsGestureAvailable** tulajdonságának lekérdezése –, akkor elkezdjük azokat feldolgozni. Megkérjük az operációs rendszert, hogy szolgáltatssa vissza számunkra azt a érintésmintát, amelyiket legutoljára észlelt. Ezt a **TouchPanel.ReadGesture()** metódusa teszi. Ha több minta is engedélyezve van, akkor meg kell vizsgálnunk a minta típusát.

A következő kis mintaalkalmazás két mintát figyel. Az egyik az egyszerű érintés, a másik pedig a szabad, folyamatos mozgatás a képernyő felületén. Ha egyszerű érintés történt, akkor megváltoztatjuk a program háttérszínét, ha pedig a szabadon való mozgatás, akkor a Windows Phone logót ábrázoló textúrát elmozdítjuk az érintés irányába.

```
private Texture2D xnaLogo;
private Texture2D wp7Logo;
private Point position;
private Color clearColor = Color.White;

protected override void Initialize()
{
    TouchPanel.EnabledGestures = GestureType.FreeDrag | GestureType.Tap;

    base.Initialize();
}

protected override void LoadContent()
{
    spriteBatch = new SpriteBatch(GraphicsDevice);

    position = Point.Zero;
    xnaLogo = Content.Load<Texture2D>( @"xna-logo" );
    wp7Logo = Content.Load<Texture2D>( @"wp7-logo" );
}

protected override void Update(GameTime gameTime)
{
    while ( TouchPanel.IsGestureAvailable )
    {
        GestureSample gesture = TouchPanel.ReadGesture();

        switch ( gesture.GestureType )
        {
            case GestureType.FreeDrag:
                position = new Point( (int)gesture.Position.X - 149 / 2,
                                     (int)gesture.Position.Y - 177 / 2 );
                break;

            case GestureType.Tap:
                clearColor = clearColor == Color.White ? Color.Black : Color.White;
                break;
        }
    }

    base.Update(gameTime);
}
```

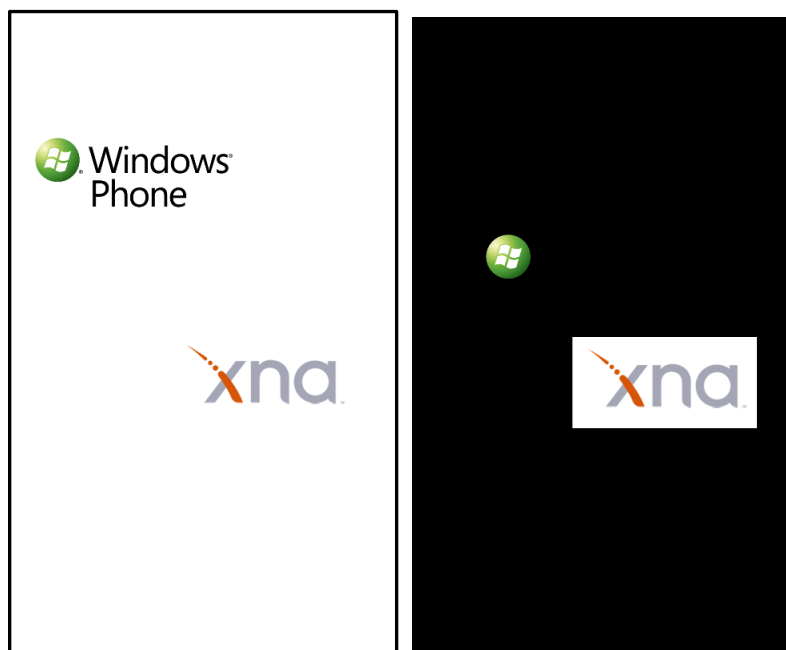


```
protected override void Draw(GameTime gameTime)
{
    GraphicsDevice.Clear( clearColor );

    spriteBatch.Begin();
    spriteBatch.Draw( xnaLogo, new Rectangle( 200, 400, 230, 114 ), Color.White );
    spriteBatch.Draw( wp7Logo, new Rectangle( position.X, position.Y, 256, 102 ),
        Color.White );
    spriteBatch.End();

    base.Draw(gameTime);
}
```

Az érintések hatásait a 11-15. ábrán láthatjuk.



**11-15. ábra: Az érintések hatása a mintaalkalmazásban**

## Ütközések detektálása

Az ütközések észlelése, azaz a játék objektumainak egymással való ütköztetése minden játékban nagyon fontos szerepet tölt be. Gondoljunk csak bele, hogy ha egy lövöldözős játékban nem kezeljük le a lövedék becsapódását, a golyó szabadon elrepülhet a végtelenbe. Ez a játékelményt nemcsak, hogy teljesen megöli, de a felhasználó szempontjából hibásnak tünteti fel a programunkat – amely teljesen jogos következtetés! Hogyan történik az ütközések kezelése? Először nézzük meg ezt általánosítva!

A fejezet korábbi részében alaposan kiveséztük a poligonvázak felépítését, és levontuk a következtetést: egy grafikus alkalmazásban minden matematikai alapra vezethető vissza. Miután összetett alakzatokról beszélünk, így sokszögek sokszöggel való ütközését (metszését) kell kezelnünk. Ha csak két szereplőt nézünk, akkor is az A szereplő összes poligonjára meg kell vizsgálnunk, hogy B szereplő valamelyik poligonjával ütközött-e. Ez még kevés objektum esetén is jelentősen vissza tudja fogni a teljesítményt, ezért más módszerre lesz szükségünk! Sokkal jobb megoldás az, ha az objektum köré valami sokkal egyszerűbb térgometriai alakzatot vonunk, pl. téglatest, gömb, stb. Ha ezeknek az ütközését figyeljük, akkor sokkal kevesebb számítással jutunk eredményre. Viszont sok objektum esetében még ez is lassuláshoz vezethet, ezért különböző struktúrákat és eljárásokat szoktak még bevezetni.

Például a BSP vagy a Quad fa a teret cellákra osztja fel, és ha két cella olyan helyzetben van egymáshoz képest, hogy tudjuk, a belül található objektumok között lehetetlen az ütközés, akkor velük már nem is

számolunk. A mai játékok jelentős hányada is ezeket a megközelítéseket használja, különböző módokon kombinálva és továbbfejlesztve azokat.

Szerencsére mi kétdimenziós környezetben dolgozunk, így jóval egyszerűbb esetekre kell csak felkészülnünk. Négyszögekkel fogjuk körülvenni az alakzatokat, és annak is a könnyebben kezelhető formájával, ahol eltekintünk a különböző forgatásoktól. Ha két négyszög akármilyen formában is metszi egymást, akkor ütközés történt, és ennek függvényében haladunk tovább a játékban.

Azokat a befogó alakzatokat, amik a sík/térbeli forgatásokra is fel vannak készítve, *Axis Aligned Bounding Volume*-nak hívják, részletes leírásokat lehet találni Wikipedián.

Az **IsCollision()** metódus a két befogó négyszöget megnézi, hogy metszik-e egymást, és az ennek megfelelő logikai értéket dobja vissza. Az **Update()** metóduson belül folyamatosan figyeljük, hogy volt-e ütközés, és ha igen, akkor az egyik objektumot átszínezzük.

```
private Color logoColor = Color.White;

protected bool IsCollision()
{
    Rectangle xna = new Rectangle( 200, 400, 230, 114 );
    Rectangle wp7 = new Rectangle( position.X, position.Y, 256, 102 );

    return xna.Intersects( wp7 );
}

protected override void Update(GameTime gameTime)
{
    ...

    if ( IsCollision() )
        logoColor = Color.Yellow;
    else
        logoColor = Color.White;

    base.Update(gameTime);
}

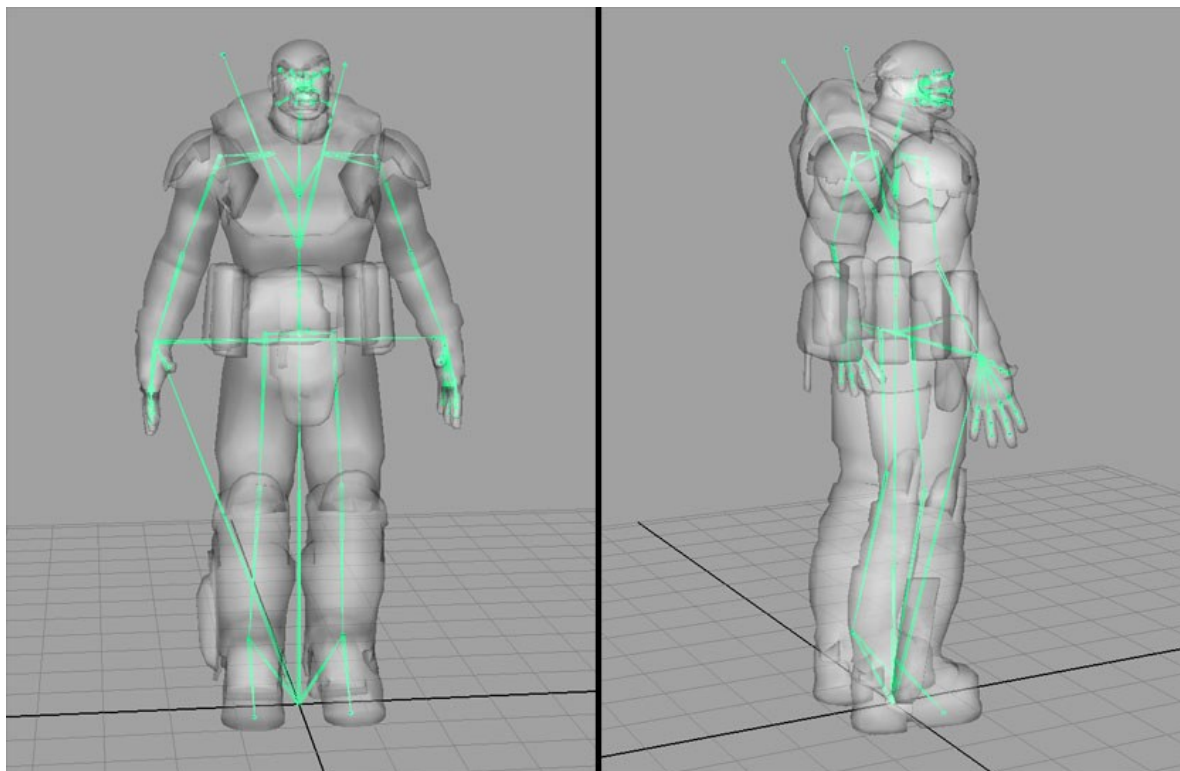
protected override void Draw(GameTime gameTime)
{
    ...
    spriteBatch.Draw( wp7Logo, new Rectangle( position.X, position.Y, 256, 102 ),
                      logoColor );
    ...
}
```

### Animációk

Amikor számítógépes játékokban animációról beszélünk, fontos megkülönböztetnünk, hogy pontosan melyik esetről is van szó. Amikor a telefon mozdításának segítségével egy autót kormányozunk, és a képernyő megérintésével például lehetőségünk van fékezni, akkor a meglévő objektumot mozgatjuk a térben különböző szabályok alapján. Egy másik eset az, amikor magát a virtuális objektumot animáljuk, az autós példánál maradva, az autónak több tengely körül forognak a kerekei vagy akár kinyílhatnak az ajtói. Ha az előbbiről van szó, akkor programozottan, kódból kell az animációt létrehozunk, és a kódban kell meghatározunk, hogy az alakzatot merre toljuk el térben, mennyire forgatjuk el, vagy mennyire kicsinyítjük le, nagyítjuk fel.

Meghatározhatunk görbéket (utakat), amin az objektum végighaladhat, akár egyes pontjaiban más animációkkal kombinálva folytatódhat tovább a cselekmény. Viszont a második esetben nem tudjuk kódból (legalábbis csak nagyon nehézkesen) létrehozni az előbb említett lépéseket, az adott geometriának a pontjait körülményes lenne külön-külön transzformálni, és ebből valamilyen valóság

mozgást előállítani, ezért ezt valamilyen modellező alkalmazás segítségével (3D Studio Max, Maya, Rhino 3D, stb) szokták végrehajtani. A térbeli animálásnak rengeteg aspektusa létezik, amelyek a program komplexitását jelentősen növelni tudják. A modern játékok jelentős része – megint csak a valós életre visszavezetve – ún. „csontozáson” alapuló karakter animációkat alkalmaz, ezt kiegészítve ízületekkel és a csontok bőrhöz (*skin*) való igazításával. A grafikus szakember a csontokat animálja, ízületi részekkel összeköti őket, amely megszabja például, hogy egy térd mekkora szögben és merrefelé tud hajlani. Ha ez megvan, akkor a poligonmodellel összeköti és definiálja az anyagtulajdonságokat. Ezek a tulajdonságok azt írják le, hogy ha a csontozat mozog, akkor a poligonváznak hogyan kell ahhoz alkalmazkodnia és deformálódnia, transzformálódnia a térben. Továbbá a csontozásos rendszer segít a játéktérben való adaptációban is, mivel a fizikai motor szintén felhasználhatja a csontokat rugalmas testek fizikai hatásainak modellezéséhez. A csontozásos animációt az 11-16. ábra mutatja be, amelyen a híres Unreal Engine játékmotor karakteranimálása látható.



**11-16. ábra: Csontozásos animáció – Unreal Engine**

Két dimenzióban azonban nincsenek olyan poligonok, amelyeket animálni lehetne akármelyik módszert is használva. Adott egy ötlet: ha a karaktereket sprite-ként jelenítjük meg, akkor az animáció lehetne a képek egymás után váltakozása adott sebességgel. Egy másodperc animációhoz tegyük fel, hogy szükség lenne 30 képkockára. Ehhez 30 különböző képet kell egymás után, gyorsan megjeleníteni. 30 külön képfájlban tárolva a képeket, azok kezelését jelentősen túlbonyolítanánk, ráadásul feleslegesen. Helyette a *sprite sheet* nevezetű megoldást fogjuk használni. Az animáció összes lépése egy adott képfájlban kerül eltárolásra, amelyet úgy dolgozunk fel, hogy sorra vesszük a kép  $X \times Y$  méretű részeit. Ha egy soron végigértünk, akkor a következő sorban folytatjuk a „kis képek” kivágdosását, amíg a végére nem érünk az adott animációnak. A 11-17. ábrán egy futónak az animációját bemutató sprite sheet látható.



**11-17. ábra: Egy futó animációjának leírása sprite sheettel**

Látható, hogy a fenti kép egy  $6 \times 5 = 30$  képkockás animációt tartalmaz. A telefonon egy játék 30 FPS-szel fut, de nem lehet minden animációnak azonos a sebessége, az teljesen furcsa hatást keltene, ha például egy autó ugyanolyan sebességgel haladna, mint a felhők. A következőkben egy olyan sprite sheet osztályt fogunk készíteni, amelyen szabályozható, hogy az adott animációt milyen sebességgel játssza le. Nézzük, milyen tulajdonságokra lesz szükségünk az osztályban:

- **FrameSize** – Egyetlen képkocka mérete (szélesség, magasság).
- **SheetSize** – Hány képkockát tartalmaz a sprite sheet (sorban, oszlopban)?
- **CurrentFrame** – Éppen melyik képet jelenítjük meg (sor, oszlop)?
- **Texture** – Az a **Texture2D** objektumpéldány, amely a sprite sheetet tartalmazza.
- **FrameRate** – Az animáció sebessége (képkocka per másodperc).
- **Position** – A kirajzolandó sprite (X, Y) koordinátái.
- **Rotation** – Z-tengely körüli elforgatás mértéke szögben mérve.

Az osztály tetszés és igény szerint tovább bővíthető, de a fenti tulajdonságok megvalósításával a program legtöbb esetben már így is jól használhatóan fog bizonyulni.

Két metódust kell létrehoznunk. Az egyik a frissítésért felel (**Update**), a másik pedig a kirajzolásért (**Draw**). Ha az osztályt a **GameComponent** őssztályból származtatjuk, akkor a későbbiekben nem kell a metódusokról saját magunknak gondoskodnunk, a játékciklus megfelelő helyein meghívásra fognak kerülni.

A frissítést megvalósító metódus paraméterül kap egy **GameTime** objektumpéldányt. Ennek feladata a játékban eltelt idő mérése. Miért van erre szükség? Gondoljunk csak bele, ha a játék nem futtatható valamilyen eszközön megfelelő sebességgel, akkor a valóságban eltelt idő mérése nem lesz elegendő,

szükségünk lesz egy olyan időmérési módszerre is, amely a játékunk „lassult” végrehajtási sebességéhez igazodik.

A metóduson belül megnézzük, eltelt-e már annyi idő, ami az FPS számunknak megfelelő, ha igen, akkor a **CurrentFrame** tulajdonságot megfelelően növeljük, hogy a következő képkockára mutasson.

A rajzolásért felelős kódunk a már ismert **SpriteBatch.Draw()** metódust hívja egy más paraméterezési móddal, mint amit a fejezet korábbi részében használtunk. Számunkra itt a második és a harmadik paraméterek a fontosak. A második paraméter feladata megmondani az XNA-nak, hogy melyik (X, Y) pozícióba rajzolja fel a textúrát és mekkora méreteekkel. A harmadik paraméter már egy kicsit izgalmasabb. Itt kell megadnunk, hogy mekkora darabját rajzolja ki az eredeti sheet-nek. Itt sorszám × képkocka\_szélesség és oszlopszám × képkocka\_magasság értékekkel kell számolnunk. A **Draw()** metódus többi paraméterét a fejezet korábbi részében már átbeszéltük.

```
public class SpriteSheet : GameComponent
{
    public Point FrameSize { get; set; }
    public Point CurrentFrame { get; set; }
    public Point SheetSize { get; set; }
    public Texture2D Texture { get; set; }
    public Point Position { get; set; }
    public float Rotation { get; set; }

    private int timeSinceLastRender = 0;
    public int FrameRate { get; set; }

    public SpriteSheet( Game game, Texture2D tex, Point frameSize,
                       Point sheetSize, int speed ) : base( game )
    {
        CurrentFrame = Point.Zero;
        Texture = tex;
        FrameSize = frameSize;
        SheetSize = sheetSize;
        FrameRate = speed;
    }

    public void Draw( SpriteBatch spriteBatch )
    {
        spriteBatch.Draw( Texture, new Rectangle( Position.X, Position.Y,
                                                  FrameSize.X, FrameSize.Y ),
                          new Rectangle( FrameSize.X * CurrentFrame.X,
                                          FrameSize.Y * CurrentFrame.Y,
                                          FrameSize.X, FrameSize.Y ),
                          Color.White, Rotation, Vector2.Zero,
                          SpriteEffects.None, 0.0f );
    }

    public override void Update(GameTime gameTime)
    {
        timeSinceLastRender += gameTime.ElapsedGameTime.Milliseconds;

        if ( timeSinceLastRender > ( 1000 / FrameRate ) )
        {
            timeSinceLastRender -= 1000 / FrameRate;

            int x = CurrentFrame.X;
            int y = CurrentFrame.Y;

            x++;
            if ( x >= SheetSize.X )
            {
                x = 0;
                y++;
                if ( y >= SheetSize.Y )

```

```
        y = 0;
    }

    CurrentFrame = new Point( x, y );
}

base.Update(gameTime);
}
}
```

Ha az osztály készen van, akkor a **Game1.cs** fájlban használjuk is fel! Hozzunk létre egy új adattagot, amely az újonnan létrehozott osztály egy példánya! A **LoadContent()** metóduson belül szükséges inicializálnunk. A pozícióját a képernyő közepére kiszámítva adjuk meg, majd a játék komponensei közé felvesszük az objektumpéldányt, ennek következtében az **Update()** metóduson belül az XNA automatikusan hívni fogja az objektumhoz tartozó **Update()** metódust. Az emulátoron való futtatás eredményét a 11-18. ábra illusztrálja.

```
private SpriteSheet walk;

protected override void LoadContent()
{
    spriteBatch = new SpriteBatch(GraphicsDevice);

    walk = new SpriteSheet( this, Content.Load<Texture2D>( "walk" ),
        new Point( 240, 296 ), new Point( 6, 5 ), 20 );
    walk.Position = new Point( ( 480 - 240 ) / 2, ( 800 - 296 ) / 2 );
    Components.Add( walk );
}

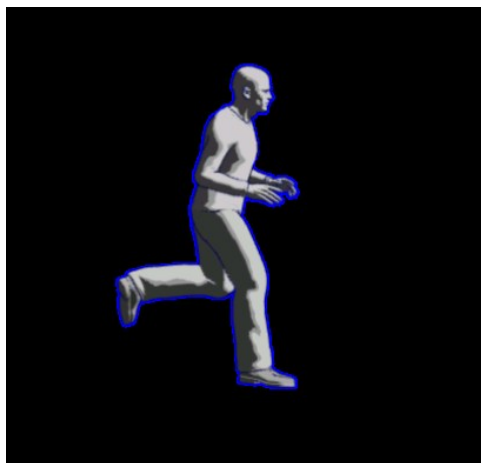
protected override void Draw(GameTime gameTime)
{
    GraphicsDevice.Clear( clearColor );

    spriteBatch.Begin();

    walk.Draw( spriteBatch );

    spriteBatch.End();

    base.Draw(gameTime);
}
```



**11-18. ábra: A sprite animáció egy pillanatfelvétele**

## XNA és Silverlight integráció

A Windows Phone 7 első verziója még élesen szétválasztotta az XNA és a Silverlight által képviselt két külön világot, így például egy Silverlightot használó programban nem jeleníthettünk meg semmilyen XNA tartalmat. A Mango egyik nagy újdonsága, hogy képes hibrid renderelésre, azaz Silverlight alkalmazásokban akár háromdimenziós, textúrázott poligonmodelleket is meg tudunk jeleníteni.

Azonban a két technológia fúziója kérdéseket vet fel! Az egyik játékciklust használ, míg a másik egy eseményvezérelt környezetet valósít meg. Hogyan tudjuk a kettőt közös nevezőre hozni? Az XNA pixelalapú tartalmakat rajzol ki, amíg a Silverlight vektoralapon számított vezérlőkkel látja el ezt a feladatot. Hogyan tudjuk a különböző tartalmakat megjeleníteni? Ezekre a kérdésekre kapunk válaszokat ebben a részben.

### *Silverlight és XNA együtt a gyakorlatban*

Ahhoz, hogy az XNA és Silverlight együttműködését gyakorlatban is megismerjük, egy Snake játékot fogunk írni. Valószínűleg mindenki jól ismeri a klasszikus kígyós játékot, ahol a szereplőnk apró blokkokból épül fel, és a játékos feladata úgy irányítani a kígyót, hogy a pályán véletlenszerűen megjelenő eleségeket összegyűjtse, miközben nem ütközik se saját testébe, se a pályát határoló falakba. A játékot bonyolítja, hogy minden egyes eleségdarabka felszedése után a kígyó hosszabb lesz, így nehezebb lesz elkerülni, hogy valahol ne ütközzön.

Ez a leírás csak a játék alapjait fogja bemutatni, a pontszámítás, a különböző játékelmény-elemek kialakítása már a kedves Olvasó feladata, teljes mértékben a kreativitására bízom.

A pálya hátterét a 11-19 ábrán látható 480×800 pixel felbontású textúra fogja adni, illetve egy apró pontot ábrázoló textúrát fogunk a kígyó szegmensekhez felhasználni. A pályán másra nem is nagyon lesz szükségünk, még egy elemet leszámítva: a szüneteltetés gomb, melyet Silverlight segítségével valósítunk meg.

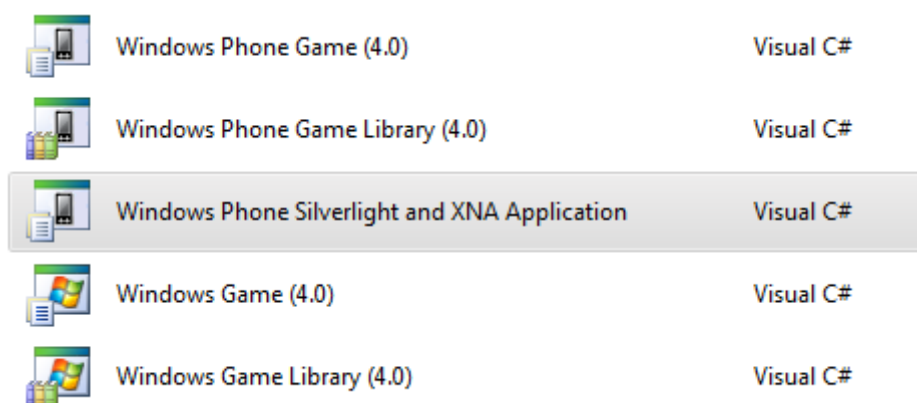
Érdekesség: természetesen XNA-ban is készülnek különböző menürendszerek és „vezérlők”, amelyeket nem játékalizációkban már megszoktunk, azonban ezeknek az elkészítése körülményesebb sokkal, ha nincs a segítségünkre a Silverlight. A játékot mint egy nagy állapotgépet kell elképzelnünk, ahol maga a játék irányítása csak egyetlen állapotot jelent, és ezenkívül számos egyéb is megtalálható az állapotgráfban, pl: menü megjelenítése, mentés megjelenítése, játék vége állapot, stb. A felrajzolás ugyanúgy a játékciklusban történik, és nincsen navigációs rendszerünk, amely az egyes alkalmazásoldalak között tud minket átvinni. Így ha sikeresen ki is alakítunk egy menürendszert, akkor a gombokat, listadobozokat, szövegmezőket még magunknak kell implementálnunk. Ezt úgy képzeljük el, hogy lekezeljük a bemenetet, amely jelen esetben csak egy érintés a telefon képernyőjén! Ha az érintés koordinátái az adott gomb által takart síkrészbe tartoznak, akkor végrehajtunk valamilyen eseményt. Azonban figyelniünk kell arra, hogy a gombot csak egyszer tekintsük lenyomottnak. Mivel a játékciklus másodpercenként harmincszor lejátsszódik, így a gombnyomás is harmincszor hajtódna végre.





**11-19 ábra: a Snake játék háttértextúrája**

Hozzunk létre egy új projektet, ennek a típusa most Windows Phone Silverlight and XNA Application legyen, a neve pedig **Snake**, ahogy a 11-20 ábrán is látható!



**11-20 ábra: új projekt létrehozása a játékhoz**

Látható, hogy a fő projekt tartalmazza a Silverlight alkalmazásokban jól megszokott MainPage.xaml fájlt, amely a felhasználói felületünknek az alapját jelenti. Továbbá bővült a projektünk az eddigiekhez képest egy GamePage.xaml fájljal is, melynek deklaratív része teljes mértékben üres, a mögöttes kód viszont az XNA-val való együttműködésre fel lett készítve.

Készült még két projekt. Az egyik az XNA tartalomkezelője, amellyel már találkoztunk a fejezet során, a másik viszont újdonság. Egy XNA könyvtár, amely hivatkozik a **Content** projektre, továbbá alkalmas arra, hogy elkészítsük benne az XNA komponenseinket. A játék jelentős részét itt fogjuk megírni.

Miután a projektet létrehoztuk, szükségünk lesz egy osztályra, amely a kígyónak a szegmenseit tárolja. Minden egyes szegmensről tudnunk kell a következő információkat:

- **Position:** a szegmens pozíciója a játék felületén (X, Y) koordinátákkal
- **Direction:** az aktuális irány, amerre haladnia kell (irányvektorként tekintünk rá)
- **PrevDirection:** akkor fogjuk felhasználni, amikor a kígyó kanyarodik valamerre
- **Texture:** a kígyószegmens textúrája

Hozzunk is létre egy új osztályt a **SnakeLib** projekten belül, melynek neve **SnakeBlock**, és a tartalma a következő osztálydefiníció a megfelelő tulajdonságokkal:



```
public class SnakeBlock
{
    public Vector2 Position { get; set; }

    public Vector2 PrevDirection { get; set; }

    public Vector2 Direction { get; set; }

    public Texture2D Texture { get; set; }
}
```

Ha megvagyunk a szegmens osztállyal, akkor készítenünk kell még egy másik osztályt is (ígérem, nem visszük túlzásba az objektum-dekompozíciót), amely magát a kígyót és a viselkedéséhez szükséges elemeknek egy nagy részét tartalmazza. Ezt is a **SnakeLib** nevezetű XNA projekten belül. Először is szükséges egy lista, amely a szegmenseket sorolja fel (kezdetben legyen ötelemű), egy kezdeti irány- és pozícióvektor. Miután a textúrákat az osztály magának fogja lekérni, a **ContentManager** létező példányát is át kell majd adni konstruktoron keresztül és letárolni azt.

A sebesség sem közömbös tényező. Ha nem korlátozzuk valamilyen módon, akkor másodpercenként 30-szor lesz léptetve a játékos, amitől teljesen irányíthatatlanná válik. Állítsuk majd be mondjuk fél másodpercenkénti frissítésre, és ehhez tároljuk le az utolsó frissítés időpontját!

A konstruktor feltölti kezdeti értékkel az adattagokat, és létrehozza az első öt szegmenst a kezdeti irány és a pozíció függvényében:

```
public class SnakeObject
{
    public List<SnakeBlock> snake { get; private set; }
    private ContentManager contentMgr;
    private readonly Vector2 baseDir = new Vector2( 0, 1 );
    private readonly Vector2 basePos = new Vector2( ( 480 - 16 ) / 2,
                                                    ( 800 - 16 ) / 2 );
    private DateTime lastTime = DateTime.Now;

    public bool IsPaused { get; set; } // megállítható lesz a játék Silverlight UI-ról

    public SnakeObject( ContentManager cm )
    {
        this.contentMgr = cm;

        snake = new List<SnakeBlock>();
        snake.Add( new SnakeBlock { Direction = baseDir,
                                     Position = basePos } );
        snake.Add( new SnakeBlock { Direction = baseDir,
                                     Position = new Vector2( basePos.X,
                                                             basePos.Y - baseDir.Y * 1 * 16 ) } );
        snake.Add( new SnakeBlock { Direction = baseDir,
                                     Position = new Vector2( basePos.X,
                                                             basePos.Y - baseDir.Y * 2 * 16 ) } );
        snake.Add( new SnakeBlock { Direction = baseDir,
                                     Position = new Vector2( basePos.X,
                                                             basePos.Y - baseDir.Y * 3 * 16 ) } );
        snake.Add( new SnakeBlock { Direction = baseDir,
                                     Position = new Vector2( basePos.X,
                                                             basePos.Y - baseDir.Y * 4 * 16 ) } );

        snake[ 0 ].PrevDirection = snake[ 0 ].Direction;
    }
}
```

Következő feladatunk a megfelelő textúra betöltése a szegmensek számára. Minden egyes blokk ugyanazt a textúrát fogja kapni. A feladat egyetlen **Texture2D** példánnyal is megoldható lett volna, azonban szerettem volna egy automatizált megoldást mutatni ciklus segítségével, amely jól használható olyan esetekben, amikor több objektum textúráit töltjük be.

```
public void LoadContent()
{
    foreach ( SnakeBlock block in snake )
        block.Texture = contentMgr.Load<Texture2D>( @"Textures/sprite" );
}
```

A fejezet példáinak a végigolvasása (esetleg végigpróbálása) után a felrajzolás nem okozhat problémát, a legegyszerűbb paraméterezési móddal fogjuk meghívni a **SpriteBatch.Draw()** metódust.

```
public void Draw( GameTime time, SpriteBatch spriteBatch )
{
    foreach ( SnakeBlock block in snake )
        spriteBatch.Draw( block.Texture, new Rectangle( (int)block.Position.X,
                                                         (int)block.Position.Y, 16, 16 ),
                           Color.White );
}
```

Betöltöttük a textúrákat, fel tudjuk rajzolni a szegmenseket, nincs más hátra, mint a frissítést végző **Update()** metódust megvalósítani. Ez egy kicsit nagyobb falat lesz, mint azt előre gondolnánk. Először is a képernyőn gesztusok segítségével tud navigálni a felhasználó, ezt mindenképpen le kell kezelünk. A kígyó irányításához, mindegy, hogy hol ér a képernyőhöz, azt fogjuk nézni, hogy a határozott, irányt mutató mozdulata merre mutat. Mivel az érintésminta csak azt tudja visszaadni, hogy vízszintesen és függőlegesen mekkora változás történt, így az irányvektort magunknak kell meghatároznunk. Arkusztangens segítségével kiszámoljuk a mozdulatból származó irányvektorhoz tartozó szöget, melyet radiánban mérve kapunk vissza. Ezt nekünk valódi szögre kell átváltanunk, amelyet egy segédfüggvény végez (**GetAngle**). Ha a szög megvan, akkor megnézzük, mely tartományba mutatott az irányvektor, és ennek függvényében előállítjuk a kígyó új irányát. A frissítésnek van még egy második része is, ami fél másodpercenként a kígyót továbblépteti a felületen. Amikor az új irányt meghatároztuk, nem azt szerettük volna elérni, hogy a kígyó összes szegmense egyszerre mozduljon el az új irányba, hanem iterációként mindig csak egy szegmens, a rá következő pedig majd egy másik iterációban. Ehhez a **Direction** és a **PrevDirection** tulajdonságokat fogjuk felhasználni, amely a szegmensek között mindig egy elcsúsztatást fog jelenteni.

```
public void Update(GameTime gameTime)
{
    if ( IsPaused )
        return;

    while ( TouchPanel.IsGestureAvailable )
    {
        GestureSample gesture = TouchPanel.ReadGesture();

        if ( gesture.GestureType == GestureType.Flick )
        {
            double angle = GetAngle( gesture.Delta.X, gesture.Delta.Y );

            Vector2 dir = snake[ 0 ].Direction;

            if ( 45 <= angle && angle < 135 )
            {
                if ( dir.Y <= 0 )
                    dir = new Vector2( 0, -1 );
            }
        }
    }
}
```

```

        else if ( 135 <= angle && angle < 225 )
        {
            if ( dir.X <= 0 )
                dir = new Vector2( -1, 0 );
        }
        else if ( 225 <= angle && angle < 315 )
        {
            if ( dir.Y >= 0 )
                dir = new Vector2( 0, 1 );
        }
        else
        {
            if ( dir.X >= 0 )
                dir = new Vector2( 1, 0 );
        }

        snake[ 0 ].PrevDirection = snake[ 0 ].Direction;
        snake[ 0 ].Direction = dir;
    }
}

if ( ( DateTime.Now - lastTime ).TotalMilliseconds >= 500 )
{
    var dir2 = snake[ 0 ].Direction;

    snake[ 0 ].Position += new Vector2( dir2.X * 16, dir2.Y * 16 );
    for ( int i = 1; i < snake.Count; ++i )
    {
        snake[ i ].PrevDirection = snake[ i ].Direction;
        snake[ i ].Direction = snake[ i - 1 ].PrevDirection;
        snake[ i ].Position += new Vector2( snake[ i - 1 ].PrevDirection.X * 16,
                                            snake[ i - 1 ].PrevDirection.Y * 16 );
    }

    snake[ 0 ].PrevDirection = snake[ 0 ].Direction;

    lastTime = DateTime.Now;
}
}

```

Mivel az érintésminta nem tudott szöget meghatározni, ezt magunknak kell megtennünk. A képernyő és a játékunk koordináta-rendszere egymáshoz képest el van forgatva, így a függőlegesen mért értéket invertálnunk kell. Továbbá negatív szögekkel nem szeretnénk számolni, ezért  $2 \times \pi$ , azaz 360 fokkal forgatással számoljuk ki az adott szöget. Ha előállt a megfelelő érték, akkor az még radiánban van, ezért át kell váltanunk valós szögmértékbe.

```

public static double GetAngle( double deltaX, double deltaY )
{
    double num = Math.Atan2( -deltaY, deltaX );
    if ( num < 0.0 )
    {
        num = 6.28318530717959 + num;
    }

    return num * 360.0 / 6.28318530717959;
}

```

Majdnem készen vagyunk a kígyót leíró osztállyal, de a játékunk mit sem érne ütközésetektáció nélkül. Két külön metódust fogunk bevezetni. Az elsőnek annyi a feladata, hogy eldöntse, beleütközött-e a kígyó a falba vagy a saját testébe, ha igen, akkor a megfelelő logikai értéket adja vissza.

A másik metódus feladata pedig az, hogy megnézzze, a kígyó feje érintkezett-e a pályán elszórt kincssel, ha igen, akkor adjon hozzá a kígyó testének legvégéhez egy új szegmenst. Az ütközésvizsgálathoz megint csak jól közelített befoglaló négyszögeket használunk, és azoknak a metszésviszonyát figyeljük.

```
public bool DetectCollision()
{
    var snakePosition = snake[ 0 ].Position;

    if ( snakePosition.X <= 16 || snakePosition.X >= 464 )
        return true;

    if ( snakePosition.Y <= 16 || snakePosition.Y >= 784 )
        return true;

    Rectangle snakeHead = new Rectangle( (int)snake[ 0 ].Position.X,
                                           (int)snake[ 0 ].Position.Y, 16, 16 );

    for ( int i = 1; i < snake.Count; i++ )
    {
        Rectangle snakeBody = new Rectangle( (int)snake[ i ].Position.X,
                                              (int)snake[ i ].Position.Y, 16, 16 );

        if ( snakeHead.Intersects( snakeBody ) )
            return true;
    }

    return false;
}

public bool DetectCollisionWithNewBlock( SnakeBlock newBlock )
{
    Rectangle snakeHead = new Rectangle( (int)snake[ 0 ].Position.X,
                                           (int)snake[ 0 ].Position.Y, 16, 16 );
    Rectangle block = new Rectangle( (int)newBlock.Position.X,
                                      (int)newBlock.Position.Y, 16, 16 );

    if ( snakeHead.Intersects( block ) )
    {
        var prev = snake[ snake.Count - 1 ];
        snake.Add( new SnakeBlock { Position = prev.Position +
            new Vector2( -prev.Direction.X * 16, -prev.Direction.Y * 16 ),
            Direction = prev.Direction, Texture = prev.Texture } );
        return true;
    }

    return false;
}
```

A kígyót leíró osztállyal készen vagyunk. A GamePage.xaml fájlban létre kell hoznunk a szüneteltetés gombot. A korábbi fejezetek során részletesen láttuk, hogy történik ez, így a következő néhány sor kód semmilyen meglepetést nem tartogat. A gombnak explicit méretezést adtam, és a képernyő jobb felső sarkába igazítottam. Továbbá létrehoztam a **Click** eseményvezérlőt is.

```
<phone:PhoneApplicationPage
    x:Class="Snake.GamePage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:phone="clr-namespace:Microsoft.Phone.Controls;assembly=Microsoft.Phone"
    xmlns:shell="clr-namespace:Microsoft.Phone.Shell;assembly=Microsoft.Phone"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    FontFamily="{StaticResource PhoneFontFamilyNormal}"
```

```

FontSize="{StaticResource PhoneFontSizeNormal}"
Foreground="{StaticResource PhoneForegroundBrush}"
SupportedOrientations="Portrait" Orientation="Portrait"
mc:Ignorable="d" d:DesignHeight="800" d:DesignWidth="480"
shell:SystemTray.IsVisible="False">

<Grid x:Name="LayoutRoot">
    <Button Width="140" Height="100"
        HorizontalAlignment="Right"
        VerticalAlignment="Top" Click="Button_Click">Pause</Button>
</Grid>

</phone:PhoneApplicationPage>

```

Mögöttes kódban több új adattagot is létre kell hoznunk. Szükségünk lesz egy új objektumpéldányára a kígyó osztályunknak, kelleni fog egy háttér textúra és egy véletlen szám generátor, amellyel az eleséget fogjuk elszórni a pályán. Ahhoz, hogy ide ne kelljen a későbbiekben visszatérnünk, elhelyeztem egy **UIElementRenderer** adattagot is, amelynek funkcióját a későbbiekben részletesen megismerjük majd, egyelőre elég, ha deklaráljuk.

```

private ContentManager contentManager;
private GameTimer timer;
private SpriteBatch spriteBatch;
private SnakeObject snake;
private SnakeBlock newBlock;
private Texture2D background;
private Random rand = new Random();
private UIElementRenderer uiRenderer;

```

Az osztály konstruktorában engedélyeznünk kell az érintésminták használatát, a tartalomkezelő felhasználásával inicializálnunk kell a kígyó objektumpéldányunkat, és fel kell iratkoznunk az oldal **LayoutUpdated** eseményére is, amely minden egyes alkalommal le fog játszódni, amikor a felület frissül.

```

public GamePage()
{
    InitializeComponent();

    contentManager = (Application.Current as App).Content;

    snake = new SnakeObject( contentManager );

    TouchPanel.EnabledGestures = GestureType.Flick;

    timer = new GameTimer();
    timer.UpdateInterval = TimeSpan.FromTicks(333333);
    timer.Update += OnUpdate;
    timer.Draw += OnDraw;

    LayoutUpdated += new EventHandler(GamePage_LayoutUpdated);
}

```

Amikor az oldalra navigálunk (először a MainPage.xaml-lal indul az alkalmazás, és onnan térünk át a játékosoldalra), akkor be kell töltenünk a tartalmakat, ami a mi esetünkben semmi mást nem jelent, mint a textúrák lekérését a tartalomkezelőtől. Itt történik meg az első felszedhető szegmens (ha az eleség név tetszetősebb, akkor hívhatjuk így is) inicializálása is.

```

protected override void OnNavigatedTo(NavigationEventArgs e)
{
    SharedGraphicsDeviceManager.Current.GraphicsDevice.SetSharingMode(true);
}

```

```
spriteBatch = new SpriteBatch(SharedGraphicsDeviceManager.Current.GraphicsDevice);

snake.LoadContent();
background = contentManager.Load<Texture2D>( "Textures/Background" );
newBlock = new SnakeBlock { Position = new Vector2( rand.Next() % 480,
                                                    rand.Next() % 800 ),
                           Texture = contentManager.Load<Texture2D>( "Textures/sprite" ) };

timer.Start();

base.OnNavigatedTo(e);
}
```

Két nagyon fontos része van vissza a játéknak. A játékciklus két komponensét el kell készítenünk. Először érdemes a frissítést végző **Update()** metódust létrehozni, mivel tartalma egészen egyszerű lesz, az ütközések két különböző típusát fogja figyelni. A sima ütközés lekezelését az Olvasóra bízom, amíg az eleség összeszedéséért felelős ütköztetést még együtt elkészítjük. Nagyon egyszerű feladat! Semmi más nem kell tennünk, amikor az elszórt szegmenst összeszedjük, mint generálni a helyére egy másikat, mivel a kígyó osztály ütközésetektáló metódusa az előző elemet automatikusan a kígyó testéhez fogja fűzni.

```
private void OnUpdate(object sender, GameTimerEventArgs e)
{
    snake.Update( null );

    if ( snake.DetectCollision() )
        ; // ezt a részt az olvasó kreativitására bízom

    if ( snake.DetectCollisionWithNewBlock( newBlock ) )
    {
        newBlock.Position = new Vector2( rand.Next() % 480, rand.Next() % 800 );
    }
}
```

Mielőtt a rajzolást is megvalósítanánk, még két dolog hiányzik. Az egyik a szüneteltetés gomb **Click** eseményvezérlője, a másik pedig a **LayoutUpdated** eseményvezérlő. Az utóbbit miért is hoztuk létre? Az XNA tartalom elsőbbséget élvez a kirajzolás során, és a Silverlight alapon definiált felület automatikusan nem kerül megjelenítésre. Ahhoz, hogy az is kirajzolásra kerüljön, textúrát kell belőle készítenünk, amelyhez a tartalmat folyamatosan frissítenünk kell.

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    snake.IsPaused = !snake.IsPaused;
}

void GamePage_LayoutUpdated(object sender, EventArgs e)
{
    if ( uiRenderer == null || LayoutRoot.ActualWidth > 0
        && LayoutRoot.ActualHeight > 0 )
        uiRenderer = new UIElementRenderer( LayoutRoot,
                                             (int)LayoutRoot.ActualWidth,
                                             (int)LayoutRoot.ActualHeight );
}
```

És a végére maradt a rajzolást végző **OnDraw()** metódusunk létrehozása, melynek feladata a szintér felrajzolása. Nagyon fontos megfelelő sorrendben dolgoznunk. Először is megkérjük a környezetet, hogy renderelje le a Silverlight tartalmat, amely egyelőre nem fog megjelenni. Utána felrajzoljuk a háttér textúrát és a kígyót. Miután ezek is megjelenítésre kerültek, akkor a textúrába renderelt Silverlight tartalmat is egy egyszerű Sprite-ként felrajzoljuk.

```

private void OnDraw(object sender, GameTimerEventArgs e)
{
    SharedGraphicsDeviceManager.Current.GraphicsDevice.Clear(Color.Black);

    uiRenderer.Render();

    spriteBatch.Begin();

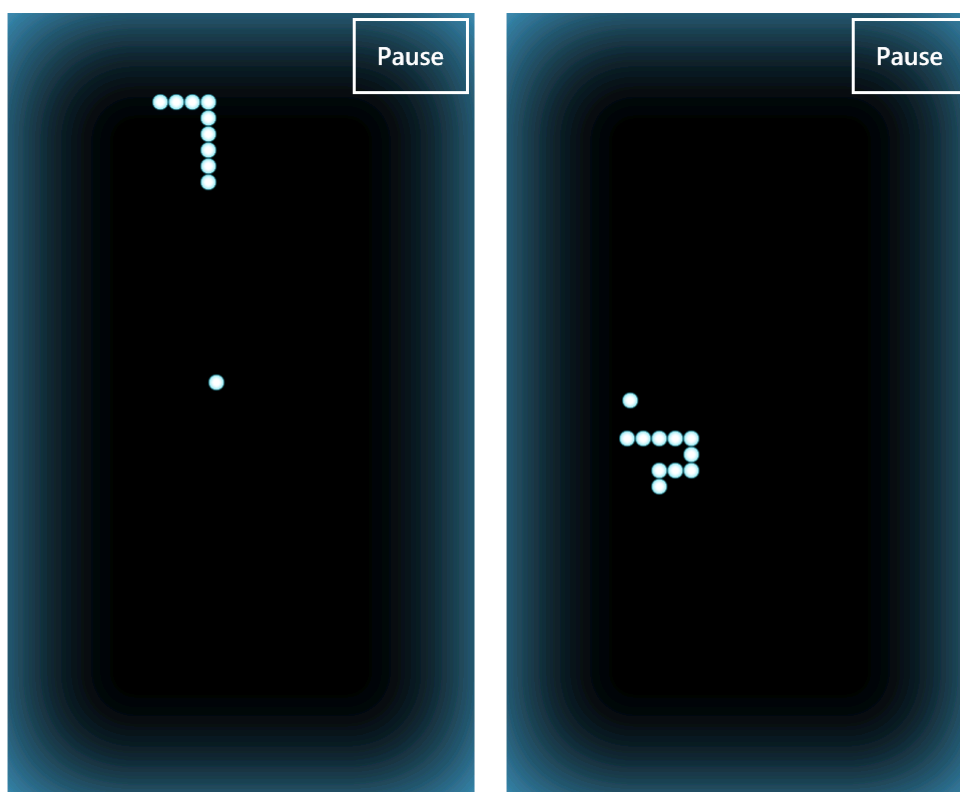
    spriteBatch.Draw( background,
        new Microsoft.Xna.Framework.Rectangle( 0, 0, 480, 800 ), Color.White );
    spriteBatch.Draw( newBlock.Texture,
        new Microsoft.Xna.Framework.Rectangle( (int)newBlock.Position.X,
                                                (int)newBlock.Position.Y, 16, 16 ),
        Color.White );
    snake.Draw( null, spriteBatch );

    spriteBatch.End();

    spriteBatch.Begin();
    spriteBatch.Draw( uiRenderer.Texture, Vector2.Zero, Color.White );
    spriteBatch.End();
}

```

Ha készen vagyunk, sehol nincsenek már a projektben szintaktikai hibák, akkor fordítsuk le, és futtassuk emulátor segítségével! A 11-21 ábrán látható eredményt kell kapnunk. Határozott mozdulatokkal irányítható a kígyó, és a Pause gomb lenyomásának hatására a játék szüneteltetésre kerül, illetve újraaktiválódik.



11-21 ábra: A Snake játék futás közben

## Összefoglalás

Ebben a fejezetben megismerkedtünk a Windows Phone platformot alkotó technológiák felépítésével, és betekintést nyertünk a számítógépes (vagy mondhatnám inkább okostelefonokra tervezett?) játékok

fejlesztésébe. Éppen csak a felszínt érintve áttekintettük a számítógépes grafika elméleti hátterét és ennek kapcsán a játékokat alkotó grafikus megoldásokat is. Megismerkedtünk az XNA Framework alapvető használatával, illetve láttuk a Mangóban újdonságként bevezetett hibrid renderelési rendszer működését és használatát is. Az utóbbira építve elkészítettük egy Snake játéknak az alapjait is, melyet az Olvasó tetszése szerint tovább bővíthet.



# 12. Marketplace

Ebben a fejezetben megismerkedhetünk a Windows Phone 7 ökoszisztémájának központi szereplőjével, a Marketplace-szel. A telefon és a rajta lévő operációs rendszer bármennyire is jó, üres lenne alkalmazások nélkül. A Windows Phone 7-hez tartozik egy piactér, ahonnan alkalmazásokat tölthetünk le, illetve fejlesztőként publikálhatjuk saját alkalmazásainkat. Azok lehetnek ingyenesek vagy díjkötelesek, és bárki publikálhat, aki megfelel a Marketplace feltételeinek. Ebben a részben a következő témákat nézzük át:

- Mi az a Marketplace?
- Készüléken történő hibakeresés
- Készülék regisztráció
- Publikálás
- Screenshot készítés
- Marketplace Test Kit
- Képek és ikonok
- Marketplace Task-ek
- Trial Mode

## APPHUB

Az AppHub a Microsoft Windows Phone 7 fejlesztői menedzsment oldala, ahol a fejlesztők kezelhetik az alkalmazásaikat, frissíthetik azokat, nyomon követhetik azok népszerűségét, és követhetik az értékesítésükből befolyt összegeket is. Az AppHub a <http://create.msdn.com> címen érhető el. A fejlesztői regisztráció éves díja jelenleg 99 USD, ami három fejlesztői készülék használatát (*developer unlock*) tartalmazza és a piactérre való publikálás lehetőségét.

Az AppHubra háromféle regisztrációtípus létezik:

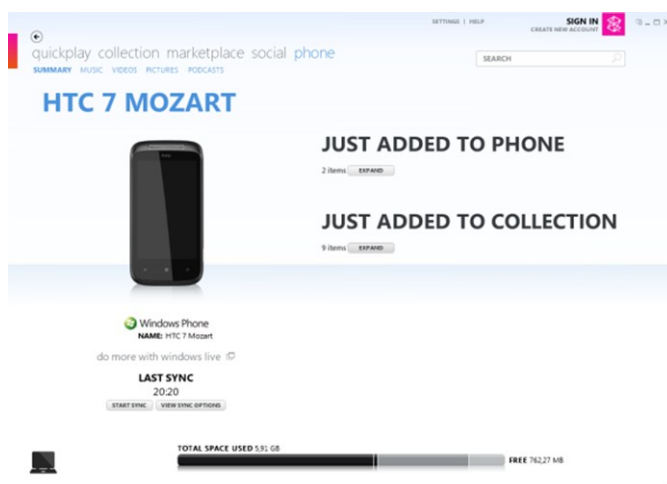
1. Cég nevében (akkor használatos, ha a fejlesztő egy cég, vagy szervezet)
2. Magánszemély (akkor alkalmazzuk, ha egy egyszerű magánszemélyről van szó)
3. Diákként (akkor válasszuk ezt a lehetőséget, ha aktív státuszú középiskolai vagy felsőoktatási hallgatók vagyunk. Ebben az esetben a DreamSpark program –<http://dreamspark.com> – keretein belül a 99 USD éves díjat nem kell kifizetnünk, és ingyenesen publikálhatunk, akár díjköteles alkalmazásokat is!)

A regisztrációnál szükségünk lesz egy Windows Live ID-ra, cégek és a magánszemélyek esetén egy bankszámlára is. Cégek esetében a regisztrációt követően egy GeoTrust nevű cég fogja felvenni a kapcsolatot annak érdekében, hogy ellenőrizze, a vállalat tényleg az általa megadott országban tevékenykedik. A regisztráció magánszemélyek esetén 1-2 napot, cégek esetén olykor 3-4 napot is igénybe vehet.

## Készüléken történő hibakeresés és regisztráció

Könyvünk eddigi fejezeteiben az alkalmazások fejlesztésénél csak az emulátorra koncentráltunk, viszont az alkalmazásainkat célszerű kipróbálni egy fizikai készüléken is, mielőtt publikálnánk azt. Egyfelől ez azért hasznos, mert felmérhetjük, hogy használhatóság szempontjából milyen az alkalmazásunk, másfelől azért, mert az emulátor és a fizikai készülék között komoly eltérések lehetnek a teljesítményt illetően. Ezért minden esetben, ha elkészítünk egy alkalmazást, célszerű azt kipróbálni egy fizikai készüléken is. Megnehezíti azonban a dolgunkat, hogy a Windows Phone 7 -nél már nemcsak egy egyszerű fájl másolásából áll a telepítés, mint a Windows Mobile esetében, hanem már csak az a fejlesztő férhet a

készülékhez, aki rendelkezik fejlesztői fiókkal (*developer account*). Ahhoz, hogy legyen egy ilyen előfizetésünk, regisztrálnunk kell magunkat a Marketplace-en. Ha a regisztrációt elvégeztük, USB kábellet kössük össze a készülékünket a számítógépünkkel! Figyeljünk arra, hogy a Zune szoftvernek ([www.zune.net](http://www.zune.net)) telepítve kell lennie! Amikor összekötöttük a Windows Phone 7-et a géppel (12-1 ábra), akkor automatikusan el fog indulni a Zune szoftver (ha ez valamilyen oknál fogva nem történne meg, indítsuk el manuálisan!), majd ezt követően el kell indítanunk a **Windows Phone Developer Registration tool** nevezetű alkalmazást (ez az alkalmazás a Windows Phone 7 SDK része). Miután elindítottuk, adjuk meg a Live ID-nkat – azt, amellyel korábban a Marketplace-re regisztráltunk –, majd ha ez megtörtént, kattintsunk az **Unregister** gombra! Ha minden rendben zajlott, a telefonunk fejlesztői zárolása feloldásra került (12-2 ábra).



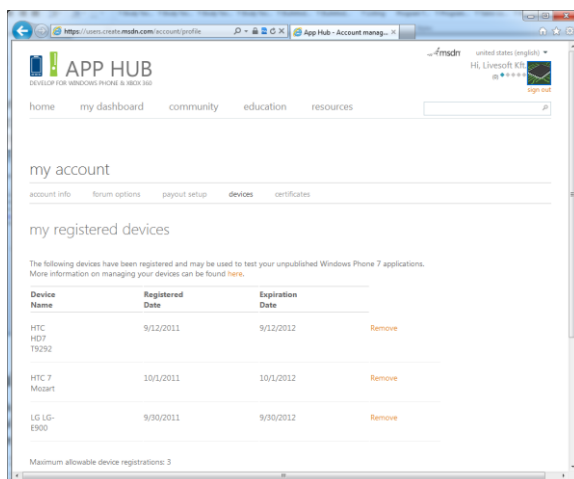
12-1 ábra: Zune szoftver

Fontos! Csak úgy tudjuk unlockolni a készüléket, ha a képernyőzár fel van oldva!



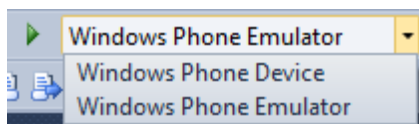
12-2 ábra: Windows Phone Developer Registration Tool

Az AppHub-on a saját fejlesztői fiókunk **devices** menüpontja alatt láthatjuk a zárolás alól feloldott készülékeink listáját. Abban az esetben, ha esetleg lecserélnénk a jelenlegi készülékünket egy újabbra, akkor ezen az oldalon tudjuk eltávolítani a zárolás feloldását is. Figyeljünk arra, hogy ha a készülékünket frissítettük vagy reseteltük, akkor a developer unlock is eltűnik, ilyenkor ezt a folyamatot meg kell ismételni!



**12-3 ábra: AppHub a feloldott fejlesztői zárolású készülékek listájával**

Ha minden rendben zajlott, akkor már lehetőségünk van az alkalmazásunkat a fizikai készülékünkön is tesztelni. Ez mind Visual Studio-ból, mind Expression Blendből egyszerűen megoldható. Abban az esetben, ha a fejlesztői környezetből publikáljuk a készülékre az alkalmazást, a debugger rögtön elindul és ugyanúgy, ahogy az emulátornál, hibát tudunk keresni az alkalmazásban. Ehhez mindössze a Visual Studio-ban kell a kimenetet átállítani Windows Phone Emulator-ról Windows Phone Device-ra. Ezt követően az F5-öt megnyomása után már a készülékre kerül fel az alkalmazás, és ott tudunk hibát keresni (12-4 ábra).

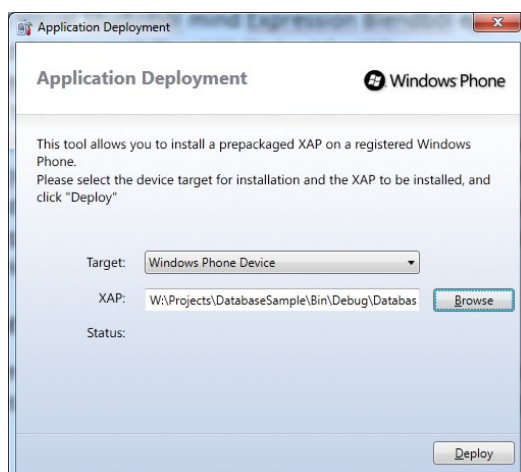


**12-4 ábra: Készülékre történő publikáció**

Figyelnünk kell arra is, hogy az alkalmazás Visual Studio-ból történő telepítésekor is fel kell oldani a képernyőzárat! Ha ezt nem tennénk meg, akkor „Access is denied” hibát kapnánk.

## Alkalmazások telepítése

Abban az esetben, ha csak egyszerűen telepíteni szeretnénk egy már lefordított alkalmazást, amelynek kezünkben van az XAP fájlja, akkor ezt az **Application Deployment Tool**-al tehetjük meg, ami szintén a Windows Phone 7 SDK része. Az alkalmazás felépítése nagyon egyszerű: először megmondjuk, hogy melyik eszköz a cél (target) – a készülék vagy az emulátor –, majd kikeressük a telepítendő XAP fájlt, és megnyomjuk a Deploy gombot (12-5 ábra). Ha minden rendben zajlott, akkor az adott alkalmazás megjelenik a készülék vagy az emulátor alkalmazásai között. Itt is fontos megjegyezni, hogy a művelet során a telefon képernyője nem lehet zárolva!



**12-5 ábra: Az Application Deployment Tool**

A készülékre maximum tíz alkalmazást tudunk feltelepíteni a feloldott fejlesztői zár birtokában. Ha ennél többre lenne szükségünk, akkor egy előző, általunk telepített alkalmazást el kell távolítanunk.

### ChevronWP7

A Microsoft támogatásával elindult a <http://labs.chevronwp7.com/> oldal (12-6 ábra), ahol a hobbifejlesztők feloldhatják telefonjuk zárolását, függetlenül attól, hogy milyen régióba tartoznak, vagy hogy az adott országból elérhető-e a Marketplace. Bár ez a szolgáltatás nem ingyenes, 9 USD-be kerül telefononként, mégis hasznos lehet. A 9 USD-t egyszer kell kifizetnünk, nem kell évente megújítanunk, mint a Marketplace esetén. Ezzel a feloldással csak a telefonra tudunk alkalmazásokat telepíteni, a Marketplace-re nincs jogunk azokat feltölteni! Ahhoz, hogy használni tudjuk, egy Live ID-re lesz szükségünk. Fontos kiemelni, hogy ez nem ún. *interop unlock*, azaz nem kapunk hozzáférést a telefon mélyebb részeihez. Ezzel a feloldással is csak maximum 10 alkalmazást telepíthetünk a készülékre (diákok 3 alkalmazást).

CHEVRONWP7 LABS

## unlock your Windows Phone

We believe Windows Phone development should be accessible to anyone. We are providing a Windows Phone developer unlocking service to developers across all skill levels and regions for just \$9 USD per phone.



### get started

We use Windows Live to manage your ChevronWP7 Labs account.

If this is your first time, sign in with an existing Live ID and register for a ChevronWP7 profile. If you've been here before, just sign in using the Live ID you previously registered with.

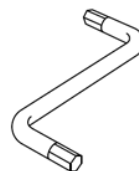
 Sign in using Windows Live



1x



1x



1x

**12-6 ábra: Chevronwp7 labs**

Ha van Marketplace előfizetésünk, általában felesleges ezt a szolgáltatást megrendelnünk.

## Publikálás

Ha kész az alkalmazásunk, itt az ideje publikálni a Marketplace-en! Ahhoz, hogy publikáljuk, az alkalmazásunknak több funkcionális és formai követelménynek meg kell felelnie. Ezeket az alábbi 4 pontban foglalhatjuk össze:

1. Az alkalmazásnak válaszképesnek kell lennie.
2. Az alkalmazásnak hatékonyan kell használni a telefon erőforrásait.
3. Az alkalmazásunk nem befolyásolhatja a telefonunk megfelelő működését.
4. Az alkalmazásunkban nem lehet rosszindulatú kód (Maleware).

Ez meglehetősen általános leírás arról, hogy mit nem tehet az alkalmazásunk, de a fentiekén kívül elég sok feltételt kell még teljesítenünk ahhoz, hogy az a piacon megtalálható legyen! A következő néhány lépésben bemutatjuk a publikálás folyamatát. Nézzük meg lépésről lépésre, hogy kell egy már elkészült alkalmazást publikálni a piacon! Ehhez öt egyszerű lépést kell elvégezni.

Látogassunk el az AppHub-ra: <http://create.msdn.com>, és a bejelentkezés után kattintsunk a Submit for Windows Phone gombra (12-7 ábra)!



12-7 ábra: AppHub oldal – Submit for Windows Phone

A következő ablak az Upload. Itt kell feltöltenünk az alkalmazásunkat, és egyedi nevet is itt kell választanunk a számára!

Nézzük meg részletesen, milyen információt kell és lehet ezen az oldalon megadnunk (12-8 ábra):

- **App name for App Hub:** Az alkalmazás neve, mely csak az AppHubon fog megjelenni. Ennek a névnek egyedinek kell lennie!
- **Distribute to:** Két menüelem közül választhatunk, az első a **Public Marketplace**, amivel az alkalmazásunkat mindenki számára elérhetővé tesszük, a másik lehetőség pedig a **Private beta test**, ami akkor lehet hasznos, ha csak meghatározott személyeknek szeretnénk az alkalmazást elérhetővé tenni, például tesztelési célból.
- **Browse to Upload a file:** Itt kell feltöltenünk az XAP fájlunkat.
- **Application version number:** Itt határozhatjuk meg az alkalmazásunk verzióját. Az alkalmazások verziózása tetszőleges. (Kezdhethetjük akár 5.0-val is.)
- **Requires technical exception:** Ha ezt az opciót megjelöljük, akkor felmentést kérünk, mivel az alkalmazásunk valamilyen speciális oknál fogva nem felel meg a „Windows Phone 7

Application Certification Requirements” által támasztott követelményeknek. Az alkalmazásunk elbírálása ilyenkor egyedi, és jóval hosszabb ideig tart, mint egy átlagos alkalmazás esetén.

### App Submission



#### submit an app!

Let's get started. Distribute your app by giving it a name and uploading the app package. You can also learn what to expect during this [submission and certification process](#).

##### \* Required fields

\* App name for App Hub:   
App name only visible in App Hub

\* Distribute to: ☒ Public Marketplace  
☐ Private Beta Test. Learn more about [beta testing](#).

\* Browse to upload a file:  [Browse](#)  
Max size: 225 MB  
Expected format: \*.xap

\* App version number:  1 .  0

Requires technical exception? ☐ Submitting a technical exception will add several days to the certification approval process unless you have been previously approved. Additionally, exception request approval is not guaranteed. [What is a technical exception and why do I need it?](#)

Save and Quit

Next

#### 12-8 ábra: Alkalmazás feltöltéséhez szükséges információ az AppHubon

Ha az oldal adatait kitöltöttük, kattintsunk a Next gombra! Néhány másodperc múlva betöltődik a **Describe** oldal (12-9 ábra). Itt írhatjuk le részletesen, hogy az alkalmazásunk mire való, milyen funkciói vannak, és a hozzá tartozó képeket, ikonokat is itt adhatjuk meg. Fontos megjegyezni, hogy ha többnyelvű az alkalmazás, akkor minden nyelvhez külön-külön leírást kell adnunk! Nézzük át, hogy milyen tulajdonságokat lehet és kell kitöltenünk!

- **Kategória:** Itt választhatjuk ki, hogy az alkalmazás milyen kategóriába és alkategóriába (**Subcategory**) tartozik. Ezt kötelező kitölteni.
- **App name:** Ez a mező automatikusan töltődik ki. Ezt az információt az XAP fájlból nyeri ki a rendszer.
- **Short Description:** Itt az alkalmazásunk rövid leírását kell megadnunk. Ez maximum 25 karakter lehet, ennek segítségével hívhatjuk fel legelőször az alkalmazásunkra a felhasználók figyelmét. Ezt a mezőt kötelező kitölteni.
- **Detailed Description:** A programunk részletes leírását adhatjuk meg, ez maximum 2000 karakter lehet, és kötelező kitölteni.
- **Keywords:** Meghatározhatunk maximum 5 kulcsszót, amivel a felhasználók könnyebben megtalálhatják az alkalmazásunkat. Ezt a mezőt kötelező kitölteni.
- **Legal URL:** Egy olyan URL cím, amely az alkalmazásunkkal kapcsolatos licencekre, illetve egyéb dokumentációra mutat, kitöltése opcionális.
- **Email address:** Egy olyan e-mail cím, amivel a felhasználók kapcsolatba léphetnek az alkalmazás fejlesztőivel, például az alkalmazás-támogatási csoporttal. Ez is opcionális, de ajánlott a kitöltése.

## Category

\* Required fields

\* Category

\* Subcategory

## Details

English →

English app name: PhoneTrial

Short description:

\* Detailed description:

\* Keywords:

help choosing effective keywords

Legal URL:

Email address:

For app support

### 12-9 ábra: A Describe fül alatt adjuk meg az alkalmazásunk leírását

Az alapadatokat ezzel meg is adtuk, most már csak az alkalmazásunkkal kapcsolatos képeket és ikonokat kell feltöltenünk. Nézzük meg sorban, mit is lehet megadni (12-10 ábra)! A képek ajánlott felbontása 96DPI. Ezek a képek nem lehetnek transzparenssek!

- **Large mobile app Tile (173x173 Fájl):** Egy png fájl, a vásárlók ezt az ikont látják meg először a piactéren, amikor az alkalmazásunkat betöltik. Opcionálisan választható a megadása.
- **Small mobile app Tile (99x99 Fájl):** Ezt a képet a felhasználók akkor látják, amikor egy katalógusban vagy egy keresési listában böngésznek, kötelező megadni.
- **Large PC app Tile (200x200 PNG fájl):** Ezt az ikont látják a felhasználók, ha a Zune szoftverből böngészik a piacteret. A kép megadása kötelező.
- **Background art (1000x800 PNG fájl):** Ez a kép akkor látható, ha az alkalmazásunk ki van emelve a Marketplace-en (ún. *featured app*). Ebben az esetben ez a kép jelenik meg háttérképként. Ezt a képet láthatjuk a Zune-ból is, ha az alkalmazásunk részletes nézetét vizsgáljuk.
- **Screenshots( 480x800 PNG fájlok):** Minimum egy, maximum nyolc képet adhatunk meg, amelyek az alkalmazásunkat működés közben mutatják be. Ezeket mindenképpen érdemes megadnunk, mert a felhasználók sokszor ezek alapján döntenek. Fontos megjegyezni, hogy ezeket a képeket nem szabad grafikai szoftverrel módosítani. A screenshots készítéséről a következő alfejezetben olvashatunk.

### Artwork

[See how these images are used.](#)

Select artwork images: **Browse**  
expected format \*.png, 96dpi  
resolution

Large mobile app tile  
173x173 px



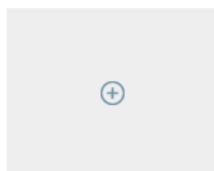
\* Small mobile app tile  
99x99 px



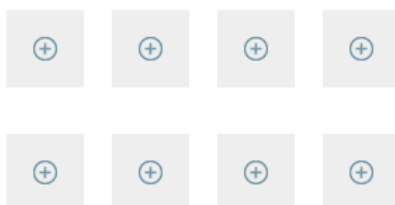
\* Large PC app tile  
200x200 px



Background art  
1000x800 px



\* In app screenshots (8)  
First screenshot is required  
480x800 px



Previous

Save and Quit

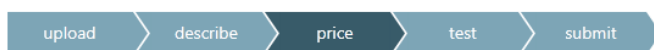
Next

### 12-10 ábra: Itt adhatjuk meg az alkalmazásunkhoz tartozó képeket és képernyőmentéseket

Ha a **Describe** oldalon minden lényeges információt kitöltöttünk, akkor kattintsunk a Next gombra! Néhány pillanat múlva betöltődik a **Price** ablak. Itt tudjuk beállítani azt, hogy az alkalmazásunk mely piacokon legyen elérhető és hogy mennyiért. Az ár meghatározásánál vegyük figyelembe, hogy a teljes ár 70%-át kapjuk meg (bizonyos országoknál a helyi adókat is be kell fizetnünk), tehát ha egy alkalmazást mi például 1 USD-ért szeretnénk eladni, az nálunk 0,7 centben realizálódik. Ezen az oldalon a következő dolgokat állíthatjuk be (12-11 ábra):

- **Enable trials to be downloaded:** Meghatározhatjuk, hogy az alkalmazásunkat trial (próba) módban engedélyezzük-e letölteni. A fejlesztőnek kell meghatároznia, hogy a trial módban mi az, amit engedélyez, illetve mit tilt le az alkalmazás funkcionalitásából. A Trial mód használatával a fejezetünk későbbi részében ismerkedhetünk meg.
- **Set Price tier:** Itt határozhatjuk meg az alkalmazásunk árát. Az ár 0.99 USD és 499.99 USD között lehet. Természetesen ingyenesen is feltölthetünk alkalmazásokat, ilyenkor a „0.00” értéket válasszuk!
- **WorldWide distribution:** Kiválaszthatjuk, hogy az alkalmazásunkat az összes olyan országban elérhetővé tesszük-e, ahonnan a Marketplace elérhető, vagy egyesével megadhatjuk, hogy mely országban vagy régióban publikáljuk az alkalmazásunkat.





## set price and market availability

Set the price tier and trial options, and select the markets you want to distribute your app to. Customers purchasing your app will see an approximate price equivalent in their currency. Sales in multiple countries may settle in different amounts due to currency fluctuations and resulting adjustments to price tiers over time.

Learn more about [taxes](#) and other worldwide legal responsibilities.

Learn about [game ratings](#) on how to get certification.

Select Price tier:  HUF

Enable trials to be downloaded: ☐

[select all](#) Worldwide distribution

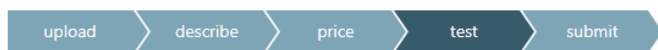
	Region
<a href="#">▶ select all</a>	Africa, Asia and the Pacific
<a href="#">▶ select all</a>	Europe
<a href="#">▶ select all</a>	North and South America

### 12-11 ábra: Price oldal – az alkalmazásunk árát és piacterületét határozhatjuk meg

Ha kitöltöttük a megfelelő adatokat, akkor kattintsunk a Next gombra! Ezt követően a **Test** ablakra jutunk. Itt néhány teszteléssel kapcsolatos adatot adhatunk meg (12-12 ábra):

- **Test notes or instructions:** A mező kitöltése opcionális. Akkor kell kitöltenünk, ha az alkalmazásunk tesztelését végző személlyel speciális információt szeretnénk megosztani.
- **Publishing options:** Azt határozhatjuk meg, hogy mikor kerüljön az alkalmazás a piactérre.
  - **None:** Ezzel a beállítással nem lehet publikálni az alkalmazást. Menthethjük a folyamatot, és ezt követően bármikor visszatérhetünk az oldalra az alkalmazást publikálni.
  - **As soon as it's certified:** Ez egy automatikus folyamat, mellyel az alkalmazásunkat rögtön publikálják a Marketplace-re, amint sikeresen átment az ellenőrzéseken.
  - **As soon as it's certified, but make it hidden:** Hasonlóan az előző beállításhoz, itt is azonnal a Marketplace-re kerül az alkalmazás, ha átmegy a teszteken, de csak azok a felhasználók érhetik el, akikkel mi megosztjuk az alkalmazás elérhetőségét (ehhez szükségünk lesz egy ún. *deep link*-re.)  
Ezt akkor használjuk, ha az alkalmazásunkat csak egy szűk réteg fogja használni. (pl: egy cég számára készítünk egyedi fejlesztést).
  - **I will publish it manually after it has been certified:** A sikeres ellenőrzés után saját magunk kézzel kell, hogy engedélyezzük a piactérre kerülést. Ezt a funkciót „my dashboard” menüpont alatt érhetjük el.

Ha ezzel is készen vagyunk, kattintsunk a Submit gombra, ekkor a soron következő köszöntő képernyő fogad. Az elfogadás folyamata néhány napot igénybe vehet: általában 1-2 nap, de ha esetleg 4-5 nap lenne, akkor sem kell aggódnunk.



### app testing and certification

Enter optional test instructions and select your publication options.

**\* Required fields**

Test notes or instructions:

**\* Publish options:** choose how and when to publish your app. [Learn more about publish options.](#)

As soon as it's certified ▼

- none
- As soon as it's certified
- As soon as it's certified, but make it hidden
- I will publish it manually after it has been certified

Previous

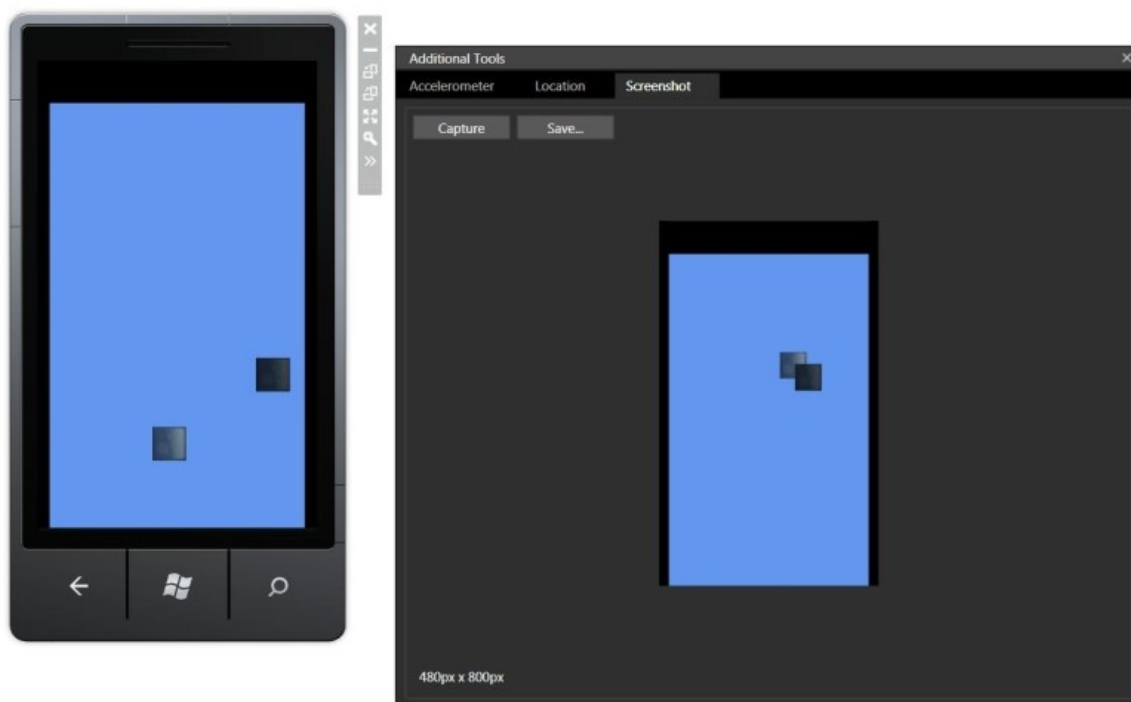
Save and Quit

Submit

**12-12 ábra: Az alkalmazás beküldése**

### Screenshot készítés

Az alkalmazásunk feltöltése során a Marketplace megköveteli, hogy legalább egy 480x800-as képernyőkép legyen az alkalmazásunkról mellékelve. A méretnek pontosnak kell lennie, és a képet nem szabad módosítanunk semmilyen grafikai eszközzel! A képernyőkép készítéséhez persze elég sokféle eszközt tudunk felhasználni: például **Snipping tool**, **OneNote**, de a legtöbbjük használata legtöbbször kényelmetlen. A neten persze található erre is jó néhány eszközt, de a Windows Phone 7.1 SDK-ja óta ez a funkció már az emulátor mellett is megtalálható (12-13 ábra).



**12-13 ábra: Screenshot készítése**

Nézzük meg használat közben! Indítsuk el az alkalmazást, majd a nagyítási beállításokat (nagyító ikon) állítsuk 100%-ra, és kapcsoljuk ki a Frame rate számlálót is! Ha ez megtörtént, akkor indítsuk el az

alkalmazást, és navigáljunk arra az oldalra, amiről a screenshotokat szeretnénk elkészíteni, majd az emulátor jobb felső sarkában található menüben válasszuk ki az Additional Tool ablakot (12-14 ábra)!



**12-14 ábra: Additional Tool**

A megjelenő ablakban válasszuk ki a Screenshot fület, és ezen kattintsunk a Capture gombra! Ezt követően már csak el kell mentenünk ezt a képet a „Save...” gomb segítségével.

## **Windows Phone Marketplace Test Kit**

A Windows Phone 7.1 SDK –val érkezett egy hasznos eszköz (Marketplace Test Kit) is, amivel megvizsgálhatjuk, hogy az alkalmazásunk megfelel-e a Marketplace követelményeinek.

Mit is árul el ez az eszköz alkalmazásunkról?

- Megfelelő-e a képek és a képernyőképek a követelményeknek?
- Megfelelő-e az XAP fájl mérete (maximum: 225 MB)?
- Elég gyorsan indul-e az alkalmazásunk?
- Nem használ-e túl sok memóriát?
- Rendeltetésszerűen működik-e a Back gomb?
- Nem használ-e olyan API-t, ami nem engedélyezett a számunkra?
- Használ-e olyan API-t, ami nem engedélyezett egy Background Agenttel?
- Nincs-e váratlan hiba, ha bezárjuk az alkalmazást?
- Stb.

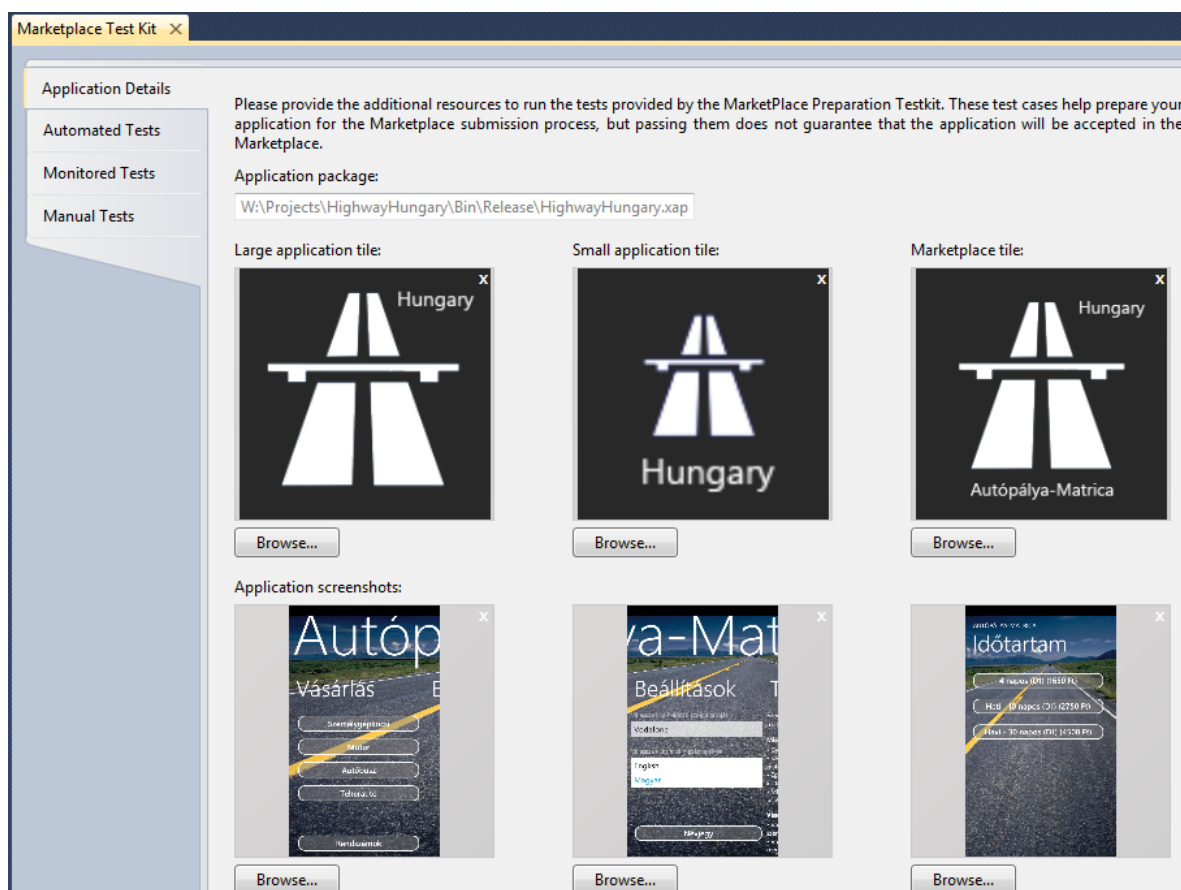
A Test Kit felhasználói felületén négy fül található, ahol különböző teszteket futtathatunk le:

- Application Details
- Automated Tests
- Monitored Tests
- Manual Test

Nézzük meg, mire alkalmasak ezek a tesztek!

### **Application Details**

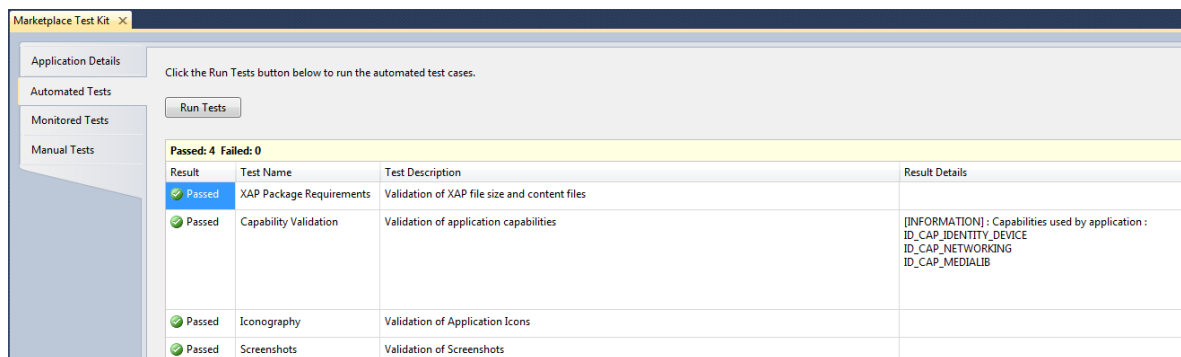
Az Application Details fülön (12-15 ábra) adhatjuk meg az ellenőrzés során használt, az alkalmazásunkhoz tartozó képeket, ikonokat és képernyőképeket. Az Application package mezőbe az XAP fájl útvonala található, az alapbeállítás szerint a **Bin\Release** mappában található XAP-ra mutat. A tesztek lefuttatásához egy Release konfigurációval történő fordításra is szükségünk van. Ezenkívül ezen a fülön adhatjuk meg az alkalmazások piacra feltöltendő képeit és képernyőképeit. Ha nem adjuk meg ezeket a képeket, akkor a tesztünk sikertelen lesz.



12-15 ábra: Application Details (Marketplace Test Kit)

### Automated Tests

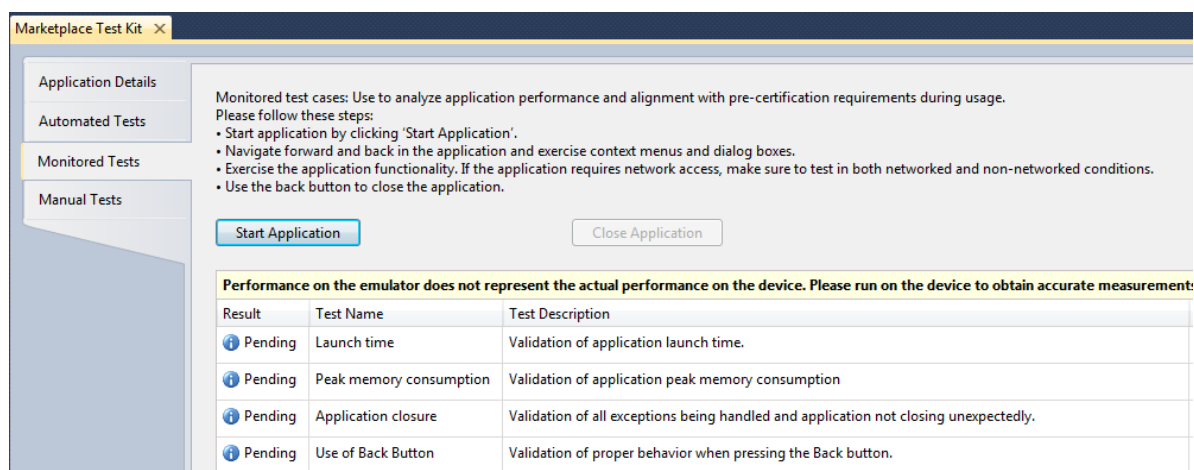
Az Automated Tests fülön (12-16 ábra) az általános kritériumokat tudjuk tesztelni, például az alkalmazás méretét és képeit. Ha lefuttatjuk a tesztet, akkor a teszt megmondja, hogy az alkalmazásunk számára milyen Capability beállításokat (kéességeket) kell meghatározunk a **WPAppManifest.xml** fájlban. Az **Application Details** alatt megadott képeket is itt teszteljük.



12-16 ábra: Automated Tests (Marketplace Test Kit)

### Monitored Tests

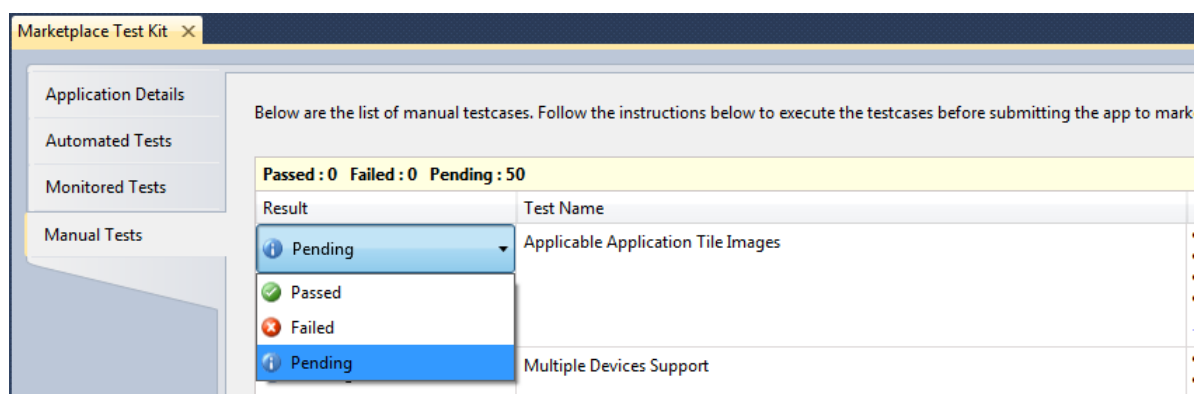
Ezen a fülön (12-17 ábra) mérhetjük meg az alkalmazásunk teljesítményét és válaszképességét. A teszt során az alkalmazást egy fizikai készüléken kell kipróbálnunk, tehát csak a fejlesztői zárolás feloldásával ellátott készüléken tudjuk ezt megtenni. A teszt közben a program által felkínált lehető legtöbb felhasználói esetet hajtsuk végre az alkalmazáson!



12-17 ábra: Monitored Tests (Marketplace Test Kit)

## Manual Test

A tesztnek van egy manuális része is, itt minden egyes tesztet végre kell hajtanunk. A Test Descriptionban leírt feltételeknek teljesülniük kell. Ha ezek teljesültek, akkor egyszerűen a Result oszlop értékét állítsuk át **Passedra**, ha nem teljesültek, akkor **Failedre** (12-18 ábra)!



12-18: Manual Tests (Marketplace Test Kit)

Ha ezeket a teszteket minden publikálás előtt elvégezzük, akkor nagy valószínűséggel semmilyen probléma nem lesz a piactér publikálási folyamata során sem. Célszerű minden kész alkalmazásnál lefuttatni ezeket a teszteket! Időt spórolhatunk vele, hisz kellemetlen, ha például egy transzparens kép miatt visszadobják az alkalmazást.

## Ikonok

Egy Windows Phone 7 –es solution úgy épül fel, hogy három speciális képfájl található benne:

- az **ApplicationIcon.png**, egy 62x62 pixeles kép, amely az alkalmazás ikonja, és az alkalmazások listájában fog látszódni;
- a **Background.png** egy nagyobb, 173x173 pixeles kép, amely a fő képernyő csempéi között jelenik meg, ha az alkalmazást a felhasználó kirakja a főképernyőre;
- ezenkívül van még egy **SplashScreenImage.jpg**, amely egy 480x800 pixeles kép, amit az alkalmazásunk indulásakor, illetve visszalépéskor láthat a felhasználó.

Abban az esetben, ha az alkalmazásunk nagyon gyorsan be tud tölteni, felesleges a **SplashScreenImage**. Ezt azért használjuk, hogy ha lassan töltődik be valamiért az alkalmazás, akkor a felhasználó ne nyomja meg rögtön a Back gombot. Célszerű ezeket a képeket mindig testreszabni. Figyeljünk oda, hogy minden kép Build Action tulajdonsága Content értékű legyen! Ha saját képet adunk

az alkalmazáshoz, akkor annak szintén a Content értéket kell felvennie! Persze nem muszáj ezeket a neveket használnunk, a beállításokat a **WMAppManifest.xml** fájlban testreszabhatjuk.

File	Méret (Pixel)	Fájl típus	Project File	Példa
Small app icon <i>ApplicationIcon.png</i>	62 x 62	PNG	<ul style="list-style-type: none"> <li>App.xaml</li> <li>ApplicationIcon.png</li> <li>Background.png</li> <li>MainPage.xaml</li> <li>SplashScreenImage.jpg</li> </ul>	
Large app tile <i>Background.png</i>	173x173	PNG	<ul style="list-style-type: none"> <li>App.xaml</li> <li>ApplicationIcon.png</li> <li>Background.png</li> <li>MainPage.xaml</li> <li>SplashScreenImage.jpg</li> </ul>	

```

<Deployment xmlns="http://schemas.microsoft.com/windowsphone/2009/deployment"
AppPlatformVersion="7.1">
  <App xmlns="" ProductID="{1b0f2313-d341-4b64-b7b7-88caa0ed57b1}"
    Title="MarketPlace Sample" RuntimeType="Silverlight" Version="1.0.0.0"
    Genre="apps.normal" Author="Turóczy Attila" Description="Sample description"
    Publisher="Livesoft">
    <IconPath IsRelative="true" IsResource="false">ApplicationIcon.png</IconPath>
    <Capabilities>
      <Capability Name="ID_CAP_CONTACTS"/>
      <Capability Name="ID_CAP_APPOINTMENTS"/>
    </Capabilities>
    <Tasks>
      <DefaultTask Name="_default" NavigationPage="MainPage.xaml"/>
    </Tasks>
    <Tokens>
      <PrimaryToken TokenID="MarketPlaceSampleToken" TaskName="_default">
        <TemplateType5>
          <BackgroundImageURI IsRelative="true"
            IsResource="false">Background.png</BackgroundImageURI>
          <Count>0</Count>
          <Title>MarketPlace Sample</Title>
        </TemplateType5>
      </PrimaryToken>
    </Tokens>
  </App>
</Deployment>

```

Ha jól megnézzük ezt a fájlt, akkor láthatjuk, hogy itt az alkalmazásunkkal kapcsolatban speciális tulajdonságokat adhatunk meg. Leírhatjuk, hogy mi legyen az alkalmazás címe (**Title**), milyen legyen a háttere (**Background**) vagy éppen az **ApplicationIcon** neve és elérhetősége, sőt azt is, hogy melyik lapot töltsse be az alkalmazás indulásakor.

## Marketplace Task

Az előző fejezetekben már foglalkoztunk a *launcherekkel*, de még nem néztük meg közelebbről a piactérre jellemzőeket. Ebben a részben áttekinthetjük, melyek ezek, és hogyan is használhatjuk fel azokat. Az alkalmazásunkból elindíthatjuk a Marketplace alkalmazást. Ehhez négy taszk áll a rendelkezésünkre:

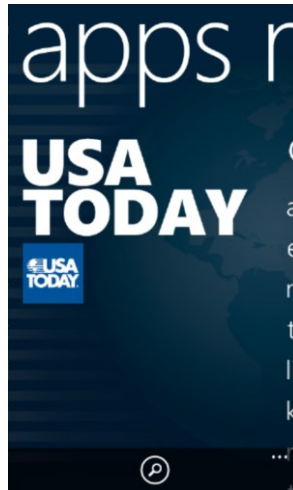
- **MarketplaceHubTask**
- **MarketplaceDetailTask**

- **MarketplaceReviewTask**
- **MarketplaceSearchTask**

Alapvetően két különböző helyen tudunk keresni: az alkalmazások és a zenék között – ezt a **ContentType** tulajdonságnál tudjuk megadni.

Kezdjük a legegyszerűbb taszkkal, a **MarketplaceHubTask**kal (12-19 ábra)! Ezzel egyszerűen megnyithatjuk a Marketplace alkalmazást (Marketplace Hubot), a **Content Type** értékével pedig meghatározhatjuk, hogy melyik típust szeretnénk megjeleníteni:

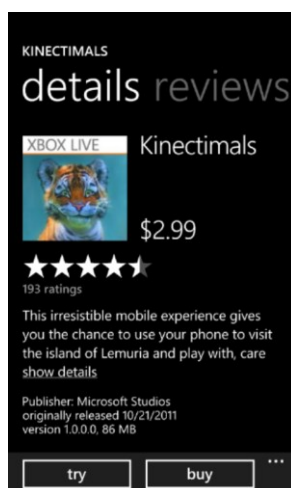
```
MarketplaceHubTask hubTask = new MarketplaceHubTask();
hubTask.ContentType = MarketplaceContentType.Applications;
hubTask.Show();
```



**12-19 ábra: Marketplace Hub**

Ha viszont egy meghatározott alkalmazás Marketplace oldalát szeretnénk betölteni, azt a **MarkerplaceDetailTask**kal tehetjük meg (12-20 ábra). Itt meg kell adnunk a hivatkozni kívánt alkalmazás egyedi azonosítóját, és amikor betöltődik, akkor a kiválasztott program részletes nézete töltődik be, ahol láthatjuk az alkalmazás logóját, leírását, képernyőképeit, és közvetlenül erről az oldalról le is töltheti a felhasználó az alkalmazást. Ezzel a launcherrel például megtehetjük azt, hogy az ingyenes próba változatban folyamatosan reklámozzuk az általunk írt alkalmazásokat.

```
MarketplaceDetailTask detailTask = new MarketplaceDetailTask();
detailTask.ContentType = MarketplaceContentType.Applications;
detailTask.ContentIdentifier = "078656ae-9f35-45bb-84b8-2f9270bbdd67";
detailTask.Show();
```



12-20 ábra: Marketplace Details Task

Nagyon fontos, hogy minél több felhasználó értékelje az alkalmazásunkat (előnye: előrébb kerül a Top listában)! Ezt elősegíthetjük a **MarketplaceReviewTask** használatával. Ennek a taszknak nincs speciális paramétere, nem tehetjük meg például, hogy előre meghatározott értékelést adjunk meg, egyszerűen csak megnyithatjuk az adott alkalmazás Review oldalát. Ezt a funkciót nem tudjuk kipróbálni az emulátorban, csak akkor tudjuk használni, ha az alkalmazásunk publikálva van, és fizikai készülékről indítjuk el az alábbi kódot:

```
MarketplaceReviewTask reviewTask = new MarketplaceReviewTask();
reviewTask.Show();
```

Az utolsó Marketplace-hez kötődő taszk a **MarketplaceSearchTask**, ami lehetőséget biztosít arra, hogy megnyissuk a Marketplace kereső ablakát, és abban alkalmazásokat, zenéket vagy podcastokat keressünk. Akár a saját magunk által készített alkalmazásokra is rákereshetünk.

```
MarketplaceSearchTask searchTask = new MarketplaceSearchTask();
searchTask.ContentType = MarketplaceContentType.Applications;
searchTask.SearchTerms = "Auto";
searchTask.Show();
```

## Trial Mode

Amikor alkalmazásunkat a piactérre publikáljuk, meghatározhatjuk azt, hogy annak van-e próba (*trial*) üzemmódja. Ha van, és ezt bekapcsoljuk, akkor a feltöltött alkalmazásunkat szabadon letölthetik a felhasználók anélkül, hogy megvennék azt. Annyiban különbözik ez a verzió a teljesítől, hogy az alkalmazáslicenc **IsTrial** értéke **true** lesz, és a mi dolgunk, hogy milyen módon korlátozzuk az alkalmazás funkcióit, vagy figyelmeztetjük a felhasználót a vásárlásra. Például megtehetjük azt, hogy figyeljük, az **IsTrial** értéket, és ha az igaz, akkor maximum háromszor indíthatja el a felhasználó az alkalmazást, utána leállítjuk; vagy időhöz köthetjük. Reklámokat és figyelemfelkeltő üzeneteket is elhelyezhetünk az alkalmazásban, jelezve a felhasználónak, hogy célszerű lenne megvásárolnia azt. A korlátozások kialakítása és kreatív megoldása a mi feladatunk, a Marketplace csak annyiban nyújt segítséget, hogy az **IsTrial** értéket megfelelően beállítja. Abban az esetben, ha megveszi a felhasználó az alkalmazást, az **IsTrial** értéke **false** lesz.

```
var license = new Microsoft.Phone.Marketplace.LicenseInformation();
var isInTrialMode = license.IsTrial();
```

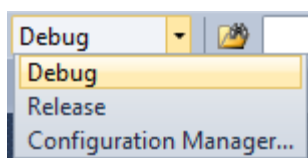
A trial mód szimulálása fejlesztési időben viszonylag problémás, ugyanis az **IsTrial** metódus eredetileg **false** értékkel tér vissza, ezért fejlesztési időben ezt szimulálnunk kell. Ennek kezelésére a DEBUG



szimbólumot tudjuk felhasználni: ha Release módban fordítjuk le az alkalmazást, akkor ez a szimbólum eltűnik:

```
public class ApplicationLicense
{
    public bool IsInTrialMode
    {
        get
        {
            #if !DEBUG
                var license = new Microsoft.Phone.Marketplace.LicenseInformation();
                return license.IsTrial();
            #else
                return true;
            #endif
        }
    }
}
```

Itt megvizsgáljuk, hogy az alkalmazás nem DEBUG módban indul-e. Ha nem, akkor az **IsTrial** metódustól kérdezzük le, hogy milyen módban fut az alkalmazás. Ebben az esetben az **false** értékkel tér vissza. Ha DEBUG módban indítjuk az alkalmazást, akkor mindig **true** értéket fogja visszaadni az **IsInTrialMode** tulajdonság.



**12-21 ábra: Release és Debug mód közötti váltás**

A feltételeket persze szabadon módosíthatjuk, a nekünk tetsző módon. Ahhoz, hogy módosítsuk a fordítási módot, a Visual Studio Toolbar-ján kell beállítanunk, hogy milyen módon szeretnénk fordítani (12-21 ábra).

## Összefoglalás

Ebben a fejezetben megismerkedtünk a Marketplace-szel, a Windows Phone egyik központi szereplőjével. Megnéztük azt, hogyan tudjuk a készülékünk fejlesztői zárolását feloldani, hogyan tudunk publikálni a Marketplace-re, és milyen feltételeket kell alkalmazásunknak teljesítenie ehhez. Azt is megvizsgáltuk, hogy milyen Marketplace taszkok állnak a rendelkezésünkre, és hogy hogyan kell próbamódú alkalmazást készíteni.



# 13. Teljesítmény

Egy asztali vagy akár laptop számítógéphez hasonlítva a telefon igencsak gyenge teljesítménnyel bír. A telefon akkumulátora egy asztali gépet csak pár percig tudna ellátni árammal. A CPU egymagos, és hiába fut 1GHz-en, számítási teljesítményben kb. egy 600 MHz-en futó Pentium III-al állítható párba (1999-ből). Egy kétéves Intel Core 2 processzor kb. 30-szor múlja ezt felül. Ami a GPU-t illeti, a Snapdragon processzor körülbelül azt a teljesítményt nyújtja, mint egy, szintén az elmúlt évezred végéről származó grafikus kártya.

De hát mihez kezdtünk ezzel az elképesztő számítási erővel? Nos, leginkább a felhasználói élmény javítására fordítottuk! Ahogy a számítógépek egyre gyorsabbak lettek, egyre jobban tudnak minket szolgálni – rövidebb várakozási idővel, több segítséggel a munkához, gazdagabb grafikával, nagyobb felbontással, sőt, már üzleti alkalmazásoknál is megjelenő animációkkal. A felhasználók mára elvárnak egy bizonyos szintű felhasználói élményt – ami 2000-ben még jó volt, 2010-re már nem elfogadható.

Az érintőképernyők megjelenésével az elvárások tovább nőttek. Az érintőképernyő közvetlen modellel váltja fel az egér közvetett interakciós modelljét – most már a saját ujjunkkal érintjük meg és mozgatjuk a képernyőn megjelenő dolgokat! Ez nagyszerű, de emiatt a „dolgoknak” a való világbeli tárgyakhoz hasonlóan kell reagálniuk. És a való világban a fizikai tárgyak nem 5-10 fps-el (*frame per second* – képkocka per másodperc) mozognak, és ha odébb lökjük őket, nem várnak még pár tized másodpercet mielőtt elkezdenének mozogni. Az asztali gépen valószínűleg észre se vennénk 300ms késleltetést a kattintás és a szoftver reakciója között, de egy érintőképernyőn ez már igen zavaró lehet.

A fejlesztői élmény is rengeteget változott – főleg a Microsoft világban. Immár a .NET keretrendszer negyedik generációját használjuk, és a mobil platformoktól is az asztali vagy webfejlesztéskor megszokott produktivitást (termelékenységet) várjuk el. A teljesítmény növelése lehetővé teszi, hogy fejlettebb, magasabb szintű eszközöket használjunk a fejlesztés során – legtöbbször már nem törődünk bitekkel, bájtokkal és memóriakezeléssel – az adatbázisokat és sokszor még a vezérlési folyamatot is grafikus eszközökkel hozzuk létre. A kliens-szerver kommunikáció immár a keretrendszer része, nem kell minden egyes projektnél újra feltalálnunk a kereket. Mi, a fejlesztők ugyanannyira nem kívánunk tíz évet visszalépni az időben, mint ahogy a felhasználók sem.

Kisebbségi csoda, hogy a zsebünkben hordozhatjuk azt, ami tíz éve csak az asztalon fért el. De ha belegondolunk abba, hogy a mai felhasználók megnövekedett igényeit kell fejlesztőként kielégítenünk, a telefon ereje igencsak kevésnek tűnik. Ez a mobil fejlesztés egyik legnagyobb kihívása. Lássuk, hogy a Windows Phone 7 platform Silverlight implementációja hogyan segít szembeszállni ezzel a kihívással!

## A tipikus teljesítmény-problémákról

Míg az átlagember sokszor csak annyit tud mondani, hogy „ööö... lassú”, a fejlesztőknek ennél pontosabban kell ismerni a tüneteket ahhoz, hogy a problémákat kezelni tudja. Nézzük tehát, milyen jelenségek tartoznak a teljesítmény-gondok témakörébe:

- **Akadozó animációk:** Mindenkinek más az ingerküszöbe, hogy mikortól zavarja egy animáció darabossága, akadozása. Abban viszont biztosak lehetünk, hogy a Windows Phone felhasználók elvárásai igen magasak – az egyik legfeltűnőbb és legtöbbet emlegetett pozitívum a platformmal kapcsolatban, hogy milyen finom (*silky smooth*) animációkkal kényeztet el minket.
- **Lassú betöltődés:** Az alkalmazás indítása után senki nem szereti, ha sokat kell várni addig, amíg az ténylegesen használható nem lesz. Sőt, ha a betöltődés 5 másodpercnél tovább tart és nincs ún. *splash screen* az alkalmazásban, az OS meg is öli azt. (Splash screen-nel 20 másodpercünk van az első képernyő megjelenéséig).

- **Hosszú válaszidő (reszponzivitás):** Ha megnyomtuk egy gombot a programban, mennyi idő alatt jön meg a válasz? Ha nincs semmi visszajelzés, még az is lehet, hogy újra meg újra megnyomjuk a gombot, azt gondolva, hogy nem érzékelte a telefon az érintést!
- **Görgetés közben üres területek jelennek meg (blanking):** Egy hosszú listát gyorsan végigpörgetve üres területeket látunk – a kirajzolás nem tud lépést tartani a görgetéssel.
- **Elfogy a memória:** Sokszor a felhasználó csak az alkalmazás lefagyását vagy hirtelen kilépését tapasztalja. Fejlesztőként legkésőbb a Marketplace ellenőrzés során találkozunk ezzel a jelenséggel, mert a Microsoft nem egyszerűséggel visszadobja a túl sok memóriát használó alkalmazásokat...

## A CPU és a GPU feladata

A CPU általános feladatokra alkalmas processzor – képes matematikai műveletek elvégzésére, utasítások és azokból szőtt algoritmusok végrehajtására. Az általunk írt program a CPU-n fut. A CPU (processzor) egy remek kis jószág, de van pár dolog, amire nem ideális. Ilyenek például a grafikai műveletek, vagy a videó dekódolás.

Ezért a telefonokban egy GPU (grafikus processzor) is található. A GPU hasonlóan működik, mint a nagytestvérei az asztali gépeken – háromszögeket képes a térben elhelyezni és megjeleníteni, több millió darabot másodpercenként. Szerencsére Windows Phone fejlesztőként nekünk nem kell a GPU-val ilyen alacsony szinten foglalkozni – sőt, a legtöbbször észre sem vesszük, hogy a Silverlight motor a GPU-t dolgoztatja. Azonban a teljesítmény-problémák orvoslásához már jó, ha tudunk pár dolgot a GPU-ról. A GPU segít levenni a CPU válláról a terhet az alábbi műveletek esetén:

- **Kompozíció** – két vagy több, egymás fedő vagy félig átlátszó textúra (bittérkép) egymásra helyezése. A mozgatás is ennek segítségével történik (a pozíció változtatása után a következő képkockában (frame-ben) újra egymásra helyeződnek a textúrák).
- **Forgatás** – egy bittérkép elforgatása
- **Átméretezés** – nagyítás vagy kicsinyítés. A GPU képes átméretezéskor az éleket elmosni, így elkerülhető a pixelesedés.
- **Perspektivikus torzítás** – a textúra elforgatása 3D-ben. Ez nem igazi 3D, inkább 2.5D, mivel nem testekről csak lapos textúrákról beszélünk - de megfelelő használatával mégiscsak megteremthető a 3D illúziója, mint ahogy a SurfCube 3D Browserben is történt (lásd 13-14. ábra). (Komolyabb 3D alkalmazást inkább XNA-val készítsünk).
- **Négyszögletes vágás** – a textúrákból a GPU is képes vágást készíteni (clip), amennyiben a vágó geometria téglalap alakú.
- **Videó dekódolás** – a Windows Phone-on a videó dekódolást és átméretezést dedikált hardver végzi – ez jó hír, mert a CPU önmagában nem lenne elég erős nagyfelbontású videók lejátszására. Így viszont marad ideje egyéb feladatokra (pl. *seek bar* megjelenítésére, feliratok elhelyezésére, stb).

Figyelmes olvasóknak feltűnhetett, hogy a GPU textúrákat, bittérképeket manipulál. De hogy lesz egy vezérlőből (pl. egy gomb) textúra? Ezt a folyamatot *renderelésnek* vagy *raszterizálásnak* hívják, és a CPU végzi. A CPU feladata még a XAML értelmezése, a vezérlők létrehozása, a layout (elrendezés) számítása, és természetesen a programunk futtatása is.

## Csillagok, csillagok...

Illusztrációként készítsünk egy egyszerű programot – ezen fogjuk megvizsgálni a fenti teljesítmény-problémákat, és a javításuk módját. A program az első lépésben egy csillagmezőt jelenít meg. Visual Studio-ban hozzunk létre egy új C# nyelvű Windows Phone alkalmazást **CpuAndGpu** néven, és a **MainPage.xaml.cs** -be írjuk be az alábbiakat:

```

using System;
using System.Diagnostics;
using System.Windows;
using System.Windows.Media;
using System.Windows.Shapes;
using Microsoft.Phone.Controls;

namespace CpuAndGpu
{
    public partial class MainPage : PhoneApplicationPage
    {
        private Stopwatch stopwatch;

        // Constructor
        public MainPage()
        {
            stopwatch = new Stopwatch();
            stopwatch.Start();
            InitializeComponent();
            CreateStarField(100);

            Loaded += OnLoaded;
        }

        private void OnLoaded(object sender, RoutedEventArgs routedEventArgs)
        {
            MessageBox.Show("Startup time: " +
                stopwatch.ElapsedMilliseconds + " ms");
        }

        public void CreateStarField(int numOfStars)
        {
            Random x = new Random();
            for (int i = 0; i < numOfStars; i++)
            {
                Ellipse ellipse = new Ellipse();
                double size = x.NextDouble()*3;
                ellipse.Width = size;
                ellipse.Height = ellipse.Width;
                ellipse.Fill = new SolidColorBrush(Colors.White);
                ellipse.Margin = new Thickness(x.Next(456), x.Next(607), 0, 0);
                ellipse.HorizontalAlignment = HorizontalAlignment.Left;
                ellipse.VerticalAlignment = VerticalAlignment.Top;

                ContentPanel.Children.Add(ellipse);
            }
        }
    }
}

```

A Solution Explorer-ben töröljük ki a **SplashScreen.jpg** fájlt.

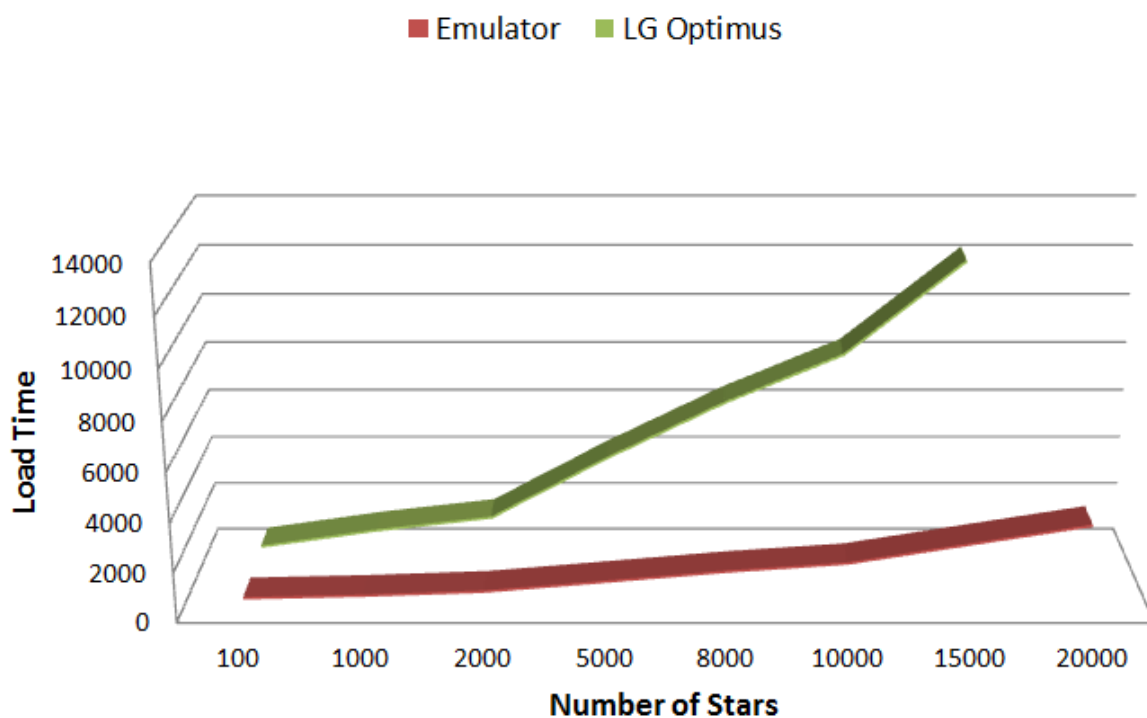
A program működése igen egyszerű. A **MainPage** osztály létrehozása után elindítunk egy stoppert, ami az indulás óta eltelt ezredmásodperceket méri. A **Loaded** eseménykezelőjében (ami a **MainPage** megjelenése után kerül meghívásra) egy **MessageBox**-ban kiírjuk az induláshoz szükséges időt. A stopper elindítása után az **InitializeComponent** függvény a XAML-ben leírtakat értelmezi, majd az általunk írt **CreateStarField** metódus a paraméterként kapott **numOfStars** számú véletlen elhelyezkedésű és méretű csillagot jelenít meg a képernyőn.

Ha elindítjuk a programot az emulátorban, a 13-1 ábrán látható eredményt kapjuk.



**13-1 ábra: 100 csillag, 388 ms**

A programnak azonban a telefonon kell futnia, indítsuk el ott is! Egy első generációs LG Optimus 7 telefonon 558 ms alatt indul el az alkalmazás. De mi történik, ha nem csak 100, hanem akár több ezer csillagot szeretnénk megjeleníteni? A mérések eredménye jól látszik a 13-2. ábra grafikonján:



**13-2 ábra: csillagok kirajzolási ideje emulátoron és telefonon**

A 20000 csillagos értéket telefonon már meg sem tudtuk mérni, mivel az alkalmazás nem indult el 20 másodpercen belül, és így az operációs rendszer „megölte” azt.

Tapasztalt fejlesztők számára feltűnhet, hogy a csillagok kirajzolásakor van lehetőség még a gyorsításra. Az ellipszis magasságát ugyanarra az értékre állítjuk, mint a szélességét (ami egy véletlen érték) - de ezt a `ellipse.Height = ellipse.Width`; sorral érjük el. Ha ezt a sort lecseréljük arra, hogy `ellipse.Height = size`, akkor a telefonon 5000 csillag már 4202 ms alatt megjelenik (szemben az eredetileg mért 4300-al). Ennek az oka, hogy nem minden egyes csillagnál egy újabb Dependency Property get-et végrehajtani. Talán nem tűnik soknak ez a csillagonkénti 0,2 ms, de egy sok ezerszer végrehajtott ciklus közepén még az ilyen apróságok is számíthatnak.

A fenti kísérletből két fontos tanulságot vonhatunk le.

- **Az emulátor performancia-karakterisztikája teljesen más, mint a telefoné.** Ne vonjunk le következtetéseket az alkalmazás teljesítményére vonatkozóan abból, hogy hogyan viselkedik az emulátorban! Bár az emulátor rendkívül pontos, és ugyanazt az OS-t futtatja, mint maga a telefon, ezt mégiscsak egy asztali gép processzorán és videokártyáján teszi. **Minden esetben ellenőrizzük az alkalmazás futását egy fizikai eszközön is, és a teljesítményt arra hangoljuk be!**
- **A hagyományos .NET optimalizáló technológiák és a józan ész** a Windows Phone fejlesztés során is segítenek.

## Lassú betöltődés kezelése

A fenti csillagos alkalmazásnál egyértelműen probléma, hogy lassan töltődik be a telefonon. Nézzük, hogy lehet ezen javítani!

### Az elrendezés (layout) költségeinek csökkentése

Figyelmesebben megnézve a `MainPage.xaml` - t, láthatjuk, hogy a csillagokat tartalmazó elem egy `Grid`. A `Grid` igen rugalmas, a rendelkezésre álló terület változásait jól kezelő, és sok mindenre jól használható konténer – ugyanakkor ez a komplexitás nyilván nincs ingyen. Cseréljük hát ki a `Grid`-et egy jóval egyszerűbb társára, egy `Canvas`-ra:

```
<!--<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0"></Grid>-->
<Canvas x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0"></Canvas>
```

Módosítsuk a kódot is, hogy margók helyett a `Canvas.Top` és `Canvas.Left` tulajdonság állításával kerüljenek a csillagok elhelyezésre:

```
public void CreateStarField(int numStars)
{
    Random x = new Random(numStars);
    for (int i = 0; i < numStars; i++)
    {
        Ellipse ellipse = new Ellipse();
        double size = x.NextDouble()*3;
        ellipse.Width = size;
        ellipse.Height = size;
        ellipse.Fill = new SolidColorBrush(Colors.White);
        ellipse.SetValue(Canvas.LeftProperty, (double)x.Next(456));
        ellipse.SetValue(Canvas.TopProperty, (double)x.Next(607));
        // ellipse.Margin = new Thickness(x.Next(456), x.Next(607), 0, 0);
        // ellipse.HorizontalAlignment = HorizontalAlignment.Left;
        // ellipse.VerticalAlignment = VerticalAlignment.Top;

        ContentPanel.Children.Add(ellipse);
    }
}
```

5000 csillaggal futtatva az alkalmazást, a **Grid**-es 4100 ms helyett 3093 ms-ot mértünk!

Általában tehát elmondhatjuk, hogy csak akkor használjunk bonyolult layout rendszereket, ha arra valóban szükség van. Minden átméretezés, a gyerekek számának megváltozása a vizuális fa teljes vagy részleges újraméretezését vonja maga után - és minél kevesebb dolgot kell ehhez a CPU-nak végeznie, annál gyorsabb lesz az alkalmazásunk.

## Splash Screen

A *Splash Screen* a program indítását követően megjelenő egész képernyős ábra. A *Splash Screen* nem gyorsít a beöltődésen – ugyanakkor, ha az indulást nem tudjuk 1-2 másodperc alá szorítani, a felhasználónak sokkal jobb érzés egy ízléses rajzot bámulnia, mint az üres kijelzőt. Figyelem, a WP7 alkalmazásablonban található **SplashScreen.jpg** nem tartozik az ízléses kategóriába! A Marketplace előírások meg is követelik a *Splash Screen* használatát, ha az alkalmazás indulása 5 másodpercnél több időt vesz igénybe. A *Splash Screen* addig látszódik, amíg az első képernyő meg nem jelenik.

A *Splash Screen* másik előnye, hogy a felhasználó – amíg nem vesz a kezébe stopperórát – gyorsabbnak is fogja érezni a betöltődést, mivel valami történik a kijelzőn. Ezt az érzést szubjektív teljesítménynek hívjuk, és a felhasználói élményt jelentősen lehet javítani hozzáértő módon alkalmazott trükkökkel (lásd később).

*Splash Screen* igen könnyű az alkalmazásba építeni – helyezzünk el egy **SplashScreenImage.jpg** fájlt a WP7 projekt gyökérkönyvtárában és állítsuk a **Build Action** tulajdonságot a **Content** értékre!

## Inicializációs teendők elhalasztása

A fenti példaprogramban a betöltést jelentősen lassítja a csillagok kirajzolása. Mi lenne, ha a betöltődés után rajzolnánk csak ki a csillagokat? Mozgassuk át a **CreateStarField** metódust a konstruktorból az **OnLoaded** eseménykezelőbe:

```
private void OnLoaded(object sender, RoutedEventArgs routedEventArgs)
{
    CreateStarField(5000);
    MessageBox.Show("Startup time: " + stopwatch.ElapsedMilliseconds + " ms");
}
```

Ha így elindítjuk az alkalmazást a telefonon, két dolgot vehetünk észre. Az egyik, hogy a stopperes üzenet megjelenésekor még nem látszanak a csillagok, viszont az üzenet 3093 helyett már csak 2876 ms-ot mutat. Ennek az oka az, hogy a tényleges rajzolás ekkor még nem történt meg, a csillagok csak a **MessageBox** lezárása után jelennek meg.

A másik jelenség az, hogy az alkalmazás szinte azonnal elindul, a fejléc-szöveg villámgyorsan megjelenik. Ez azonban ne tévesszen meg senkit – az alkalmazás a csillagok rajzolása során nem reagálna az inputra, mivel a UI szál a csillagokkal van elfoglalva. Ezt könnyen ellenőrizhetjük, ha a **MainPage.xaml.cs**-ben elhelyezünk egy gombot:

```
<Canvas x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <Button Canvas.ZIndex="1">I am not doing anything...</Button>
</Canvas>
```

A gomb megnyomására a gomb inverzbe vált. Azonban ez a raszterizálás az UI szálon történik, ami el van foglalva a csillagok rajzolásával, így csak a csillagok kirajzolása után fog a gomb az érintésre reagálni.

## Háttérszálak alkalmazása

Tegyük hát át a csillagokat egy háttérszálra! Ehhez az **OnLoaded** eseménykezelőbe kell belenyúlni, illetve egy új, **BGCreateStars** metódust létrehozni:



```

private void OnLoaded(object sender, RoutedEventArgs routedEventArgs)
{
    Thread t = new Thread(BGCreateStars);
    t.Name = "Stars";
    t.Start();

    //          CreateStarField(5000);
    //          MessageBox.Show("Startup time: " + stopwatch.ElapsedMilliseconds + " ms");
}

private void BGCreateStars()
{
    for (int i = 0; i < 500; i++)
    {
        UIDispatcher.BeginInvoke(() => CreateStarField(10));
        Thread.Sleep(30);
    }
}

```

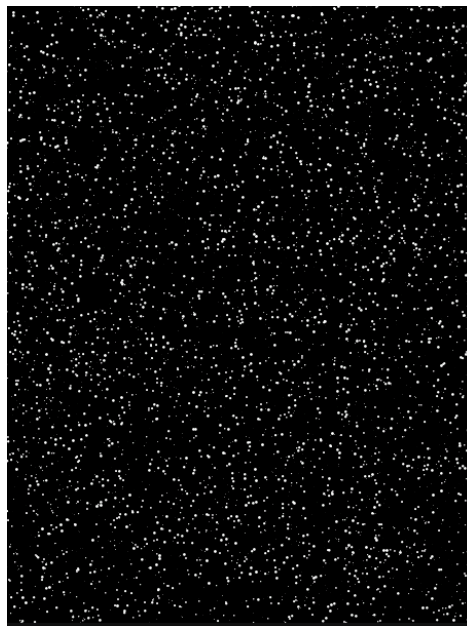
A **BGCreateStars** 500-szor lefuttatja a **CreateStarField** metódust 10 csillaggal, és minden futás után pihen 30 ms-ot. A **CreateStarField** azonban az UI szálon kell, hogy fusson, mert az UI-t manipulálja. Ezért az **UIDispatcher.BeginInvoke** híváson keresztül kell elérni a **CreateStarField** metódust.

A fentiek alapján módosított alkalmazásból kivettük a **MessageBox** hívását is, mert az még a csillagok kirajzolása előtt megjelenne, így nincs értelme.

A programot futtatva már igen kellemes felhasználói élményben van részünk: az indulási idő minimális, a program azonnal reagál a gombnyomásra. A csillagok pedig szépen lassan jelennek meg, szinte olyan, mintha az alkalmazás egyesével rajzolgatná őket.

### ***Előre elvégzett munka***

Ez mind szép, de mi van, ha nekünk a teljes csillagmezőre van szükségünk, a betöltődés után azonnal? Ebben az esetben is van megoldás – egyszerűen készítsünk egy screenshotot a kívánt csillagmezőről, és tegyük be a **ContentPanel** háttérképének (13-3. ábra). Ehhez a projektben el kell helyezni a fájlt (pl. **starsBG.png** néven), és a tulajdonságait beállítani: a **Build Action** legyen **Content**, a **Copy to Output Directory** pedig **Copy if newer**. A **MainPage.xaml.cs** tartalmát teljes egészében visszaállíthatjuk az eredetire (a konstruktorban egy **InitializeComponent** hívás marad csak).



**13-3 ábra: A starsBG.png**

A **MainPage.xaml** - ben pedig a **ContentPanel**t a következőre cseréljük:

```
<Canvas x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <Canvas.Background>
        <ImageBrush Stretch="None" ImageSource="/starsBG.png"/>
    </Canvas.Background>
    <Button Canvas.ZIndex="1">I am not doing anything...</Button>
</Canvas>
```

Tádááám! Az alkalmazás fél másodpercen belül elindul még 5000 csillaggal is! Ha mégis szükségünk lenne a véletlenszerű csillagmezőre, egyszerűen készítsünk 10 különböző képet, és azok egyikét állítsuk be kódból háttérnek.

### Kis összefoglalás

A fenti alapelvek persze „valódi” alkalmazásoknál is használhatók. Az iniciális teendők elhalasztására jó példa a SurfCube 3D Browser alkalmazás 4.0 változata. Mivel az app indulása túllépte az 5-6 másodpercet, egyre több felhasználó panaszkodott. Volt, aki egyenesen azt gondolta, hogy a Splash Screen az ő bosszantásukra van ott, és csak azért tettük oda, mert tetszett nekünk a logó. Ezt mind el is fogadta volna, de az már felháborította, hogy az alkalmazás megvásárlása után is túl sokáig volt látható a Splash Screen!

A SurfCube 4.0-ban ezért a kocka oldalait már eleve „Collapsed” állapotban töltjük be. Az első forgatáskor váltunk át innen „Visible”-re. Ez kicsit kényelmetlen talán (bár némi loading... felirattal és animációkkal ellensúlyoztuk a dolgot), de lehetővé tette, hogy az indulás 3.5 másodpercre rövidüljön.

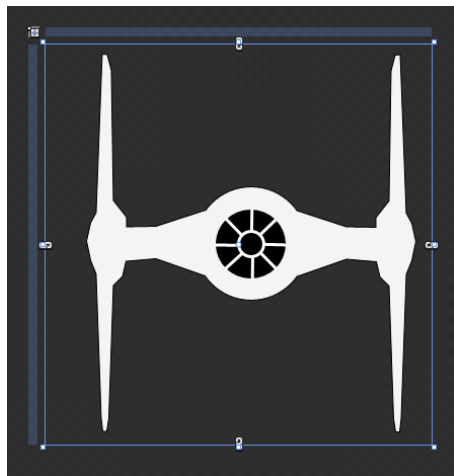
A másik dolog, ami sokat segíthet, ha a ritkán használt funkciókat külön assembly-be tesszük. Ekkor, amíg az assembly-re nem történik hivatkozás a kódban, azt nem tölti be az operációs rendszer.

Az előre elvégzett munkára pedig jó példa a „Névnap” alkalmazásom. Itt egy SQL CE adatbázis tartalmazza a névnapokat. Ez az adatbázis egy szöveges fájl alapján, az első futtatás során jön létre – a lapkák és az alkalmazás későbbi futása már az SQL CE-t használják. Még jobb megoldás ilyen esetben, ha a (csak olvasásra szánt) SQL adatbázist már a fejlesztés során a XAP fájlba helyezzük, így a legelső futás is gyors lehet.

### Akadozó animációk okai, megoldása

Windows Phone Silverlightben többféle módon lehet animációt létrehozni. Kezdjük a legritkábban használatos, de a teljesítmény szempontjából a legkritikusabb megoldással – az ún. procedurális animációval!

Az animációval azt akarjuk szimulálni, hogy mit lát egy részeg, sokszemű úrhajós, akivel szembe jön a csillagos űrben egy TIE figther. Ehhez először is elkészítettem egy **TIEFighter.xaml** user controlt. Ennek kódját felesleges lenne ide bemásolnom – elérhető a könyvhöz tartozó letölthető anyagok között (13-4 ábra).



13-4 ábra: a TIEFighter.xaml Expression Blend-ben

A procedurális animáció lényege, hogy minden képkockában (frame) újraszámoljuk az objektumok helyzetét. A teljes kód a fejezethez mellékelt kódban elérhető, itt csak a lényeget emelném ki.

Ahhoz, hogy a **ProceduralAnim** példa induljon el, a **Properties/WMAppManifest.xml** fájlban a **DefaultTask NavigationPage** értékét módosítani kell:

```
<DefaultTask Name = "_default" NavigationPage="ProceduralAnim.xaml"/>
```

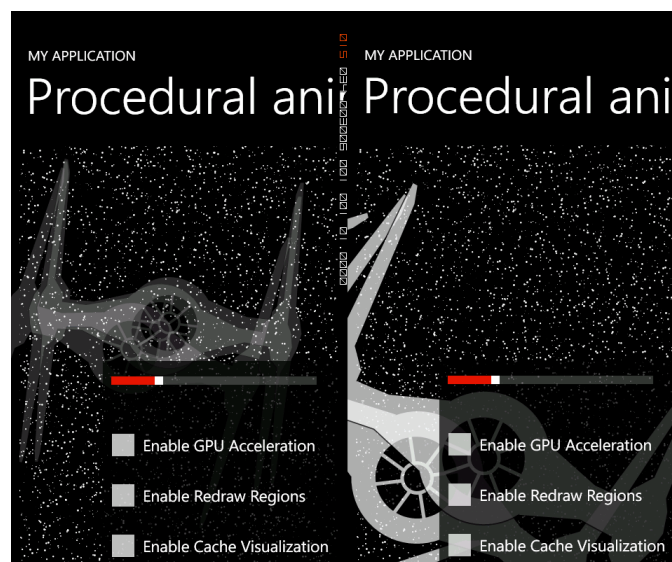
Maga az animáció a **ProceduralAnim.xaml.cs** fájlban, a **CompositionTargetOnRendering** eseménykezelőben történik. Ez a **CompositionTarget** osztály **Rendering** eseménye, amely minden alkalommal lefut, amikor az UI szál egy új képkocka rajzolására készül. Az eseménykezelő a következőképpen néz ki:

```
private int frameCount = 0;
private void CompositionTargetOnRendering(object sender, EventArgs eventArgs)
{
    if (frameCount >= 100)
        frameCount = 0;

    for (int i = 0; i < Ties.Count; i++)
    {
        var tieFighter = Ties[i];
        CompositeTransform tr = (CompositeTransform) tieFighter.RenderTransform;
        tieFighter.Opacity = frameCount/100.0;
        double scale = 1 + frameCount/100.0;
        tr.ScaleX = scale;
        tr.ScaleY = scale;
        tr.Rotation = frameCount / (i+1.0);
    }

    frameCount++;
}
```

A **Ties** egy **TIEFighter** objektumokat tartalmazó lista. A fenti függvény nem csinál mást, mint végigmegy az összes TIE Fighter-en, és mindegyiknek átállítja az átlátszóságát, méretét és elforgatását. A 13-5 ábra mutatja, hogy három TIE Fighter esetén (vagyis ha a részeg űrhajósnak három szeme van nyitva) milyen animációt láthatunk.



**13-5. ábra: három TIE fighter animációja**

Az ábrák jobboldalán látható számok az ún. *frame rate counter*-ek – érdemes itt kitérni rájuk.

### Frame Rate Counters

Az `App.xaml.cs` fájlban a konstruktorban a következő sorokat találhatjuk:

```
// Show graphics profiling information while debugging.
if (System.Diagnostics.Debugger.IsAttached)
{
    // Display the current frame rate counters.
    Application.Current.Host.Settings.EnableFrameRateCounter = true;

    // Show the areas of the app that are being redrawn in each frame.
    //Application.Current.Host.Settings.EnableRedrawRegions = true;

    // Enable non-production analysis visualization mode,
    // which shows areas of a page that are handed off to GPU with a colored overlay.
    //Application.Current.Host.Settings.EnableCacheVisualization = true;

    // Disable the application idle detection by setting the UserIdleDetectionMode property of
the
    // application's PhoneApplicationService object to Disabled.
    // Caution:-
    Use this under debug mode only. Application that disables user idle detection will continue to
run
    // and consume battery power when the user is not using the phone.
    PhoneApplicationService.Current.UserIdleDetectionMode = IdleDetectionMode.Disabled;
}
```

Láthatjuk, hogy még két eszköz áll rendelkezésre a grafikai teljesítmény analízálásához - a redraw regions (újrarajzolt területek) és a cache visualization (cache vizualizáció). Ezekről később lesz szó.

A fenti sorok hatására, ha egy alkalmazást debug módban indítjuk (F5-el), a képernyő oldalán megjelenik néhány számláló (ezeket hívjuk *Frame Rate Counter*-eknek), amint az a 13-6 ábrán is látható.



13-6. ábra - Frame Rate Counters

Mit is jelentenek ezek a számok? Az egyes Frame Rate Counterek jelentését az alábbi táblázat foglalja össze, amely a számokat balról jobbra sorolja fel:

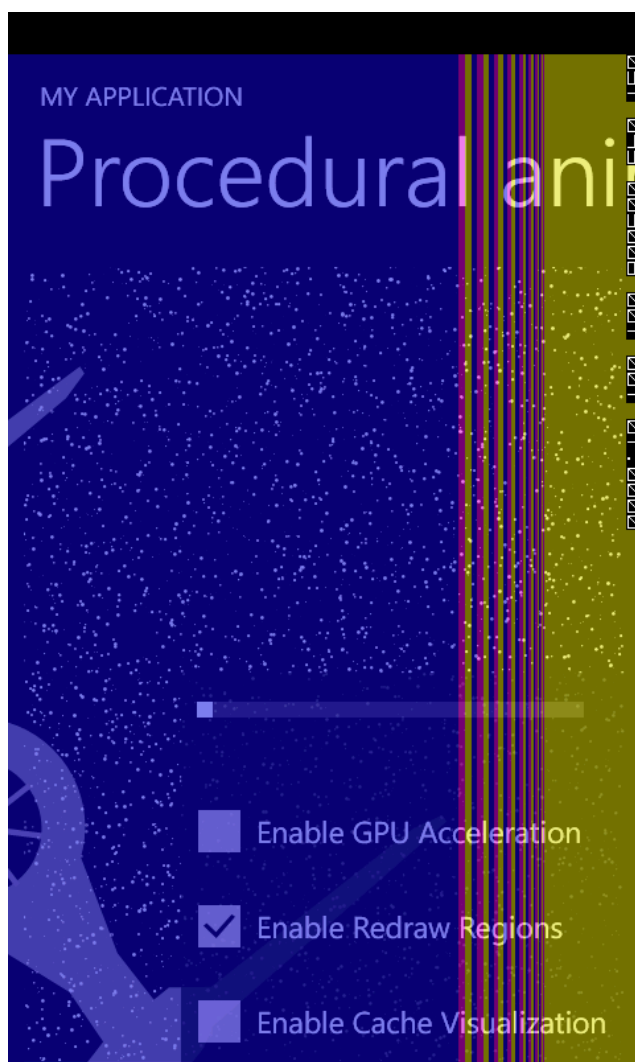
Counter	Magyarázat
<b>Composition szál frame rate</b>	Azt mutatja meg, hogy másodpercenként hányszor kerül a képernyő frissítésre. Ha ez a szám 60-nál (egyes eszközökön 50-nél) kisebb, az animációk veszítenek folyamatosságukból. Ha 30 alá csökken, a felhasználók is észreveszik a problémát, és a szám pirosra vált.
<b>UI szál frame rate</b>	Az UI szál az, ahol a raszterizáció (renderelés) történik, és ahol az általunk írt kód fut (hacsak nem indítunk új szálat). Ezért, az alacsony UI szál frame rate azt jelenti, hogy az alkalmazás nem

Counter	Magyarázat
	válaszol időben az inputra. Előfordulhat, hogy ekkor a Composition szálon és a GPU-n továbbra is folyamatosan futnak az animációk, de a felhasználói élményt igencsak rosszul érinti, ha az UI szálát leterheljük.
<b>Textúra-memória használata</b>	Azt mutatja, hogy a textúrák mennyi videó- és rendszer-memóriát használnak
<b>Explicit felületek (surface) száma</b>	Az explicit felületek számát mutatja, amellyel a GPU dolgozik. Ha egy <b>FrameworkElement</b> példányon a <b>CacheMode</b> tulajdonságot <b>BitmapCache</b> értékre állítjuk, vagy a <b>Composition</b> szálon futtatunk egy animációt, a számláló értéke megnő.
<b>Implicit felületek (surface) száma</b>	A GPU által használt implicit felületek számát mutatja (lásd később)
<b>Fill Rate</b>	<p>A GPU-k egyik legfontosabb teljesítmény-mutatója a Fill Rate. A GPU-k egy másodperc alatt csak bizonyos számú pixelt tudnak megrajzolni. A Windows Phone 7 esetén a kijelző 800x480 pixelt tartalmaz, és ezt egy első generációs hardver GPU-ja képkockánként kb. 2.5-szer tudja megtölteni.</p> <p>A Fill Rate 1-es értéke azt jelenti, hogy a GPU egy képernyőnyi pixelt rajzol fel frame-eként. Ha ez a szám 2.5 fölé emelkedik, a GPU szűk keresztmetszetté válik. Ha a számláló 3 fölötti értéket mutat, pirossá válik, mert az animációk elvesztik folyamatoságukat a frame rate csökkenésével. Figyeljünk oda arra, hogy ha az alkalmazásnak megadunk egy háttérszínt, ez önmagában 1-el növeli a Fill Rate-et, mert a GPU-nak először ezt a hátteret kell megrajzolnia. Ezért célszerű a hátteret átlátszónak hagyni, ha a rendszer témáját használjuk.</p>

A fentiek alapján látható, hogy három vadászgéppel az emulátor kb. 15 FPS-el tudja futtatni az UI szálát. (A telefonom 12-13 fps-t mutat). A problémát nem a magas Fill Rate okozza, mivel az fixen 1. A telefon még 1 TIE esetén is könnyen lemegy 20-30 FPS környékére. Mi lehet ennek az oka?

### ***Redraw Regions***

Ha bekapcsoljuk a Redraw Region opciót, hozzáértő szem számára rögtön nyilvánvalóvá válik a probléma. A 13-8 ábrán látható, hogy mit okoz a bekapcsolása.



13-8. ábra: Redraw Regions

A Silverlight a nagyobb sebesség érdekében spórol a rajzolással. Így csak azokat a területeket rajzolja újra, amelyek változnak két képkocka között. A Redraw Region ezeket az újrarajzolt területeket minden egyes frame esetén különböző színnel megfesti (ez némi CPU időbe telik, így ne végezzünk FPS méréseket, ezzel az opcióval).

Az ábrán jól látszik, hogy a képernyő jelentős részét újra kellett rajzolni, ahogy a TIE Fighter elhalad a csillagok fölött és a kezelőszervek alatt. Ráadásul maga a TIE Fighter is újrarajzolásra kerül, mivel folyamatosan változik a mérete és az elforgatása. Ezt a rasterizálást minden frame-ben a CPU végzi – így nem csoda, ha nem képes másodpercenként 20-25-nél többször új képkockát készíteni.

Korábban említettem, hogy a GPU sok olyan feladatot képes hatékonyan ellátni, amire itt is szükség van. Kompozíció, forgatás, átméretezés, stb... hát akkor:

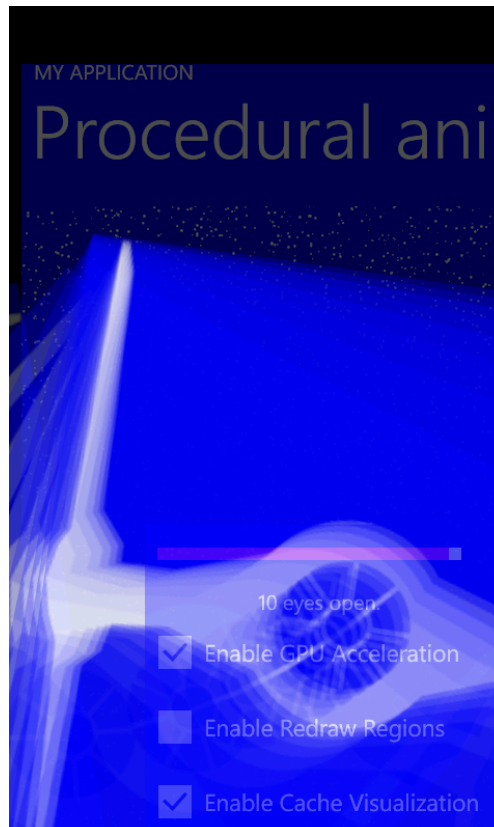
### ***Használjuk a GPU-t!***

Az Enable GPU Acceleration gomb bekapcsolásával a helyzet drámaian megváltozik. A Redraw Region abbahagyja az őrült villódzást. 3 vadászgép esetén a telefon még érezhető lassulás nélkül animál. Még tíz TIE Fighternél sem esik 20 FPS alá a sebesség. Ugyanakkor az emulátor egy erős videokártyával mindennemű lassulás nélkül elvisel tíz vadászgépet, de hát az a videokártya ugye egész más kategória, a Call Of Duty MW 3-ra van méretezve...

Mi is történik ilyenkor?

Ez persze azt is jelenti, hogy bizonyos animációk gyorsabban futnak, mint mások. A CPU animálja pl. a nem téglalap alakú vágásokat, a szín-animációkat, a layout-ot, gradienseket, stb. Ha ilyen animációkat használunk, tegyük azt inkább kis méretben – így a CPU-nak nem kell annyi pixelt átszámolnia, és így fenntartható a magas FPS érték.

## Cache vizualizáció



MY APPLICATION

# Procedural animation

10 eyes open

- ☒ Enable GPU Acceleration
- ☐ Enable Redraw Regions
- ☒ Enable Cache Visualization

331



A cache vizualizáció kékkel fest meg, és félig átlátszóvá tesz minden textúrát, amit a GPU megjelenít. Így nyomon követhető az, hogy mely felületeket tárolja a GPU. Érdekes újra végigvenni a Frame Rate Counter-eket:

- **UI és Composition thread FPS** – ezek jóval magasabbak a telefonon tapasztalhatónál, mivel a screenshot emulátoron készült.
- **Használt textúra memória** – 8 Mbyte, nem vészes
- **Textúrák száma** – a GPU jelenleg 12 textúrával dolgozik. Ebből 10 a TIE Fighter-eké (mindegyikre jut egy), egy a háttéré és egy a kezelőszerveké. Érdekes megjegyezni, hogy bár nem adtuk meg a háttérképnek és a kezelőszerveknek a **BitmapCache** értéket, azokat a Silverlight automatikusan a GPU-n tárolja, mivel előttük és mögöttük cache-eltek textúrák találhatók. Ezeket a felületeket *implicit surface*-nek nevezzük, és a 2-es érték mutatja a számlálók között.
- **Fill Rate** – A probléma forrása itt keresendő, a telefon szépen be is pirosította nekünk. A Fill Rate 11.6 értéke azt jelenti, hogy a GPU minden frame-ben 11.6 képernyőnyi pixelt rajzol ki. Ez igen magas szám – az első generációs telefonokon már 2.5 fölélt elkezd csökkenni a frame rate, és 3.5 környékén már bántóan elkezd akadózni az animáció. A magas fill rate-et természetesen a sok (és egyre növekvő méretű) vadászgép okozza, bár esetünkben ezen sokat javít az, hogy mire megnőnek, egy jelentős részük kimegy a képernyőről. A Fill Rate az új generációs telefonokban az erősebb GPU-nak hála már magasabb értéknél kezd lassítani – azonban még jó darabig figyelembe kell vennünk az első generációs telefonok sebességét, így nem engedhetjük meg magunknak az ellustulást.

A cache vizualizáció megmutatja a feleslegesen használt GPU textúrákat, így lehetővé válik azok kiküszöbölése. De akármennyi trükközést csinálunk is, előbb-utóbb elérjük a hardver teljesítőképességének a határát – 50 teljes méretű vadászgépet már egyik telefon sem fog élvezhető sebességgel mozgatni.

### Beépített animációk és a GPU

A fenti animáció persze jóval egyszerűbben reprodukálható Blend-ben, **Storyboard**-ok segítségével. Az eredmény a **StoryboardAnim.xaml** fájlban található – sőt néhány **Easing**-gel megbolondítva sokkal természetesebb eredményt kapunk. A futtatáshoz a már ismert módon írjuk át a **WMAppManifest.xml** fájlban a **DefaultTask**-ot:

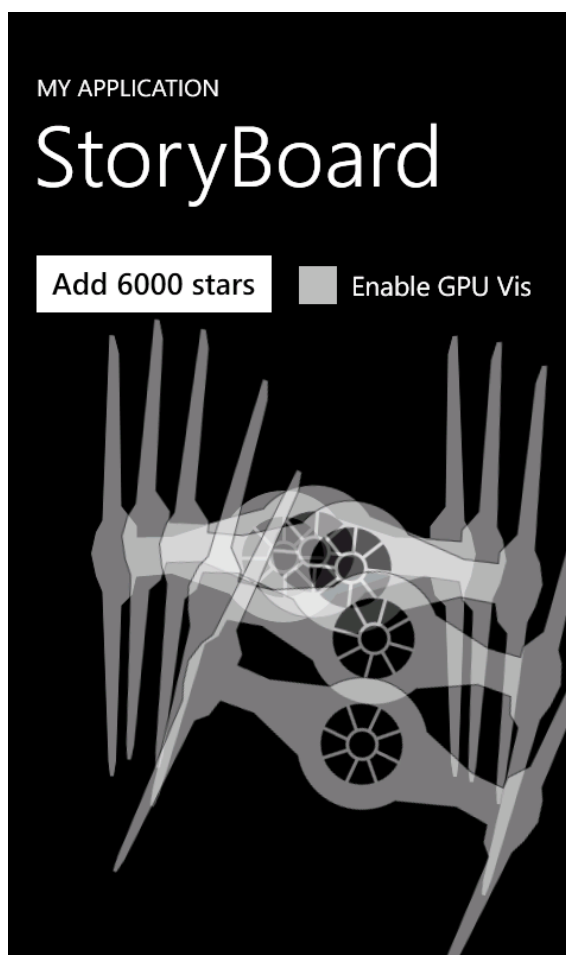
```
<DefaultTask Name = "_default" NavigationPage="StoryboardAnim.xaml"/>
```

Ha így elindítjuk az alkalmazást, feltűnhet, hogy az animáció tökéletesen folyamatos, még öt TIE Fighter-rel is. Könnyen ellenőrizhetjük ezt az állítást az Enable GPU Vis opció bekapcsolásával – jól látható, hogy minden vadászgép saját GPU textúrát kapott. Ez csak GPU használatával lehetséges – amit mi sehol nem kapcsolunk be. Valóban, a Silverlight futtatókörnyezete automatikusan bekapcsolja a GPU-t, amennyiben **Storyboard** animációt használunk procedurális helyett. A legtöbbször nem is kell ezen gondolkozni, ez ám a kényelem!

### UI, Compositor és Input szálak

A **StoryboardAnim.xaml** oldal nem tartalmaz csillagokat. Van azonban egy „Add 6000 stars” gomb a képernyőn. Ha ezt megnyomjuk, pár másodperccel később megjelenik a 6000 csillag. A kódot megvizsgálva azonban láthatjuk, hogy ennek a gombnak az eseménykezelőjében egyszerűen csak meghívjuk a **CreateStarField** metódust 6000-es számmal. Semmi háttérszál és egyéb varázslat. A fejezet elején leírtakból tudhatjuk, hogy ez a megközelítés hosszú másodpercekre teljesen lefoglalja az UI szálát. És valóban, az UI szál szóhoz sem jut: a gomb „benyomva” marad jóval az elengedése után is (13-10 ábra), mivel az UI szál a csillagok kirajzolásával van elfoglalva, nem hagyunk időt arra, hogy a nem benyomott állapotot kirajzolja.





**13-10 ábra - A UI szálát teljesen blokkolja a 6000 csillag kirajzolása, még a gomb is benyomva maradt.**

Ugyanakkor az animáció egy pillanatra sem lassult le a nagy csillaggyártás közben! Ennek az az oka, hogy az animációk egy külön szálon, a *Compositor szálon* futnak (és a GPU-n kerülnek megjelenítésre). Ez mindaddig így van, amíg olyan animációkat alkalmazunk, amikhez nincs szükség arra, hogy az elemeket újrarszterezze az UI szál, esetleg az elrendezés (layout) változása miatt a vizuális fát újra kelljen számolni. (A mozgathoz és forgathoz használt **RenderTransform** a layout után kerül alkalmazásra, így az elrendezést nem befolyásolja).

A Mangóban egy új, harmadik szál is bevezetésre került az input kezelésére. Ez és néhány további hangolás a WP7 első kiadásában tapasztalható legtöbb, rossz válaszdőjű szituációt megoldja. A legfeltűnőbb probléma a listák görgetésekor jelentkezett – a görgetett lista animációja folyamatos volt, de újabb érintésre (pl. ha a gördülés megállításához megérintettük a kijelzőt) néha csak másodpercnyi késleltetéssel reagált. Ennek oka, hogy a UI szál (amely Mango előtt az inputot is kezelte) el volt foglalta a folyamatosan érkező elemek raszterizálásával, és így csak későn észlelte az érintés eseményt. Mondanom sem kell, hogy ez igen kellemetlen felhasználói élményt okozott, amit a Mangóban (WP7.5) szerencsére már javítottak a külön szálon futó input-kezelés bevezetésével.

## Hosszú válaszdő kezelése

A hosszú válaszdőnek számos oka lehet. A jó hír, hogy a fentiekben bemutatott technikák a gyorsabb alkalmazás betöltésre és a folyamatosabb animációk elérésére a legtöbb esetben segíthetnek itt is.

Ha bonyolult UI-t kell megjelenítenünk, vegyük elő az alkalmazás-indítás optimalizálásánál leírtakat. Ha hosszú számítást kell végeznünk, használjunk háttérszálakat, akár a **BackgroundWorker** osztály segítségével is. Az elkerülhetetlen várakozást pedig könnyebbé tehetjük, ha a felhasználót tájékoztatjuk a folyamat állásáról (lásd a szubjektív teljesítmény résznél).

## Általános tippek

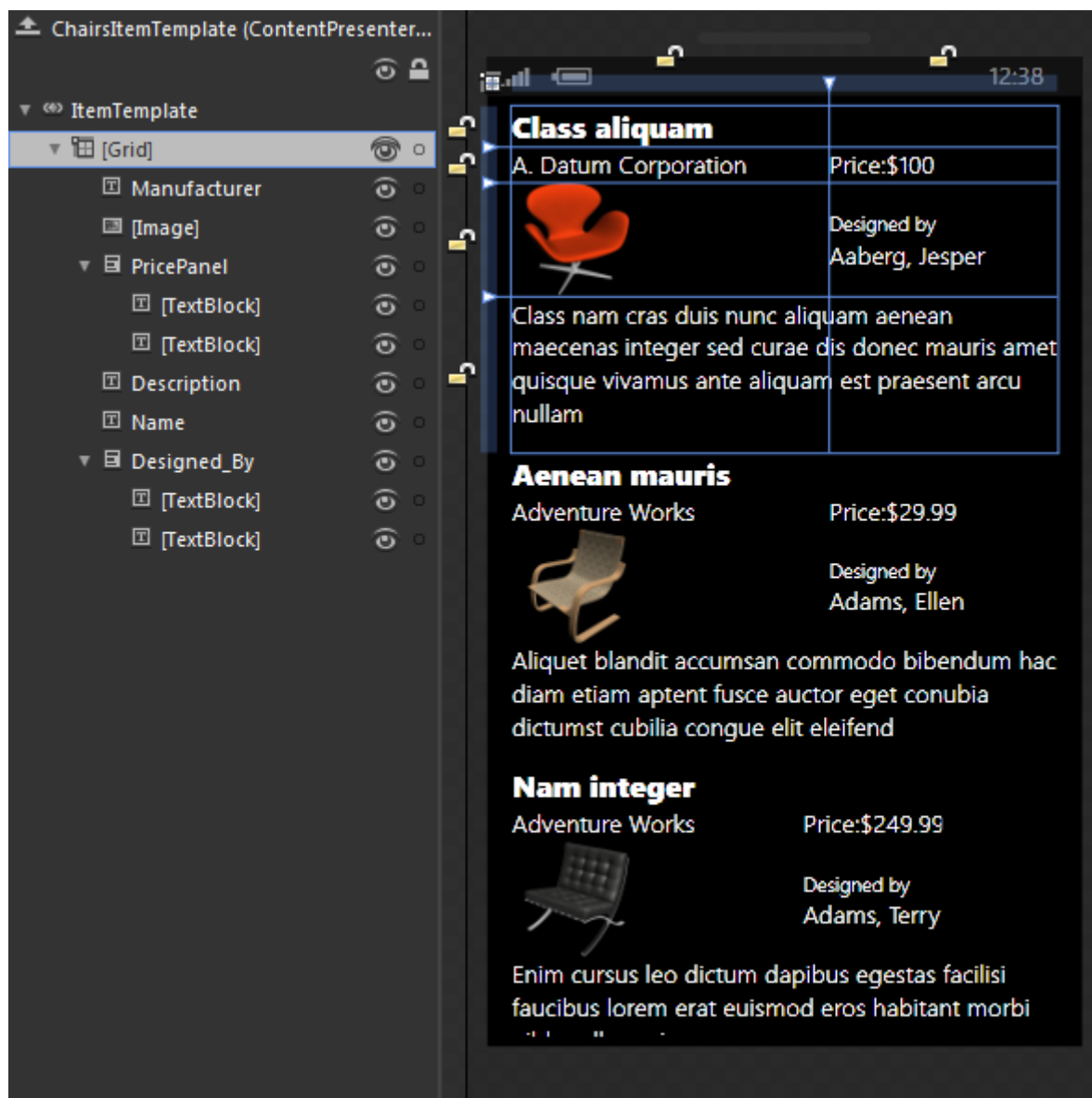
Számos további tipp található a neten, amivel a teljesítményt javítani lehet. A legfontosabb annak megértése, hogy semmi nincs ingyen – a Grid fejlett layout képességeinek ára, hogy sokat kell számolni az elemek elhelyezésén, a könnyű, deklaratív UI leírás ára a XAML interpretálásának költsége, az adatkötés rezsije. A teljesítmény-optimalizálás művészete abban áll, hogy meg kell találni azokat a pontokat, ahol érdemes a kód karbantarthatóságából áldozni, plusz munkát befektetni a szemmel látható eredmény érdekében. Nincs értelme például órákat ölni abba, hogy az alkalmazás indulása 200 ms-al gyorsabb legyen. Ugyanakkor egy hosszú listában az elemek kirajzolását akár 30ms-al is hasznos lehet megrövidíteni annak érdekében, hogy a görgetés közben kevesebb legyen az ún. „blanking” – vagyis, amikor üres elemek jelennek meg, mivel a layout és raszterizáció nem képes lépést tartani a gyors görgetéssel. Lássunk tehát néhány ilyen tippet a részletes példák mellőzésével:

- A **Pivot** vezérlő gyorsabb, mint a **Panorama**.
- Egyszerűsítsük a layoutot - használjunk **StackPanel**-t, **Canvas**-t **Grid** helyett, ahol lehet! Listákban a **DataTemplate**-eknél ez különösen hasznos lehet.
- Amit ma megtehetsz, azt nyugodtan megteheted holnap is (lásd a folyamatos csillag kirajzolás az első példákban a gyorsabb alkalmazás-indulás érdekében).
- Ha a képeket nem **Resource**-ként, hanem **Content**-ként helyezzük az alkalmazásba, gyorsabb lesz annak indulása.
- A **Compositor** szálát lehetőleg tartsuk 60 fps körül, a Fill Rate-et 2.5 alatt!
- Ha határozatlan **ProgressBar**-t (**IsIndeterminate=True**) alkalmazunk, kapcsoljuk ki ezt a beállítást, ha már nincs rá szükség – még akkor is, ha a vezérlő nem látszik a képernyőn! Még jobb, ha a Mango-ban megjelent **SystemTray.ProgressIndicator**-t (bár ez csak a képernyő tetején jelenik meg), esetleg a Silverlight for Windows Phone Toolkitből a **PerformanceProgressBar** vezérlőt használjuk.
- A **ScrollViewer**-ben (és listákban) az animációkhoz hasonlóan automatikusan GPU cache-elésre kerülnek az elemek.
- Adatbázis használatkor definiáljuk a **Version** oszlopot, implementáljuk az **INotifyPropertyChanged** interfészt az entitásokban, és használjunk előre fordított lekérdezéseket (compiled queries).
- A képek dekódolása alapértelmezésben az UI szálon történik. A **CreateOptions="BackgroundCreation"** beállításával ez háttérszálra tehető. Ez a funkció a leghasznosabb listáknál (pl. Facebook ismerősök listázása), de szinten minden esetben érdemes megfontolni a használatát (talán az egyetlen kivétel a **Panorama** háttere – a **Panorama** vezérlőt csak a kép betöltődése után érdemes megjeleníteni).
- A **jpg** dekódolása gyorsabb, mint a **png**-é. Használjunk **jpg**-t, hacsak nincs szükségünk a **png** átlátszóságára (vagy vonalas grafikákon a jóval kisebb fájlméretre).
- Az alkalmazásunkat daraboljuk több DLL-re, így megrövidítve a betöltési időt!
- Az adatkötés lassabb, mint a tulajdonságot közvetlen állítása (pl. **FirstNameTextbox.Text = Person.FirstName**). Ne kezdjünk el azonban megszabadulni minden adatkötéstől, ritka, hogy ilyen drasztikus változtatásokra van szükség!
- Az UI szálát hagyjuk szabadon – a hosszabb műveleteket háttérszálon végezzük, és lehetőleg ugyanitt kommunikáljunk a felhővel is.

## Listák

Mérvadó vélemények szerint a Windows Phone 7.0 kriptónitja (a zöld ásvány, ami Supermant legyengítette) a listák teljesítménye volt. Szerencsére a Mangóban számos javításon esett át a listakezelés, és többek között a fentebb már említett Input szálnak köszönhetően nagyságrendeket javult az interaktivitása.

Olyannyira így van ez, hogy a könyvhöz készítettem egy „állatorvosi lovat”. Egy olyan listát, aminek az **ItemTemplate**-je tele van „teljesítménygyilkos” elemekkel. Összetett, nehezen számolható elrendezés, sok szöveg, kép (13-11 ábra).



13-11 ábra - Összetett ItemTemplate

Amikor kipróbáltam a telefonon, jött a meglepetés – nem volt tapasztalható semmi probléma! Az animáció 60 fps-el döngetett, a lista azonnal reagált a mozdulatokra. „Blanking” sem volt tapasztalható (az a jelenség, amikor olyan gyorsan pörög egy lista, hogy az UI szólnak az újabb elemeket nincs ideje kirajzolni, ezért üres pixelek jelennek meg).

Természetesen a Mangóban is lehet rosszul teljesítő listákat készíteni. Egy-két lassú **ValueConverter** vagy hatalmas képek picire lekicsinyítve, esetleg némi gradiens a háttérben már gondot okozhatnak. De ha rendeltetésszerűen használjuk a listákat és odafigyelünk a Fill Rate-re, ritkán fogunk problémába ütközni.

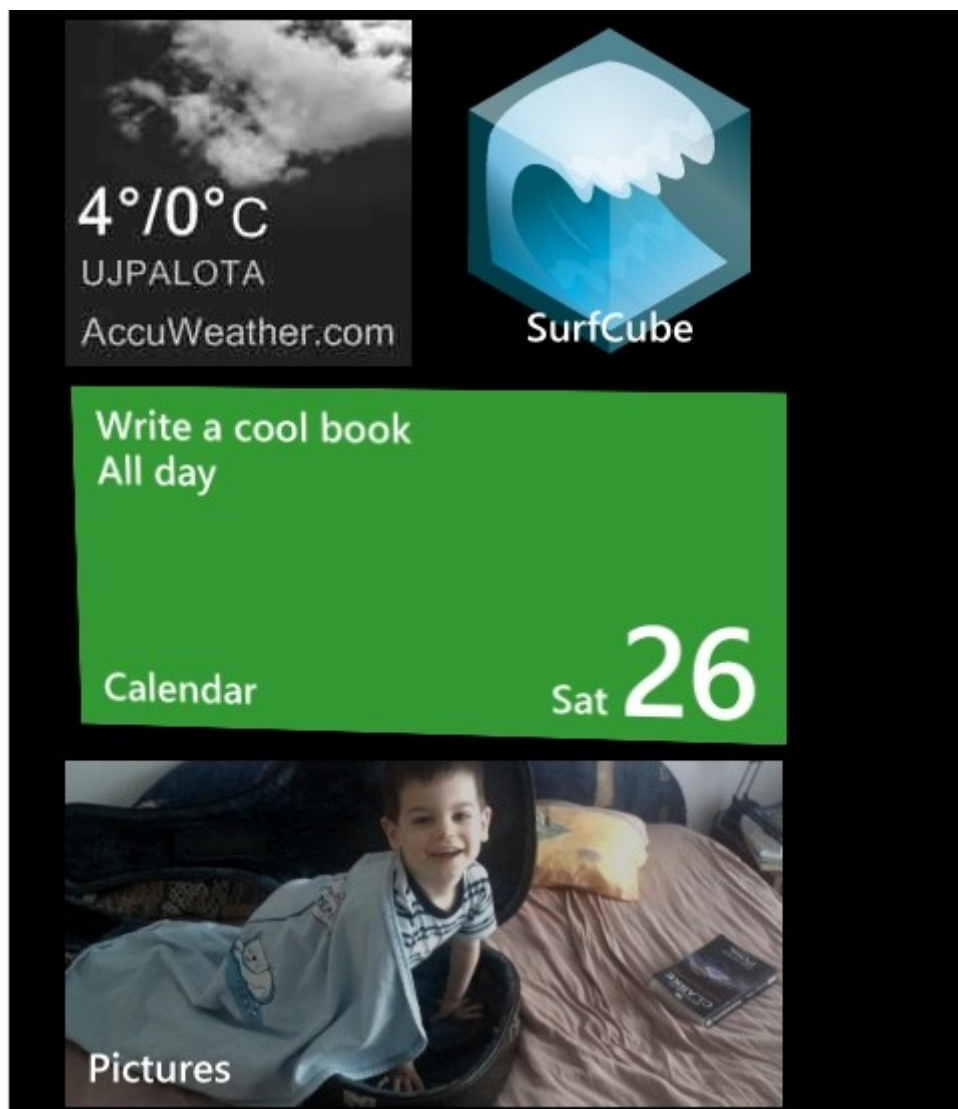
## Szubjektív teljesítmény

Szubjektív teljesítmény alatt az alkalmazás sebességének az *érzetét* értjük, szemben a valódi, mérhető sebességgel. Nemegyszer a szubjektív teljesítmény fontosabb a felhasználónak, mint a valóságos.

Már a fejezet elején is volt arról szó, hogy az interakció közvetlensége miatt a teljesítmény-elvárások nagyobbak egy érintőképernyős alkalmazással szemben, mint egy egeres-billentyűzetes programnál. Ha a program azonnal reagál, és jelzi, hogy vette az utasításunkat, esetleg egy kis animációval szórakoztat, amíg dolgozik, sokkal gyorsabbnak érezzük. Ráadásul elmúlik az a bizonytalan érzés, hogy „megnyomtam? Nyomjam meg még egyszer?” – ami a taktilis visszajelzés hiánya miatt amúgy is gyakori az érintőképernyők használóinál.

#### *Tilt effektus*

A *tilt* (bedöntés) effektus elsősorban a visszajelzést segíti. Ez a Metro nyelv egyik jellegzetes eleme – az aktiválható elemek az érintés helyétől függően 3D-ben bedőlnek, elengedés után pedig visszaugranak a helyükre. A bedöntés működése jól látható a Start képernyőn (13-12 ábra). Használatával már a lenyomás pillanatában tudathatjuk a felhasználóval, hogy a telefon „vette” az érintést.



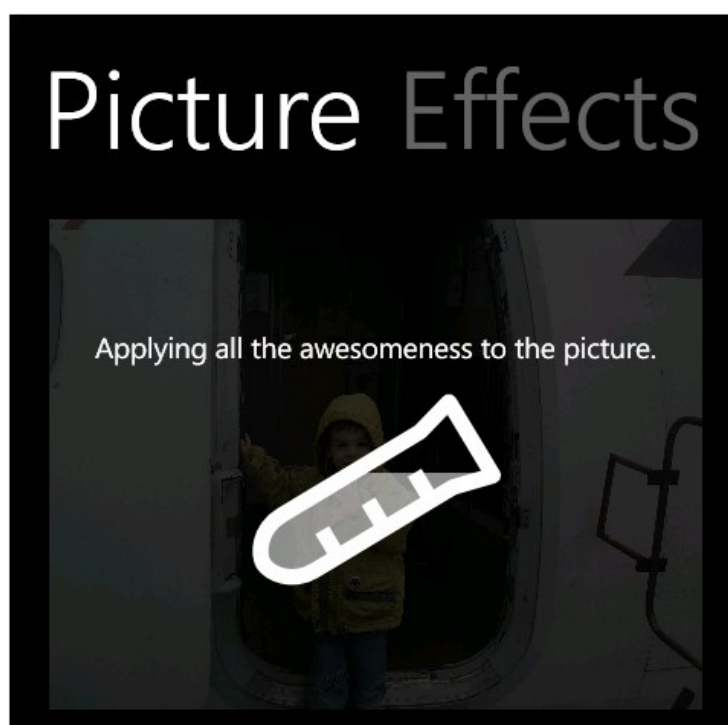
**13-12 ábra: a tilt (bedöntés) effektus segíti a visszajelzést**

A hatás létrehozásához a Silverlight for Windows Phone Toolkit-ban található `TiltEffect` osztályt használhatjuk (lásd még <http://www.windowsphonegeek.com/articles/Silverlight-for-WP7-Toolkit-TiltEffect-in-depth>). Akár egyetlen kódsorral az egész `PhoneApplicationPage`-re engedélyezhetjük az effektust, ezt a kis pluszmunkát mindenképpen megéri.

## A folyamat jelzése

Hosszan tartó folyamatoknál (pl. szerverre várakozás, hosszas számítások) érdemes a felhasználónak jelezni, hogy az alkalmazás dolgozik. Erre bevált módszer a **ProgressBar** használata, vagy egy animáció (pl. pörgő kör, mozgó pöttyök) megjelenítése.

Ha tudjuk, hogy a folyamatban nagyjából hol tartunk, érdemes határozott folyamatjelzőt választanunk. Erre használhatjuk akár a **ProgressBar**-t, a **SystemTray.ProgressIndicator**-t, a Toolkit-ből a **PerformanceProgressBar** vezérlőt, vagy saját megoldást, mint ahogy a Pictures Lab készítője tette (13-13 ábra). Azonban, ha a folyamat időtartama bizonytalan (pl. egy szerver válaszára várunk) a **ProgressBar** használata kerülendő, mert a másik két megoldással szemben túlságosan lefoglalja az UI szálát. Határozatlan esetben használhatunk még saját animációt is (pl. egy kör forgatása), amennyiben az a GPU-n fut. Hosszan tartó folyamatoknál fontos még, hogy a munkát háttérszálon végezzük, megőrizve ezzel az UI szál válaszképességét.



13-13 ábra: a Pictures Lab képszerkesztő program saját design-nal jelzi, hogy hol tart a számítási folyamatban (engedéllyel használva)

## Köztes animációk

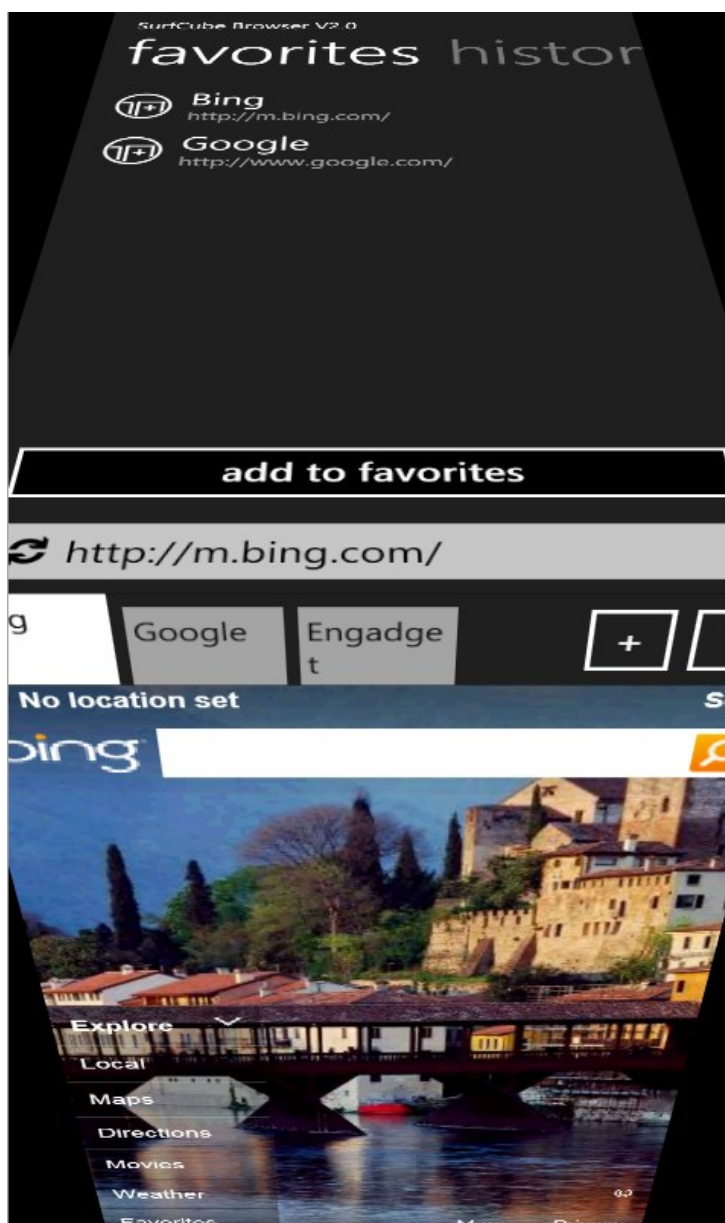
Akár egy fél- vagy egy másodperces számítást is azonnalinak lát a felhasználó, ha a kérdéses másodperc alatt történik valami a képernyőn. A WP7 animációi ideálisak erre a célra, főleg, ha GPU-optimalizáltan tudjuk őket végrehajtani. Például egy lapozás vagy *fade* animáció közben megtörténhet a betöltendő oldal layout-számítása és raszterizálása, és így az animáció végén az új információ azonnal megjelenhet (vagy beúszhat).

## Amikor a lassabb gyorsabb(-nak tűnik)

A szubjektív teljesítmény optimalizálása némileg hasonlít egy bűvészmutatványhoz. A figyelem elterelésével leplezzük a lényegét és átverjük az érzékeket. Jó példa lehet erre a SurfCube böngészőben a könyvjelzőre való navigálás. Mivel a könyvjelzők a kocka tetején helyezkednek el (13-14. ábra), egy könyvjelző kiválasztása után a kockát vissza kell forgatni az első oldalára, ahol a böngésző található. Az objektív gyorsaság érdekében a könyvjelző megérintése után azonnal elindítottuk a betöltődést és a forgatást. Azonban a böngésző vezérlő a CPU-t lefoglalja (főleg navigáció elején), és így a visszaforgató animáció nem látszódik - ami még rosszabb, a könyvjelző lenyomásáról csak pár tized másodperc késéssel



értesül a felhasználó. Ezért változtattunk a működésen, és a böngészőt most már csak a forgatás animáció befejeztével irányítjuk át az új címre. Így a teljes oldal-betöltődési idő (az érintéstől számítva) ugyan néhány tized másodperccel megnőtt, de az azonnali reakció miatt a felhasználók egyértelműen (de tévesen) úgy érezték, hogy az alkalmazás sokat gyorsult.



**13-14 ábra: a SurfCube visszafordul az első oldalra a könyvjelző kiválasztása után, és csak a fordulás után kezdi meg az oldal betöltését (engedéllyel használva)**

## Memória-optimalizálás

A memória takarékos használata a szűkös erőforrások mellett már csak azért is fontos, mert a Marketplace az alkalmazás befogadásakor igen szigorúan ellenőrzi a memória használatát. A szabály az, hogy 256 Mbyte-os telefonon max. 90 Mbyte-ot használhat a program, nagyobb memóriával rendelkező készülékeken ennél többet is.

Természetesen a Silverlight-ban és a .NET-ben megszokott optimalizáló fogások Windows Phone-on is működnek. Érdekes azonban áttekinteni néhány platform-specifikus területet.

## A memória foglálás kijelzése

Sajnos a fentebb már részletezett teljesítmény-mérő eszközök között nem szerepel a használt memória kijelzése. Szerencsére Peter Torr készített egy **MemoryDiagnosticsHelper** osztályt (<http://blogs.msdn.com/b/ptorr/archive/2010/10/30/that-memory-thing-i-promised-you.aspx>), aminek segítségével folyamatosan megjeleníthető az alkalmazásunk memóriahasználata. Miután a **MemoryDiagnosticsHelper.cs** fájlt hozzáadtuk az alkalmazáshoz, egyetlen sorral elindítható:

```
MemoryDiagnosticsHelper.Start(TimeSpan.FromMilliseconds(500), true);
```

Amennyiben az alkalmazást debuggerrel futtatjuk, a **MemoryDiagnosticsHelper** még egy **Debug.Assert**-el is jelzi, hogy túlléptük a 90 Mbyte-os limitet.

A **MemoryDiagnosticsHelper** – ami a fenti példában 500 ms-onként fut és a Garbage Collector-t minden alkalommal működteti – hátrányosan befolyásolja az alkalmazás teljesítményét: akadozhatnak az animációk, lassabbak lehetnek a számítások. Jómagam is jó néhány felesleges órát eltöltöttem korábban folyamatosan működő animációk akadozása után nyomozva, miközben csak a **MemoryDiagnosticsHelper** bekapcsolása okozta a gondot.

## Optimalizálás

Ha a fenti kódot a fejezet elején található példaprogramba beírjuk, láthatjuk, hogy 500 csillag esetén kb. 9Mbyte-ot foglal az alkalmazás, míg 5000-nél 16-ot. Nyilvánvaló tehát, hogy a vizuális fa mérete befolyásolja a memória-használatot, de szerencsére viszonylag ritka, hogy több ezer vagy tízezer elem található a fában, így ennek hatása elhanyagolható.

A legnagyobb memória-falók természetesen a képek. Hiába csak 100Kbyte-os egy 800x480 pixeles fénykép **jpg** formában – a telefonnak minden egyes pixellel kell számolnia, így az 800x480x4 byte-ot (több, mint 1.5 Mbyte) foglal a memóriában. Egy **Panorama** háttere ennek többszöröse is lehet. Ráadásul a megjelenített vezérlők szintén bitmap formában vannak jelen. Egy lista esetén a látható elemeken kívül még néhány további is renderelésre kerül, és a pixeljei a memóriában tárolódnak. A már messzire elgörgötött elemek ugyan törölődnek a memóriából, de a helyükre a görgetés során új elemek kerülnek (ezt hívjuk virtualizációnak, és a **ListBox** vezérlőben ez az alapértelmezett működés). Könnyen elképzelhető, hogy egy sokelemes panorama, ahol minden elem egy vagy több további, gördíthető listát tartalmaz, már igen komoly memóriát felemészt. Az alapértelmezett **Panorama** alkalmazás például 28 Mbyte memóriát eszik, anélkül, hogy képek lennének a listákban.

A képeknél ezért (és a letöltési, nemeg a dekódolási sebesség érdekében is) érdemes a képeket a végleges méretükben kezelni. Tehát, ha egy 100x100-as thumbnail-t jelenítünk meg, de a szerverről 500x500-as képet kapunk, akkor legjobb, ha a szerveren készítünk 100x100-asat is. Ha ez nem lehetséges, akkor a kép dekódolásakor már a végső méretet adjuk meg:

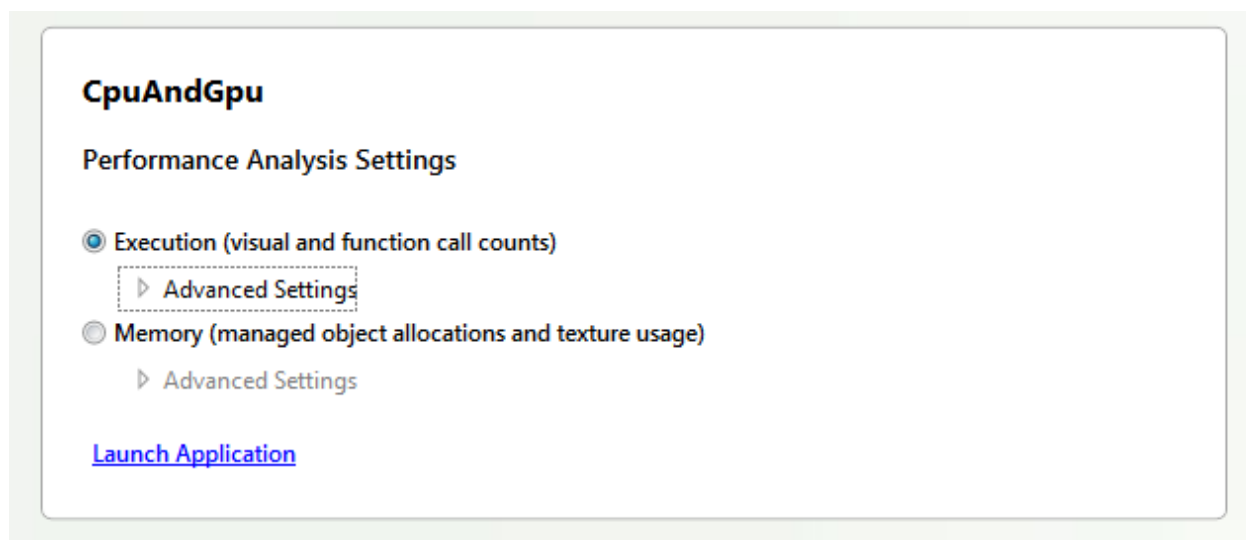
```
PictureDecoder.DecodeJpeg(sourceStream, 100, 100);
```

## Windows Phone Performance Analysis

Egy bonyolultabb alkalmazásban nem egyszerű megtalálni a teljesítmény-problémák okát. Szerencsére a Windows Phone SDK része a Windows Phone Performance Analysis (WPPA) eszköz, amely pont ebben segít. Próbáljuk ki a fejezet első mintapéldáján, az 5000 csillagot kirajzoló programon!

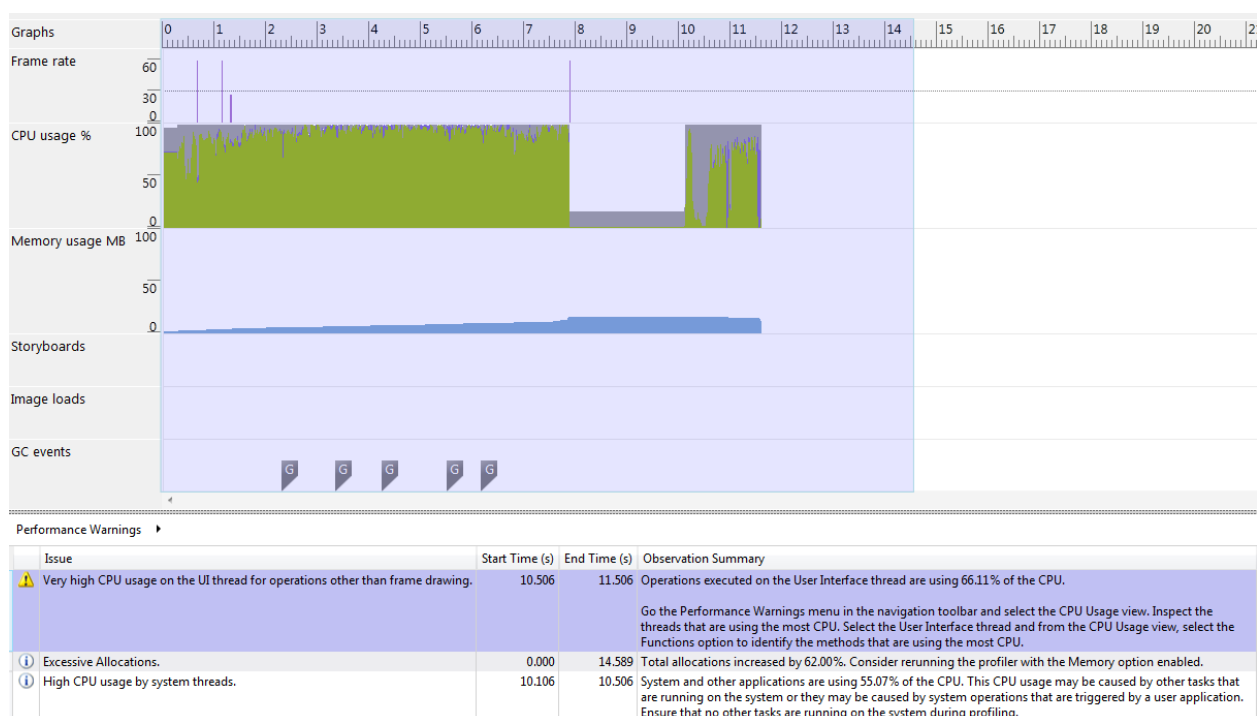
A tesztelést minden esetben a fizikai eszközön végezzük, mivel az emulátor – a már említett okok miatt – hamis eredményt fog adni. A pontosabb mérés érdekében váltsunk Release konfigurációba, majd a Visual Studio Debug menüjében található „Start Windows Phone Performance Analysis” menüponttal indítsuk el a WPPA-t! A 13-15 ábrán látható képernyő jelenik meg, majd a Launch Application megnyomása után

elindul az alkalmazás a telefonon. Várjuk meg, amíg megjelennek a csillagok, majd állítsuk le az adatgyűjtést!



**13-15 ábra: A Windows Phone Performance Analysis indítóképernyője**

Némi számolás után a 13-16 ábrán látható eredményt kapjuk (nyomjunk egy jobb gombot, majd a Select All-t a grafikonon). Mit jelent mindez?



**13-16 ábra: Az analízis eredménye**

A felső skála a másodpercekben mért időt mutatja. Alatta a Frame Rate azt mutatja, hogy egy másodperc alatt a képernyő hányszor került újrarajzolásra. Mivel a csillagok kirajzolása után az alkalmazás nem csinál semmit, ezért nincs itt sok értékelhető adat. Jellemzően 30 és 60 közötti értékeket szeretünk itt látni.

A CPU-használatot mutató grafikonon a fehér szín azt jelzi, amikor a CPU nem csinál semmit. A zöld szín az UI szálat, a szürke a rendszer-szálat jelenti, az esetenként megjelenő lila pedig a az alkalmazás nem UI szálaikat (beleértve a Compositor és a háttér-szálaikat).



A memória-használat grafikonja egyértelmű. Alatta a program által indított **Storyboard**-ok (piros – CPU, lila – nem CPU) és képbetöltések találhatók – ezek ebben az alkalmazásban nem fordulnak elő. Legvégül pedig a Garbage Collector eseményei.

Ha fent kijelölünk egy legalább fél másodperces időtávot, az alsó részben a tipikus teljesítmény-problémákra hívja fel a figyelmünket a program. Például az első sorban azt jelzi, hogy magas CPU használatot mért az UI szálon – olyan CPU használatot, ami nem feltétlenül a rajzolással van összefüggésben.

Ha a Performance Warnings melletti kis nyílacsikra kattintunk, megtekinthetjük a frame-eket, illetve a CPU-használatot. Először válasszuk ki a Frames-t, ekkor a 13-17 ábrán látható eredményt kapjuk!

Performance Warnings ▶ Frames ▶							
Frame	Start Time (s)	Duration (ms)	CPU Time (ms)	CPU Usage %	GPU Time (ms)	Fill Rate	Texture Count
0	0.638	1.465	0.000	0.00 %	1.129	0.00	0
1	0.648	4.730	1.000	20.00 %	4.608	0.00	0
2	1.129	1.221	0.000	0.00 %	1.129	0.00	0
3	1.272	36.499	0.000	0.00 %	36.407	0.00	0
4	6.767	1095.215	1012.000	92.42 %	3.296	0.88	1

**13-17 ábra: Az alkalmazás első 5 képkockájának adatai**

Látható, hogy az alkalmazás a futása során 5 képkockát készített (0-tól számozva). Ebből az ötödik (utolsó) kocka az, amikor az ellipsziseket kirajzoltuk. Ezt több mint 1 másodpercig rajzolta a telefon (Duration és CPU time). Látható még az adott frame Fill Rate-je és a textúrák száma is.

Nézzük meg, mi került egy másodpercbe a 4-es képkocka kiszámításánál! Ehhez válasszuk ki a 4-es kockát, majd a Frames szövegtől jobbra található kis háromszögre kattintva válasszuk a Functions menüpontot. Láthatjuk, hogy a 4. képkocka megrajzolásához milyen függvényhívásokat hajtott végre a rendszer. Az Inclusive samples-re kétszer kattintva, csökkenő sorrendbe rendezhetjük a hívásokat, és a 13-18 ábrán látható eredményből le is vonhatjuk a következtetést: a legtöbb időt a **MeasureOverride** és **ArrangeOverride** – tehát az elrendezéshez szükséges feladatok veszik el. Ezt az időt rövidíthettük le, amikor Grid helyett Canvas-t használtunk.

Az Inclusive azt jelenti, hogy az adott függvényből hívott további függvények is beleszámítanak, az exclusive pedig, hogy csak a függvény saját idejét számítjuk.

Method Name	Inclusive Samples	Exclusive Samples	Inclusive Samples (%)	Exclusive Samples (%)	Thread Name
[ Native Function ]	28	28	100.00 %	100.00 %	User Interface Thread
MS.Internal.XcpImports.FrameworkElement.MeasureOverride(System.Windows.FrameworkElement, System.Windows.Size)	24	0	85.71 %	0.00 %	User Interface Thread
Microsoft.Phone.Controls.PhoneApplicationFrame.MeasureOverride(System.Windows.Size)	24	0	85.71 %	0.00 %	User Interface Thread
System.Windows.FrameworkElement.MeasureOverride(int, double, double, double&, double&)	24	0	85.71 %	0.00 %	User Interface Thread
System.Windows.FrameworkElement.MeasureOverride(System.Windows.Size)	10	0	35.71 %	0.00 %	User Interface Thread
Microsoft.Phone.Controls.PhoneApplicationFrame.ArrangeOverride(System.Windows.Size)	4	0	14.29 %	0.00 %	User Interface Thread
System.Windows.FrameworkElement.ArrangeOverride(int, double, double, double&, double&)	4	0	14.29 %	0.00 %	User Interface Thread
MS.Internal.XcpImports.FrameworkElement.ArrangeOverride(System.Windows.FrameworkElement, System.Windows.Size)	4	0	14.29 %	0.00 %	User Interface Thread
System.Windows.FrameworkElement.ArrangeOverride(System.Windows.Size)	4	0	14.29 %	0.00 %	User Interface Thread
MS.Internal.FrameworkCallbacks.ManagedPeerTracer.Invalidate(int, int, Rute, Rute)	1	0	3.57 %	0.00 %	User Interface Thread

**13-18 ábra: A 4. képkocka elkészítéséhez szükséges függvényhívások**

De mi a helyzet az ellipszisek megrajzolásával? Ehhez a Frames melletti háromszögből válasszuk az Element Types lehetőséget, és láthatjuk az 5000 ellipszist. Még látványosabb, ha a Visual Tree for Frame 4 opciót választjuk (13-19 ábra).

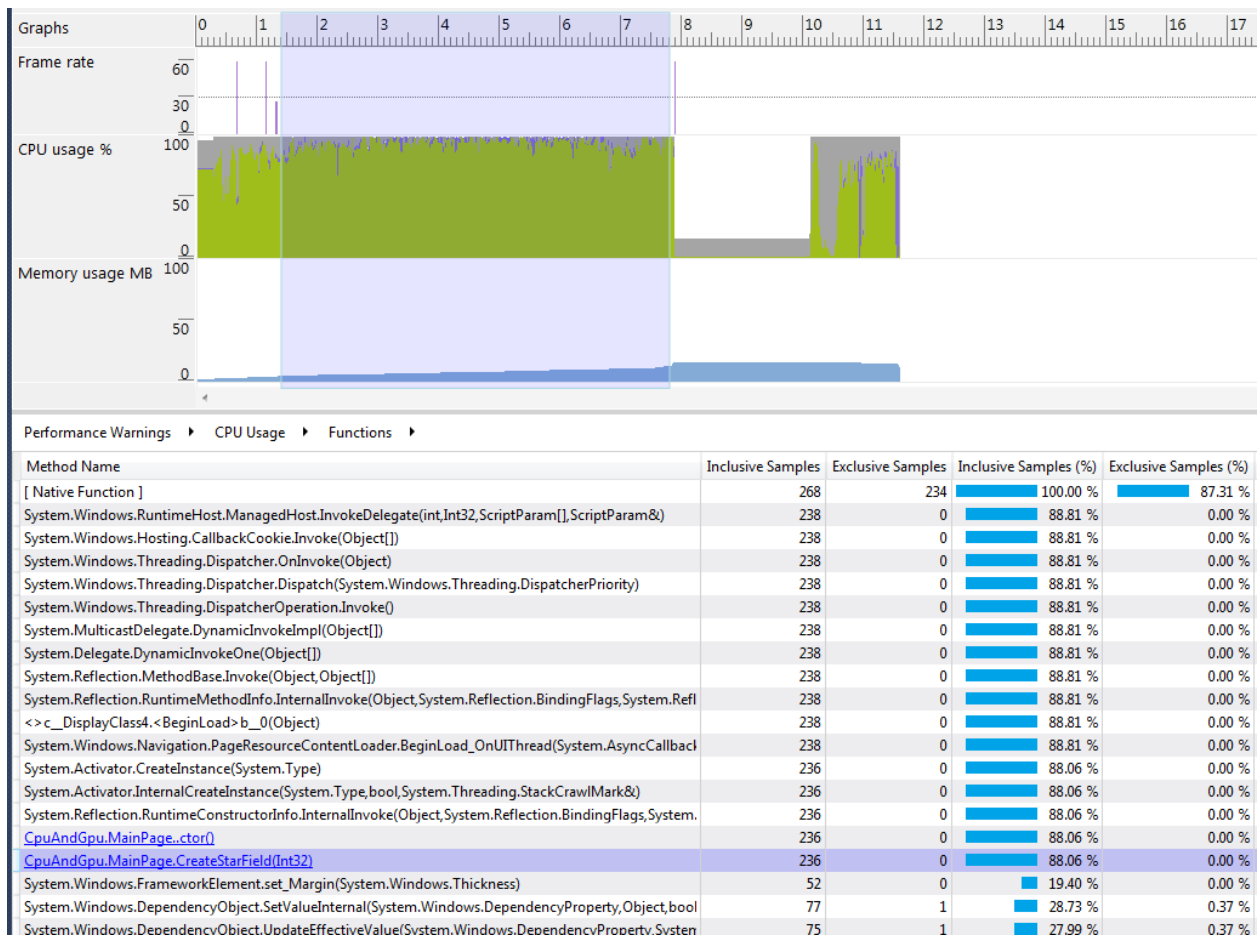
Performance Warnings ▶ Frames ▶ Visual Tree for Frame 4 ▶

Type Name	Name	Updates Check	Total Draw Time (incl.) (ms)	Total Draw Time (incl.) %
System.Windows.IRootVisual		1	965.000	100.00 %
Microsoft.Phone.Controls.PhoneApplicationFrame		1	964.000	99.90 %
System.Windows.Controls.Border	ClientArea	1	951.000	98.55 %
System.Windows.Controls.ContentPresenter		1	951.000	98.55 %
CpuAndGpu.MainPage		1	680.000	70.47 %
System.Windows.Controls.Grid	LayoutRoot	1	679.000	70.36 %
System.Windows.Controls.StackPanel	TitlePanel	1	107.000	11.09 %
System.Windows.Controls.TextBlock	ApplicationTitle	1	101.000	10.47 %
System.Windows.Controls.TextBlock	PageTitle	1	5.000	0.52 %
System.Windows.Controls.Grid	ContentPanel	1	572.000	59.27 %
System.Windows.Shapes.Ellipse		1	0.000	0.00 %
System.Windows.Shapes.Ellipse		1	0.000	0.00 %
System.Windows.Shapes.Ellipse		1	0.000	0.00 %
System.Windows.Shapes.Ellipse		1	0.000	0.00 %
System.Windows.Shapes.Ellipse		1	0.000	0.00 %

**13-19 ábra: A 4. képkocka vizuális fája és az egyes elemek megrajzolásához szükséges idő**

Itt fában kibontva látható, hogy melyik elemmel mennyit foglalkozott a telefon. Talán meglepő lehet, hogy az **ApplicationTitle TextBox** kezelése egy tized másodpercbe került. Ennyire lassan rajzolna szöveget a futtatókörnyezet? A **PageTitle** viszont csak 5 ms. A jelenség oka, hogy az első szöveg kirajzolásánál a környezetnek még a **TextBox** osztályt is be kellett töltenie és értelmezni, míg a második **TextBox**-nál már erre nem volt szükség, csak magára az elrendezésre és a rajzolásra.

Visszatérve a képkockákra (13-17 ábra), éles szeműek észrevehették, hogy a 4-es kockát a program indítása után 6.767 másodperccel kezdte csak el a telefon számolni. Mi történt vajon a 3-as kocka vége és a 4-es eleje között, közel 6 másodpercig? Erre a kérdésre a választ úgy kaphatjuk meg, ha a diagramon kijelöljük a 3-as és 4-es képkocka közötti területet, majd a Performance Warnings - CPU Usage - Functions nézethez navigálunk (13-20 ábra).



**13-20 ábra: A CreateStarField vitte el a 3-as és 4-es frame közötti idő 88%-át.**

Látható, hogy a telefon az ideje jelentős részét a **MainPage** konstruktorában, sőt azon belül a **CreateStarField** metódusban töltötte.

Érdeemes kísérletezni még a többi lehetőséggel, és persze saját alkalmazásokkal is! A Windows Phone Performance Analysis elsajátítása nem egyszerű, de az egyik leghasznosabb eszköz lehet a fejlesztés során a teljesítmény-problémák azonosításában.

## Összefoglalás

Egy egész könyvet lehetne írni arról, hogy miképp készítsünk gyors, kitűnő teljesítményű Windows Phone alkalmazásokat. Ez a fejezet csak az út elejét tudta megmutatni – a CPU és GPU használatát, a lassú betöltődés és navigáció, az akadozó animációk lehetséges okait és megoldását. Szó volt a szubjektív teljesítményről, a memóriefoglalás minimalizálásának módjairól és a mindezeket elemezni képes Windows Phone Performance Analysis eszközről is.