Novák István, Fülöp Dávid, Petró Emil, Fár Attila Gergő,
Farkas Bálint, Turóczy Attila, Kiss Balázs, Árvai Zoltán

Windows 8 fejlesztés

lépésről lépésre

Microsoft

J.O.S.

Készült a devPortal.hu támogatásával

**A könyv nyomtatott verziója megvásárolható a könyvesboltokban,
és a kiadó webáruházában: www.joskiado.hu**

**Novák István, Fülöp Dávid, Petró Emil,
Fár Attila Gergő, Farkas Bálint,
Turóczy Attila, Kiss Balázs, Árvai Zoltán**

Windows 8 fejlesztés lépésről lépésre

JEDLIK OKTATÁSI STÚDIÓ
Budapest, 2012

Szerkesztette és szakmailag lektorálta: 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
Felelős kiadó: a Jedlik Oktatási Stúdió Kft. ügyvezetője

Nyomta: LAGrade Kft.
Felelős vezető: Szutter Lénárd

ISBN: 978-615-5012-19-8
Raktári szám: JO-0342

Köszönöm kedvesem, Olgi végtelen türelmét.

— Farkas Bálint

Ajánlom a felfedezőknek, akik nem tudnak nyugodtan ülni egy platform felett, mindig valami újat akarnak látni - és ajánlom a harcosoknak, akik már megérkeztek, hogy egy szál IntelliSense-szel felvértezve szembenézzenek a kihívással.

— Fülöp Dávid

Szeretettel ajánlom Szüleimnek és Zsuzsinak, akik végig ösztönözték a munkámat.

— Kiss Balázs

Szerettel Henriettnek, Eszternek és Rékának

— Novák István

Tartalomjegyzék

Előszó

Először a 80-as évek végén találkoztam a Windows operációs rendszerrel – ez még az egyetemen történt –, és az a Windows 2.0-s változata volt. Az élet úgy hozta, hogy az egyetem után az első nagyobb projektem egy másfél évig tartó Windows fejlesztés (egy orvosi képeket kezelő kórházi információs rendszer) volt. Ma több mint 20 évvel a kezdetek után visszatekintve az akkori operációs rendszerre, megdöbbentőnek tűnik a fejlődés...

A Windows történetében a Windows 95 megjelenése óta nem történt ekkora változás az operációs rendszer szemléletében és az alkalmazások fejlesztésének megközelítésében, mint most a Windows 8 kapcsán. A Windows az asztali gépekről átkerült a táblagépekre és mobil eszközökre is. Olyan fejlesztői platformot kínál – teljesen új stílusú alkalmazásokkal –, amelynek segítségével nemcsak a .NET fejlesztők, de a más platformokról érkezők is felhasználhatják meglévő C++ vagy HTML/JavaScript tudásukat arra, hogy rövid idő alatt látványos, jól kezelhető alkalmazásokat hozzanak létre. Ez a könyv azt tűzte ki céljául, hogy bevezeti az Olvasót a Windows 8 stílusú alkalmazások készítésének alapjaiba.

A könyv kialakítása során a korábban már jól bevált gyakorlatot követtük: összegyűjtöttük a közösség lelkes szakembereit, akik nemcsak arra vették a fáradságot, hogy kipróbálják a Windows 8 korai kiadásait, hanem késztetést is éreztek arra, hogy tapasztalataikat és tudásukat más fejlesztőkkel megosszák. A szerzők rekordidő alatt hozták létre ezt a könyvet! A Windows 8 végleges változatát 2012. augusztus közepétől lehetett letölteni, és az azóta eltelt néhány hét alatt készült el az itt olvasható 13 fejezet. **Árvai Zoltán** (7. fejezet), **Fár Attila Gergő** (9. és 10. fejezetek), **Farkas Bálint** (11. fejezet), **Fülöp Dávid** (2. és 4. fejezetek), **Kiss Balázs** (8. fejezet), **Novák István** (1., 3. és 12. fejezetek), **Petró Emil** (5. és 6. fejezetek) és **Turóczy Attila** (13. fejezet) az itt leírtakat mind saját tapasztalataik, a Windows 8 megismerésébe fektetett munkájuk alapján állították össze.

Az egyes fejezeteket igyekeztünk úgy kialakítani, hogy azok az elsőtől az utolsóig folyamatosan feldolgozhatók legyenek. Nem törekedtünk arra, hogy azt tankönyvszerűen – egyenetlen és száraz stílusban – írjuk, minden egyes fejezet alkotójának stílusát, megközelítésmódját tükrözi.

A könyv és a hozzá tartozó programozási mintapéldák a **Devportal.hu** oldalon is elérhetők.

Novák István

Budapest, 2012. október

1. A Windows alkalmazásfejlesztés rövid története

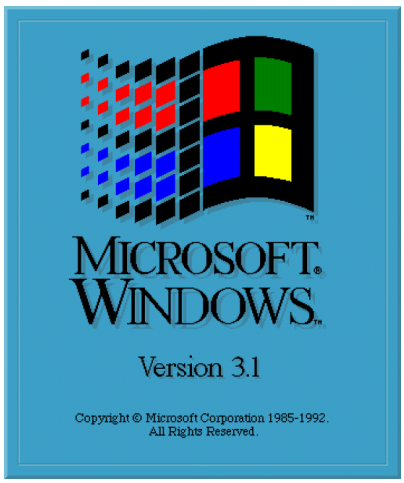
Ebben a fejezetben az alábbi témákat ismerheted meg:

- A Windows operációs rendszer életútja és fejlődése az elmúlt 27 évben
- A Windows alkalmazásfejlesztését leginkább meghatározó eszközök és technológiák – a kezdetektől napjainkig
- A Windows 8 paradigmaváltása az alkalmazásfejlesztés megközelítési módjában
- A Windows 8 stílusú alkalmazások fogalma

A Windows életútja

Hosszú út vezetett a Windows 8 megjelenéséig a Windows első változatának 1985. november 20-ai kibocsátása óta. Akkoriban a Microsoft – a grafikus felhasználói felületek iránti érdeklődésre adott válaszként – az MS-DOS operációs rendszeréhez egy kiegészítő komponenst készített, amely teljesen megváltoztatta a személyi számítógépekhez kapcsolódó világot. Ez a komponens a Windows volt. Az első változat nagyon egyszerű grafikát használt, és valójában egy új front-end felület volt az MS-DOS-hoz, és nem önálló operációs rendszer.

Közel hét évnek kellett eltelnie, mire az első széles körben használt változat – a Windows 3.1 – 1992 márciusában megjelent. Ez a 16-bites operációs rendszer lehetővé tette a *multitasking* alkalmazását – egy olyan környezetben, ahol a felhasználók nem voltak ehhez hozzászokva. Virtuális eszközkezelőket tartalmazott, amelyek megoszthatók voltak MS-DOS alkalmazások között. Védett üzemmódjának (*protected mode*) köszönhetően képes volt több megabájt memória megcímzésére – abban az időben az 8086 processzorcsaládba tartozó chipek csak 640 kilobájtot engedélyeztek alap üzemmódban – virtuális memóriakezelő szoftver nélkül. Az ebben az időben számítógépekkel foglalkozó felhasználók jól emlékezhetnek az operációs rendszer induló képernyőjére, ahogyan az 1-1 ábrán is látható.



1-1 ábra: A Windows 3.1 induló képernyője

Az 1995 augusztusában kibocsátott Windows 95 egy valódi 32-bites operációs rendszer volt, amely a *preemptive multitasking* működést is támogatta – vagyis az operációs rendszer képes volt egy alkalmazástól elvenni a vezérlést annak aktív közreműködése nélkül. A Windows 95 már nem egy MS-DOS-

ra épülő kiegészítés volt, hanem valóban egy minden szükséges funkcionalitással bíró operációs rendszer – még akkor is, ha ezt a tényt sokan, hosszú ideig vitatták. A Windows XP 2001-es megjelenéséig ezt még néhány változat (Windows 98, Windows ME) megelőzte.

A Windows XP (széles körben ismertté vált logója az 1-2 ábrán látható) az operációs rendszer család legnépszerűbb változata volt a telepítések száma alapján. Furcsa, de sikerét nem annyira az újszerű felhasználói élménynek – az XP az „élmény" (experience) szóból jön – vagy az olyan innovációknak, mint a GDI+ grafikus alrendszer, a ClearType megjelenítés és a 64-bites támogatás köszönhette. A legfőbb oka annak, hogy az XP sikeres lett, az utódjához, a Windows Vistához kapcsolható.

1-2 ábra: A Windows XP logója

A 2006 novemberében megjelent Windows Vista új dizájnelemeket és jelentősen megerősített biztonsági képességeket kapott – ez utóbbi erős kontrasztot jelentett a Windows XP-vel szemben, ahol három szervizcsomagra is szükség volt a biztonsági rések betöméséhez. Ez elegendő lehetett volna, hogy az elődjénél nagyobb népszerűségnek örvendjen, de ezekért a képességekért cserébe a Vistának fejlettebb, nagyobb teljesítményű hardverre volt szüksége. A legtöbb nagyvállalat az IT költségvetésének jelentős részét ekkorra már elköltötte az XP stabilizálására – a Windows XP Service Pack 3 (SP3) után –, és nem találta ésszerűnek a Vistára való átállást. A Vista rövid idő alatt az operációs rendszer család legrövidebb életű tagjává vált.

Ahogyan azt Steven Sinofsky, a Microsoft Windows fejlesztéséért felelős vezetője néhány alkalommal bevallotta, a cég tanult ebből a leckéből, amikor a Windows 7 tervezéséhez hozzákezdtek. A Windows 7 2009 júliusában jelent meg, két évvel és nyolc hónappal a Windows Vista után. Az új operációs rendszer jelentős teljesítményjavulást mutatott a Windows XP-hez és a Vistához képest – gyorsabb indulást és leállást, jobb feladatütemezést a többmagos processzorokon és így tovább. A felhasználói élmény is rengeteget fejlődött, olyan új képességek jelentek meg, mint például a felugró listák, az Aero Peek és Aero Snap technikák, valamint rengeteg apró dolog, amely egyszerűbbé tette az alkalmazások és az ablakok közötti gyors navigációt.

A Windows 7 sikeresen tüntette el a Vista fiaskó nyomait. A Windows csapatnak könnyű lett volna a Windows 7 által kijelölt utat követni, azonban ők egy sokkal nagyobb kihívást vállaltak fel.

A Windows 8 paradigmaváltása

Bár a Windows olyan korban született, amikor a személyi számítógépek a mindennapi életünk részévé váltak, a Windows mégis olyan operációs rendszer maradt, amely elsősorban a vállalatokra és az információval dolgozó munkatársakra fókuszált. Az operációs rendszer legtöbb funkciója azt akarta – gyakran megkövetelte –, hogy a felhasználók a rendszerbe „bedrótozott" munkafolyamatokat kövessék. A felhasználók meg sem tudtak mozdulni anélkül, hogy olyan koncepciókkal kelljen megismerkedniük, mint a fájlok, mappák, biztonsági csoportok, jogosultságok, megosztások, regisztrációs adatbázis és így tovább. Ez a megközelítés tükrözte azt a módot, ahogyan az operációs rendszer tervezői a felhasználókról gondolkodtak.

Azoknak az alkalmazásoknak köszönhetően, amelyek szépen lassan elkezdték felélni az operációs rendszer erőforrásait, rendszeres takarításokra és karbantartásokra volt szükség. Bár minden újabb Windows változat enyhített ezeken a terheken, azokat teljes egészében eltüntetni egyik sem tudta.

A Microsoft megteszi az első lépéseket a fogyasztók felé

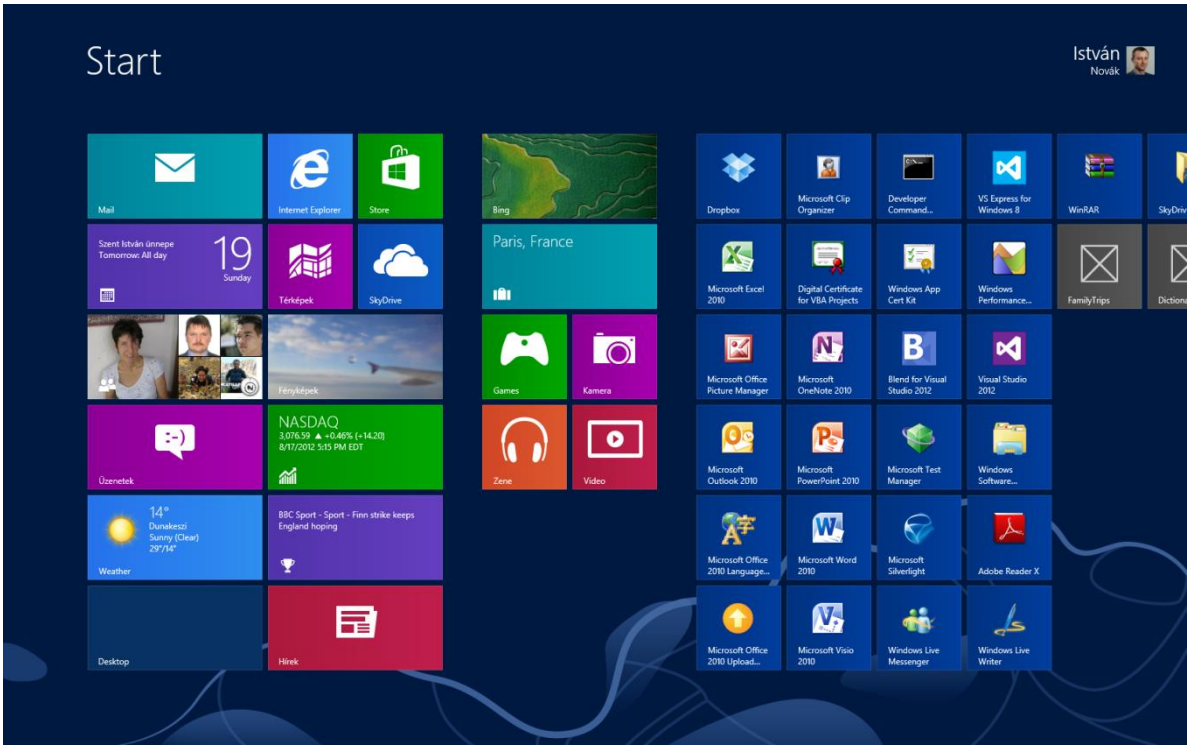
Az Apple felhasználó-centrikus termékei – az iPhone és az iPad – megmutatták a világnak, hogy létezik intuitív együttműködés is a felhasználók és a szoftver között anélkül, hogy szükség lenne a fájl, könyvtár, regisztrációs adatbázis fogalmának vagy a telepítési eljárás folyamatának ismeretére. Hosszú ideig úgy tűnt, hogy a Microsoft nem igazán érti meg ezt a megközelítési módot, de a piaci trendek és eladások arra késztették a céget, hogy a felhasználó-központú eszközök és operációs rendszerek irányába forduljon.

Az első komoly változást a Microsoft viselkedésében 2010 februárjának közepén lehetett észrevenni, amikor a barcelonai Mobile World Congress eseményen a cég először mutatta be a Windows Phone 7-et. Ez az operációs rendszer teljes egészében a felhasználói élményről szólt. A Windows 8 vizuális dizájnja, egyszerűsége, a szép és jól eltalált animációk a felhasználói felületet és a készülék használatát intuitívvá tették. A piac értékelte ezt a váltást, és most, a Windows Phone 7.5 „Mango" változata után a Microsoft már a harmadik versenyző a mobil operációs rendszerek piacán, egyre közelebb kerülve az Apple iOS-éhez és a Google Androidjához.

> *A Windows operációs rendszer családnak már a Windows Phone 7 előtt volt beágyazott eszközökre szánt változata (Windows Embedded), illetve mobil telefonokon működő változata (Windows CE, majd Windows Mobile).*

A Windows 8 megjelenik a színen

A Windows Phone felhasználó-centrikus megközelítése a Windows 8 operációs rendszerhez kapcsolódó élmény része. Amikor az operációs rendszer elindul – a betöltési idő jelentősen lecsökkent a Windows 7-hez képest – az új Windows 8 Start képernyő nem is emlékeztet a korábban megjelenő asztalra az alján a tálcával. Ehelyett a felhasználó szögletes lapkákkal találja magát szemben, amelyek egy-egy alkalmazást reprezentálnak, amint az az 1-3 ábrán látható.

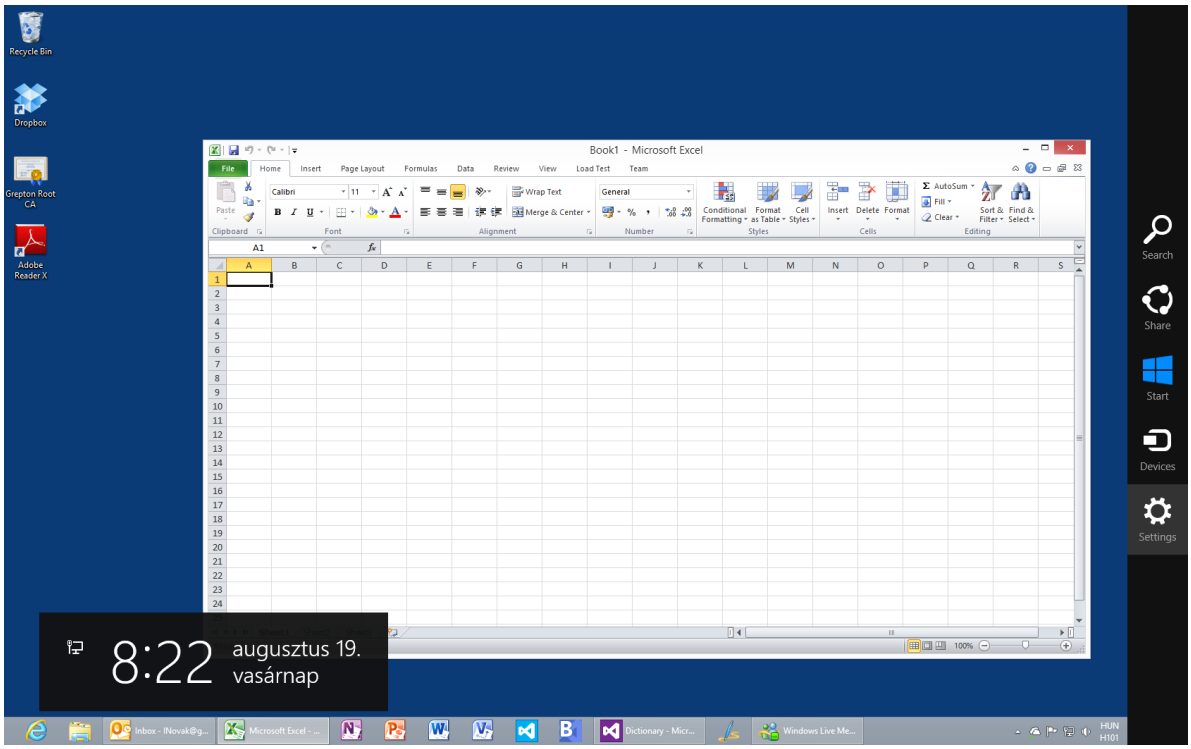


1-3 ábra: A Windows 8 Start képernyője

Ez az új arc nyilvánvaló üzenetet küld a fogyasztóknak. A Windows nem olyan operációs rendszer, amely csak informatikusoknak vagy edzett számítógép-használóknak szól, hanem egyúttal egy fogyasztókat megcélzó eszköz, többpontos érintéssel és táblagépekkel a fejben tervezve. A Start képernyő felülete nagyon intuitív, és a legtöbb ember azonnal birtokba tudja venni használati útmutató nélkül is. Azok a

felhasználók, akik már találkoztak érintőképernyővel okostelefonokon vagy táblagépeken, természetesnek érzik a Start képernyő használatát, az alkalmazások indítását, görgetést, nagyítást és azoknak a gesztusoknak az alkalmazását, amelyeket korábban már más eszközzel megtanultak.

Azoknak a felhasználóknak, akik a Windows-t üzleti alkalmazások futtatására (pl. Word, Excel vagy más az adott vállalatra jellemző alkalmazásra) használják, ez a felhasználói felület paradigmaváltás talán ijesztő lehet. A Windows 8-at úgy tervezték, hogy az teljesen kompatibilis maradjon a már meglévő alkalmazásokkal, és így annak egy asztali üzemmódja is van. Amikor egy a Windows korábbi változatával készített alkalmazást indít el a felhasználó – vagy egy olyat, amelyik nem a Windows 8 stílusú felületet használja –, az a már jól ismert asztali üzemmódban indul el. Például az Excelt elindítva az az 1-4 ábrán látható módon jelenik meg.



1-4 ábra: Az Excel a Windows 8 asztali módjában fut

Ez a Windows 8 második arca, ismerős mindenkinek, aki korábban már használta az operációs rendszert. Ha a megszokott Start menü látható lenne, az aktuális dátum és idő pedig nem, azt hihetnénk, hogy a Windows 7-et használjuk.

Az új Windows 8 Start képernyő nemcsak egyszerű kiegészítése a Windows 7-nek! E mögött az egyszerű képernyő mögött a fogyasztó-központú alkalmazások egy teljesen új világa, a Windows 8 stílusú alkalmazások állnak. Az asztal, a rajta lévő ikonok és téglalap alakú ablakok helyett a felhasználó egyidejűleg egyetlen alkalmazást lát, amely a teljes képernyőt birtokolja. Nincsenek az ablakoknak fejlécei, lezáró gombjai, méretezhető kereteik, sőt egyetlen más elemük sem (ezek összességét az angol terminológiában „chrome” néven említjük), amely a felhasználó figyelmét elvonná az alkalmazás tartalmáról. A Weather alkalmazás, amely az 1-5 ábrán látható, jó példája ennek.

1-5 ábra: A Weather alkalmazás Windows 8 stílusú megjelenítése

Ebben a fejezetben néhány alapvető információt fogsz megtanulni erről az új felhasználói felület paradigmáról, illetve az ilyen alkalmazások fejlesztéséről. Mielőtt azonban az alkalmazásfejlesztés mélységeibe merülnénk, érdemes visszatekinteni a Windows alkalmazásfejlesztés történetére.

A Windows programozási felületek és eszközök rövid története

A Windows története igencsak hiányos lett volna a platformon futó alkalmazások és az ezeket az alkalmazásokat kifejlesztő programozók nélkül. A Microsoft mindig olyan cég volt, ami erős fejlesztői közösséget épített ki termékei körül, beleértve a zászlóshajót, a Windows-t.

> *A fejlesztői közösség fontosságát gyakran hangsúlyozzák a Microsoft vezetői. Ha az interneten rákeresel Steve Ballmer nevére, aligha kerülöd el azokat a videókat, ahol szenvedélyesen ismétli a „developers, developer, developers...” szavakat tucatnyi alkalommal.*

2012-ben a Windows platform már 27 éves. Hosszú élete során nemcsak az operációs rendszer, de az alkalmazások programozási felülete (API) és a fejlesztőeszközök is rengeteget fejlődtek. Ebből a nézőpontból a Windows 8 jelenti a legnagyobb ugrást az operációs rendszer történetében. Ahhoz hogy ezt megértsük, érdemes visszatekintenünk az időben, egészen a Windows alapú alkalmazásfejlesztés első pillanataiig.

A C programozási nyelv hatalma

Bár manapság a Windows alkalmazások készítése mindennapos dolog, ez nem volt így a kezdetekben. Abban az időben a fejlesztők MS-DOS programokon nőttek fel, és azt tapasztalták, hogy a Windows furcsa, kifordított módon működik. Amíg egy jól viselkedő MS-DOS alkalmazás saját maga vezérelt mindent, és meghívta az operációs rendszer funkcióit, addig a Windows őrültségeket csinált! Az operációs rendszer irányította az alkalmazást és hívta annak funkcióit, amikor valamire, például a képernyő frissítésére vagy egy parancs végrehajtására akarta ösztökélni azt.

És nem ez volt az egyetlen merénylet szegény programozók ellen! A C programozási nyelvet használva – akkoriban ez volt „a” nyelv – a legegyszerűbb „Hello, World” alkalmazás nagyon egyszerű volt:

```
#include <stdio.h>

main()
{
    printf("Hello, world");
}
```

Ugyanezt az eredményt a Windows alatt elérni már némi munkát kívánt. Egészen komoly kódot kellett a **`printf`** utasítás köré keríteni, és ezt bárminek hívhattuk, de intuitívnak semmi esetre sem, amint azt az 1-1 kódlista jól mutatja:

1-1 kódlista: A „Hello, World” program À la Windows 3.1 (kivonat)

```
#include <windows.h>

/* Export the entry point to be called by Windows */
long FAR PASCAL _export WndProc(HWND, UINT, UINT, LONG)

/* The entry point of the application */
int PASCAL WinMain(HANDLE hInstance, HANDLE hPrevInstance,
    LPSTR lpszCmdParam, int nCmdShow)
{
    static char szApplication[] = "HelloW";
    HWND hwnd;
    MSG msg;
    WNDCLASS wndClass;

    /* Create a Window class */
    if (!hPrevInstance)
    {
        wndClass.Style = CS_HREDRAW | CS_VREDRAW;
        wndClass.lpfnWndProc = WndProc;
        /* A few code lines omitted for the sake of brevity */
        wndClass.hbrBackground = GetStockObject(WHITE_BRUSH);
        wndClass.lpszMenuName = NULL;
        wndClass.lpszClassName = szApplication;
        RegisterClass(&wndClass);
    }

    /* Create a window instance for the class */
    hwnd = CreateWindow(szApplication,
        "My Hello World Program",
        WS_OVERLAPPEDWINDOW,
        /* A few parameters omitted for the sake of brevity */
        hInstance,
        NULL);
    /* Initiate displaying the window */
    ShowWindow(hwnd, nCmdShow);
    UpdateWindow(hwnd);

    /* Manage the message loop */
    while (GetMessage(&msg, NULL, 0, 0))
    {
        TranslateMessage(&msg);
        DispatchMessage(&msg)
    }
}
```

```
/* Processing messages */
long FAR PASCAL _export WndProc(HWND hwnd, UINT message,
    UINT wParam, LONG lParam)
{
    HDC hdc;
    PAINTSTRUCT ps;
    RECT rect;

    switch (message)
    {
        case WM_PAINT:
            hdc = BeginPaint(hwnd, &ps);
            GetClientRect(hwnd, &rect);
            DrawText(hdc, "Hello, world", -1, &rect,
                DT_SINGLELINE | DT_CENTER | DT_VCENTER);
            EndPaint(hwnd, &pd);
            return 0;

        case WM_DESTROY:
            PostQuitMessage(0);
            return 0
    }
    return DefWindowProc(hwnd, message, wParam, lParam);
}
```

Ez a program rengeteg kódsorból áll, mert egy alacsony szintű szolgáltatásokat kínáló API-t használ. Bár a kód hosszú, fontos belső Windows részleteket fed fel. Ezek még a Windows 8-ban is megtalálhatók, de természetesen már továbbfejlesztett formában:

- A program a legelején egy ún. *window class*-t hoz létre a **`wndClass`** struktúra mezőinek feltöltésével és a **`RegisterClass`** metódus hívásával. A *windows class* egy olyan eljárást (*window procedure*) azonosít, amely egy adott ablakhoz küldött üzeneteket dolgoz fel.
- A program a **`CreateWindow`** metódus segítségével létrehoz egy ablakot a regisztrált window class használatával, majd megjeleníti azt a **`ShowWindow`** hívásával. Az **`UpdateWindow`** metódus egy üzenetet küld az ablaknak, hogy frissítse a felhasználói felületét.
- Az alkalmazás lelke az ún. üzenethurok (*message loop)*:

```
while (GetMessage(&msg, NULL, 0, 0))
{
    TranslateMessage(&msg);
    DispatchMessage(&msg)
}
```

- Ez hozzájut az üzenetsorban lévő üzenethez, a billentyűkódokat a megfelelő üzenetekre fordítja le (pl. úgy, mintha egeret használnánk), majd azokat a megfelelő window procedure-höz juttatja el.
- Ha az üzenethurok az alkalmazás lelke, akkor a window procedure a szíve. Az 1-1 kódlistában a **`WndProc`** eljárást az üzenethurok hívja. A **`message`** paraméter az üzenet kódját tartalmazza, és a **`switch`** utasítás veszi körbe az egyes üzenetek feldolgozásához szükséges kódot.
- A **`WM_PAINT`** üzenet mondja meg az ablaknak, hogy frissítse a saját felületét. A **`BeginPaint`** eljárással egy ún. eszköz-kontextust kér, hogy újra tudja az ablak felületét rajzolni. Ez a kontextus írja ki az ablak közepére a „Hello, World” szöveget. A **`ReleasePaint`** eljárás elereszti ezt a kontextust – amely egyébként meglehetősen korlátozott erőforrás a rendszerben.

A kód alapján elképzelheted, milyen időrabló és fájdalmas volt a Windows alkalmazásfejlesztés abban az időben amiatt, hogy a fejlesztőknek alacsony szintű konstrukciókat kellett a Windows API-n keresztül használniuk.

A C++ átveszi a C helyét

1983-ban, mindössze néhány évvel azután, hogy Brian Kernigham és Dennis Ritchie a C nyelv első változatát publikálta (1978-ban), Bjarne Stroustrup új nyelvet alkotott, amely objektum-orientált szemléletet adott a C nyelvhez. Ez a C++ nyelv volt, amely hamarosan a Windows platformon is népszerű lett.

A C++ lehetővé teszi, hogy az egybetartozó adatokat és műveleteket egy osztályba zárjuk, támogatja az öröklődést és a polimorfizmust. A Windows lapos API-ja ezekkel a képességekkel entitások kisebb halmazaival reprezentálható, amelyek az API összetartozó műveleteit és adatstruktúráit egy logikai kontextusba zárják. Például minden olyan művelet, amely ablakok létrehozásához, megjelenítéséhez, kezeléséhez tartozik, betehető egy **`Window`** nevű osztályba.

A C++ megközelítési módja segítette a fejlesztőket abban, hogy jobban áttekinthessék a Windows API-t és könnyebb legyen a Windows fejlesztéshez hozzákezdeniük. Például,a „Hello, World” program 1-1 kódlistában leírt változatának lényegi része objektumok köré szervezhető, amint azt az 1-2 kódlista is mutatja.

1-2 kódlista: A „Hello, World” program váza a C++ programozási nyelvben (kivonat)

```
// --- Class representing the main program
class Main
{
  public:
    static HINSTANCE hInstance;
    static HINSTANCE hPrevInstance;
    static int nCmdShow;
    static int MessageLoop( void );
};

// --- Class representing a window
class Window
{
  protected:
    HWND hWnd;
  public:
    HWND GetHandle( void ) { return hWnd; }
    BOOL Show( int nCmdShow ) { return ShowWindow( hWnd, nCmdShow ); }
    void Update( void ) { UpdateWindow( hWnd ); }
    virtual LRESULT WndProc( UINT iMessage, WPARAM wParam, LPARAM lParam ) = 0;
};

// --- The class representing this program's main window
class MainWindow : public Window
{
    // --- Implementation omitted for brevity
};

// --- Extract from the implementation of the Main class
int Main::MessageLoop( void )
{
    MSG msg;

    while( GetMessage( (LPMSG) &msg, NULL, 0, 0 ) )
    {
        TranslateMessage( &msg );
        DispatchMessage( &msg );
    }
    return msg.wParam;
}
```

```
LRESULT MainWindow::WndProc( UINT iMessage, WPARAM wParam, LPARAM lParam )
{
    switch (iMessage)
    {
        case WM_CREATE:
            break;
        case WM_PAINT:
            Paint();
            break;
        case WM_DESTROY:
            PostQuitMessage( 0 );
            break;
        default:
            return DefWindowProc( hWnd, iMessage, wParam, lParam );
    }
    return 0;
}
```

A C++ által kínált objektum-orientált megközelítésben az objektumok újrahasznosítható kód-könyvtárokba voltak csomagolhatók. A fejlesztők ezekre a könyvtárakra építhették programjaikat, pusztán azokat az objektum-viselkedéseket definiálva, amelyek különböztek a beépítettektől. Például, az 1-2 kódlistában csak a **`Paint`** metódust kellett felüldefiniálni saját ablakaik felhasználói felületének kirajzolásához.

A Windows programozása rengeteget változott a C++ és az objektumkönyvtárak használatával. A Microsoft két könyvtárat is készített, az MFC-t (Microsoft Foundation Classes) és az ATL-t (Active Template Library), amelyek a mai napig ott találhatók a cég kiemelkedő fejlesztőeszközében, a Visual Studióban.

A Visual Basic

A C vagy C++ nyelveken írt alkalmazások rengeteg belső részletet tárnak fel a Windows működésével kapcsolatosan. Néhány esetben fontos ezeknek a dolgoknak az ismerete, de a legtöbbször ez inkább zavaró, és eltereli a programozók figyelmét az alkalmazás valódi funkcionalitásáról.

Az 1991 májusában kibocsátott Visual Basic drasztikusan megváltoztatta ezt a programozási stílust. A belső Windows részletek feltárása helyett a Visual Basic elrejtette azokat, és magas szintű programozási konstrukciókat, például vezérlőket, űrlapokat, modulokat, osztályokat és kódfájlokat kínált. Ahelyett, hogy tucatnyi kódsort kellett volna leírni egyszerű funkcionalitás megvalósításához, a Visual Basic lehetővé tette, hogy a fejlesztők alkalmazásaik valódi, üzletileg hasznos funkcióira fókuszálhassanak. A „Hello, World” programot egyetlen kódsorban le lehetett írni:

```
MsgBox("Hello, World!")
```

Semmi window class, semmi regisztráció, semmilyen üzenethurok programozás! A nyelv magas szintű konstrukciói teljesen szükségtelenné tették az alacsony szintű részletekkel foglalkozást, azok a Visual Basic Runtime-ban kaptak helyet.

Az alkalmazásfejlesztésnek ez a módja — grafikus felhasználói felület (IDE — Integrated Development Environment) használata — ma is a legkedveltebb és a leghatékonyabb. A fejlesztők grafikusan tervezik az alkalmazások űrlapjait és dialógusait az eszközkészletről kiválasztott vezérlők tervezőfelületre húzásával. Minden egyes vezérlő néhány eseménykezelővel rendelkezik, amelyek a környezetből érkező eseményeket dolgozzák fel, például azt, ha a felhasználó kiválaszt egy elemet egy legördülő listában. A programozás nem más, mint az eseménykezelők kódjának megírása.

1993-ban a Microsoft kidolgozott egy bináris szabványt, a COM-ot (Component Object Model), amely lehetővé tette más alkalmazásokban is újrahasznosítható objektumok létrehozását. Több olyan technológia is épült a COM-ra – például az OLE (Object Linking and Embedding) –, amely lehetővé tette az alkalmazások automatizálását. Az 1993 után kibocsátott Visual Basic változatok a COM és OLE képességeket szem előtt tartva kerültek kifejlesztésre. Ez a megközelítésmód olyan sikeres volt, hogy a nyelv egy dialektusa, a VBA (Visual Basic for Applications) a Microsoft Office makrók programozási nyelvévé vált.

A Delphi

Nem a Visual Basic volt az egyetlen fejlesztői környezet, amely letért a C/C++ által járt útról. A Delphi, amit eredetileg a Borland cég fejlesztett ki, az Object Pascal programozási nyelvet használta. Amíg a Visual Basic egy objektumalapú nyelv volt – lehetővé tette objektumosztályok létrehozását beágyazott adatokkal és műveletekkel, de nem támogatta az öröklődést és a polimorfizmust –, az Object Pascal valódi objektum-orientált nyelv volt. A Delphi grafikus felülete – az első változatát 1995-ben bocsátották ki – nagyon hasonlított a Visual Basic felületéhez. A Delphit olyan RAD (Rapid Application Development) eszköznek tervezték, amely adatbázis alapú alkalmazások fejlesztését támogatja, az egyszerűbbektől egészen a nagyvállalati alkalmazásokig.

A termék nagyon gyorsan fejlődött, az első öt évben öt változatot bocsátottak ki. A Delphi volt az első fejlesztőeszköz a Windows platformon, amely 32-bites alkalmazásokat tudott fordítani. A Visual Basic vezérlőihez hasonlóan száznál is több vizuális komponenst tartalmazott (ezek alkották a Delphi Visual Component Library gyűjteményét), amelyeket a fejlesztők azonnal használatba vehettek. Mindezek mellett a fejlesztők létrehozhatták saját komponenseiket, és a meglévő gyűjteményhez adhatták azokat.

A .NET felbukkanása

2002-ben a .NET keretrendszer új lendületet adott a Windows alkalmazásfejlesztésnek. A .NET programok egy közbenső nyelvre (MSIL, Microsoft Intermediate Language) fordulnak, és ez a közbenső kód kerül átalakításra a CPU-n futtatható gépi kódra futásidőben az ún. JIT (Just-In-Time) fordító segítségével. Ez az új megközelítési mód jó néhány fontos paradigmát hozott a Windows alkalmazásfejlesztésbe, amint azt a következő lista a teljesség igénye nélkül bemutatja:

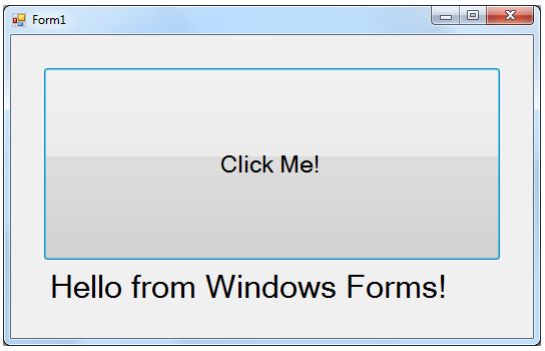
- A .NET előtt minden egyes nyelv (és fejlesztői környezet is) a saját futásidejű könyvtárát használta. Egy új nyelv megtanulása egy új futásidejű könyvtár megtanulását is jelentette. A .NET-ben minden nyelv ugyanazt a közös futásidejű könyvtárat használja, szóval az abban lévő „tudás” minden .NET nyelvben felhasználható. Bár 2002-ben még csak két programozási nyelvet támogatott a Microsoft (ezek a C# és a Visual Basic.NET voltak), ma már ezeknek a nyelveknek a száma 100 felett jár. A Microsoft saját maga az F# nyelvet adta még ehhez a listához, és szintén támogatja a saját közössége által kifejlesztett IronRuby és IronPython nyelveket is.
- A futásidejű környezet kezeli a memóriafoglalásokat és az objektumok megszüntetését is az ún. „szemétgyűjtő” (garbage collection) mechanizmus segítségével. Ez egyaránt segít a hatékonyabb kódírásban és a kevesebb hibát tartalmazó programrészek készítésében, mivel a mechanizmus jelentősen csökkenti annak az esélyét, hogy a program memórialéket tartalmaz.
- Az alacsony szintű programozási felületek helyett a fejlesztők olyan objektumokkal dolgozhatnak, amelyek elfedik a mögöttük lévő API-k bonyolultságát. Ahelyett, hogy a fejlesztőknek minden apró-cseprő mögöttes Windows részlettel foglalkozniuk kellene, magasabb szintű absztrakciókat használhatnak, amelyekkel termelékenyebbek lesznek.
- A .NET magas szintű együttműködést kínál a COM és .NET objektumok között. A .NET kód nemcsak elérni tudja a COM objektumokat, de olyan objektumokat is készíteni lehet vele, amelyeket a COM világ tud fogyasztani.

A .NET-et *felügyelt környezet*nek nevezzük, a programozási nyelveit pedig *felügyelt nyelv*eknek – megkülönböztetve őket a *natív programozási nyelv*ektől, mint például a C, Object Pascal (Delphi) és a C++, amelyek közvetlenül CPU-specifikus utasításokra fordulnak.

> *Nem a .NET volt az első felügyelt kódú futásidejű környezet. Ez a címke a Java nyelvet illeti, amelyet a Sun Microsystems 1995-ben jelentett be. A .NET a Microsoft válasza volt a Java jelenségre, és sok képességét a Sun Java implementációja inspirálta.*

Nemcsak a nyelv, de a vele párban kibocsátott Visual Studio fejlesztőeszköz is jelentős szerepet játszott a .NET sikerében. A Visual Studio kb. egy tucatnyi projektsablonnal érkezett, amelyek gyors indulási lehetőséget biztosítottak a Windows alkalmazásfejlesztéshez. Ma a Visual Studio Ultimate változata már száznál is több projektsablont tartalmaz.

A Visual Studio projektsablonok jelentős hatékonysággal bírnak. Például az 1-6 ábrán látható alkalmazás néhány kattintással összerakható a „Windows Forms Application” sablon segítségével.



1-6 ábra: Egyszerű Windows Forms alkalmazás a Visual Studióval létrehozva

A Visual Studio eszközeinek a használatával könnyen kialakíthatod az 1-6 ábrán látható alkalmazást. Az egyetlen kódrészlet, amit kézzel kell az alkalmazáshoz adni az 1-3 kódlistában félkövér betűkkel kiemelt részlet.

1-3 kódlista: Az 1-6 ábra képernyője mögött lévő kód

```
using System;
using System.Windows.Forms;

namespace HelloWorldWinForms
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        private void button1_Click(object sender, EventArgs e)
        {
            label1.Text = "Hello from Windows Forms!";
        }
    }
}
```

Bár a .NET gazdag objektumkönyvtárakkal és nagyszerű eszközökkel volt felszerelve, az alsó rétegei még mindig olyan programozási felületeket használtak, amelyek az első Windows változatok tervezésekor jöttek létre. Ez olyan volt, mintha egy mai Lamborghinit egy az 1960-as évekből való motorral használnánk.

Az új felhasználói felület technológiák

Hosszú ideig a Windows felhasználói felületét a GDI (Graphics Device Interface) API kezelte, amely a Windows XP kibocsátásakor a GDI+ technológiára változott. A GDI és a GDI+ is raszter-bázisú technológiák voltak, és minden alapvető Windows vezérlő ezeket használta saját megjelenítéséhez. Ha egy fejlesztő meg szerette volna változtatni a vezérlők vizuális megjelenését, az egyetlen lehetősége annak a Windows eseménynek a felülírása volt, amelyik a vezérlő elemeinek kirajzolásáról gondoskodott.

A .NET 3.0-ban bevezetett WPF (Windows Presentation Foundation) grafikus alrendszer (ez később a Windows Vista beépített komponense is lett) teljesen átalakította a GDI paradigmát. A felhasználói felület imperatív megvalósítása helyett – vagyis egy adott programozási nyelven leírt utasítássorozat helyett – a WPF a XAML nyelvet, az XML egy leszármazottját használja a felület elemeinek és viselkedésének leírására. A WFP a GPU (Graphics Processing Unit) nagyszerű hardveres gyorsítási képességeit is kihasználja.

A Silverlight, a Microsoft gazdag webes élményt biztosító keretrendszere is a XAML-t használja a felhasználói felület leírására. Az 1-4 kódlista egy nagyon egyszerű XAML mintapéldát mutat, amely a „Hello, World” alkalmazást valósítja meg Silverlightban.

1-4 kódlista: A „Hello, World” leírása XAML-ben (Silverlight)

```
<UserControl x:Class="HelloFromSL.MainPage"
    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"
    d:DesignHeight="300" d:DesignWidth="400">

    <Grid x:Name="LayoutRoot" Background="White">
        <TextBlock FontSize="48">Hello from Silverlight</TextBlock>
    </Grid>
</UserControl>
```

Ez a kód az 1-7 ábrán látható megjelenést hozza létre. Az ábrán megjelenő szöveget az 1-4 kódlistában félkövér betűtípussal kiemelt részlet jeleníti meg.

1-7 ábra: Az 1-4 kódlista eredménye

A WPF és a Silverlight nagyszerű technológiák! Ezeket használni olyan, mintha az 1960-as évekből származó Lamborghini motort (GDI/GDI+ technológia) lecserélnénk egy modernre (WPF/Silverlight)! Ezek a technológiák nemcsak a felület megjelenését, de annak dinamikus viselkedését is definiálják – az esetek jelentős részében kód nélkül vagy minimális kóddal –, és összekapcsolják a felületet az alkalmazás logikai részével (modell) is. A tejesség igénye nélkül, itt egy lista ezek legfontosabb jellemzőiről:

- A WPF és Silverlight technológiák úgy lettek kialakítva, hogy elsőrendű felhasználói élményt kínáljanak asztali és webes alkalmazásokban egyaránt. Az egyszerű felhasználói elemek, mint például szövegdobozok, nyomógombok, lenyíló listák, képek stb. mellett teljes szabadságot adnak animációval és médiaelemekkel ellátott tartalom létrehozásához. A hagyományos – akár unalmasnak is mondható – szögletes felületi elemekkel szemben a WPF és Silverlight lehetővé teszi, hogy az alkalmazásod teljes felületét lecseréld.
- Ezek a technológiák rugalmas elrendezést kínálnak, amely lehetővé teszi, hogy a felület automatikusan olyan tényezőkhöz alkalmazkodhasson, mint például a képernyő mérete, felbontása, a megjelenített elemek száma, mérete, beállított nagyítás stb.
- A stílusok és sablonok olyan képességek, amelyek a szoros együttműködést teszik lehetővé a fejlesztők és a dizájnerek között. A fejlesztők az alkalmazás logikájával foglalkoznak, és a felhasználói felület vizuális tulajdonságait sohasem állítják közvetlenül. Ehelyett jelzik, ha a felhasználói felület állapota megváltozik. A dizájnerek hozzák létre a felület megjelenítését, figyelembe véve annak lehetséges állapotait.
- A Silverlight és a WPF adatkötést használ – ez olyan technológia, amely deklaratív módon kapcsolja össze a felület elemeit az alkalmazás adataival, illetve más felületi elemekkel. Az adatkötés együttműködik a stílusokkal, sablonokkal, sőt még az animációkkal is. Ez a mechanizmus különösen hasznos üzleti alkalmazások létrehozásánál. Az adatkötés segítségével az adatbázisból érkező és a logikai réteg által feldolgozott információk közvetlenül a felhasználói felület elemeihez köthetők, kód írása nélkül.

A WPF és Silverlight legnagyobb erénye valószínűleg az a tény, hogy lehetővé teszi a felhasználói felülethez tartozó feladatok szétválasztását a fejlesztőkhöz, illetve a dizájnerekhez tartozókra. A mai fogyasztó-központú alkalmazások készítésénél ez nyilvánvaló előny azokhoz a megközelítési módokhoz képest, ahol a fejlesztők és a dizájnerek feladatai keverednek.

A Windows alkalmazásfejlesztés 22-es csapdája

A Windows alkalmazásfejlesztés evolúciója során a technológiák és technikák elágaztak. Az egyik ág a natív alkalmazásfejlesztés, amely a C programozási nyelvvel indult a Windows korai napjaiban. A másik ág felügyelt kód alapú fejlesztés, amely a .NET keretrendszert használja a kapcsolódó nyelvekkel és technológiákkal együtt.

A CPU-specifikus utasításokra való fordításnak és a Windows API-hoz, valamint a rendszer erőforrásaihoz való alacsony szintű hozzáférésnek köszönhetően a natív alkalmazások olyan jó teljesítményt mutatnak, amelyet nem gyűrhetnek le a felügyelt kódot használó alkalmazások. Azonban ha üzleti alkalmazások írásáról van szó, a natív kód kevésbé termelékeny, és sokkal munkaigényesebb a használata, mint a .NET keretrendszerben használt felügyelt kódé. Mivel az üzleti alkalmazások általában adatbázisokkal dolgoznak, a felügyelt kódhoz tartozó alacsonyabb teljesítmény nem érzékelhető, mert az alkalmazás jelentős részét a szolgáltatásréteg és az adatbázis közötti kommunikációval tölti.

Yossarian, a bombázó pilótája érezhette magát hasonló helyzetben Joseph Heller regényében, A 22-es csapdájában, mint manapság a Windows fejlesztők, amikor egy adott alkalmazáshoz a megfelelő technológiát igyekeznek kiválasztani.

Rengeteg esetben, kifejezetten az asztali alkalmazások készítésénél nincs igazából optimális választás a natív és a felügyelt kód között. Bár a WPF és a Silverlight nagyszerű technológiák látványos alkalmazások készítéséhez, nem érhetők el egyszerűen natív kódból. A WPF és a Silverlight nincs elég mélyen az operációs rendszerbe ágyazva. Például amikor egy WPF alkalmazást elindítunk, időbe telik, amíg a WPF alrendszer betöltődik a memóriába. Rengeteg esetben, amikor számítási kapacitásra van szükség, a .NET gyengébb eredményt produkál, mint a natív kód.

A Visual Studio nagyszerű példája ennek a tudathasadásos helyzetnek. Vegyes kódbázist használ, a legtöbb komponens C++-ban készült, bizonyos részek pedig C#-ban, WPF-et használva. Az indulóképernyőnek azonnal meg kell jelennie, amint a felhasználó elindította az alkalmazást, ez C++-t használ GDI+-szal, mert a WPF nem volna ehhez elegendően gyors. A Visual Studio kódszerkesztője WPF-et használ, mert az megfelelően gazdag képességeket és eszközkészletet kínál ilyen összetett komponens befogadására – nagyszerű felhasználói felület biztosítása mellett.

Üzleti alkalmazások fejlesztése során a termelékenység az egyik legfontosabb tényező. A natív alkalmazások teljesítményelőnye nem ad valós értéket ezekhez, mert az esetek többségében az adatbázis réteg jelenti a szűk keresztmetszetet. Felügyelt kód használatával az üzleti alkalmazások programozási fázisa általában sokkal gyorsabb és hibáktól mentesebb.

Asztali alkalmazások készítésénél a natív kód optimális teljesítményt biztosít, de ebben az esetben nincs lehetőséged a WPF vagy Silverlight használatára. Ha felügyelt kódra térsz át, javul a hatékonyságod, de veszítesz az alkalmazás végső teljesítményéből. Végső elkeseredésedben öszvér megoldást is választhatsz – ahogyan azt a Visual Studio is teszi –, de ez rengeteg egyéb feladatot is jelent. Nem könnyű dönteni, igaz?

A Windows 8-cal ez a helyzet teljesen megváltozik. Webfejlesztők HTML5/CSS3/JavaScript tapasztalattal, edzett C++ programozók és .NET-en felnőtt fejlesztők mindannyian azt érezhetik, hogy tudásuk elegendő a Windows 8 stílusú alkalmazások fejlesztéséhez. Nincs kitüntetett fejlesztői tábor, mint például a C programozóké volt a Windows-korszak elején. Mindenki ugyanazokat az eszközöket és technológiákat használhatja – kompromisszumok nélkül.

Összegzés

A Windows platform elképesztő fejlődésen ment át életútjának elmúlt 27 évében. Egyszerű MS-DOS kiegészítésből komplex és minden képességgel bíró operációs rendszerré vált, amely a világ személyi számítógépeinek jelentős részén fut. A Windows-t információval dolgozó, illetve tapasztalt számítógép-felhasználókkal a fejben tervezték. Bár minden egyes újabb verzió egyre közelebb és közelebb került a kevesebb tudással rendelkező felhasználókhoz, soha nem vált az olyan egyszerű fogyasztói eszközök operációs rendszerévé, mint például az Apple iPhone-ja vagy iPadje.

A Windows 8 megváltoztatja ezt a helyzetet. A felhasználói számára már ismert hagyományos asztali üzemmód mellett az operációs rendszernek új megjelenést biztosítanak a Windows 8 stílusú alkalmazások, amelyek intuitív felhasználói élményt biztosítanak. Ezeknél az alkalmazás a teljes képernyőt, és így a felhasználó teljes figyelmét is birtokolja.

Nemcsak az operációs rendszer, de a fejlesztőeszközök és a programozási felület is rengeteget változott. Amíg a kezdetekben csak a C és C++ programozók kiváltsága volt a Windows alkalmazások készítése, addig ma már a C++, Visual Basic, C# és Delphi fejlesztők is írhatnak alkalmazásokat kedvenc eszközeikkel. A kiválasztott programozási nyelv és eszköz függvényében a fejlesztők különböző API-k és technológiák közül választhatnak. Amíg a natív nyelvek (C, C++ és Delphi) CPU-specifikus kódot fordítanak és jobb teljesítményt kínálnak, a felügyelt kód nagyobb termelékenységet biztosít az üzleti alkalmazásoknál, illetve azok használatba vehetik a modern felhasználói felület technológiákat, a WPF-et és a Silverlightot.

A Windows 8 stílusú alkalmazások ugyanazt az API-t kínálják a natív és a felügyelt kódú alkalmazások számára egyaránt. Sőt, ez az API elérhető a HTML5/CSS3/JavaScript fejlesztők számára is!

A Windows 8 nemcsak a felhasználói felületet cseréli le, de azt a módot is, ahogyan azt használjuk. A hagyományos beviteli eszközök mellett, mint például a billentyűzet és az egér, első osztályú élményt kínál az operációs rendszerrel és a Windows 8 alkalmazásokkal való kapcsolattartásra többpontos érintéssel, írópálcával, illetve giroszkópon vagy gyorsulásérzékelőn keresztül.

2. Bevezetés a Windows 8 használatába

Ebben a fejezetben az alábbi témákat ismerheted meg:

- Miért gondolta újra a Microsoft a Windows felületét, és miért hasznos ez?
- Hogyan használhatod hatékonyan az új felületi elemeket és a Windows Store alkalmazásokat?
- Hogyan futtathatsz tradicionális „asztali" alkalmazásokat Windows 8-on, és ezek hogyan viszonyulnak az újfajta alkalmazásokhoz?

A Windows 8 az első olyan operációs rendszer, melynek egyik fő célja, hogy két teljesen különböző világot kössön össze: a hagyományos asztali számítógépek és laptopok lassan fél évszázada ismert, fejlődő világát, valamint az előbbitől eddig teljesen elkülönülő, még gyerekcipőben járó táblagépek (tabletek) működését, energiatakarékosságát és alkalmazásmodelljét.

Mindezt pedig úgy kívánja nyújtani, hogy a felhasználó ne érezze magát korlátozva. Megkapja a hagyományos Windows operációs rendszerek rugalmasságát, használhat bármilyen korábbi Windowsra írt szoftvert, de emellett kihasználhatja a táblagépekre készített operációs rendszerek előnyeit is, mint például a természetes felhasználói felület (*Natural User Interface* – NUI). Vagy – megfordítva a dolgot – a Windows 8 teszi először lehetővé, hogy a felhasználóknak ne kelljen más-más operációs rendszereket használniuk akkor, ha éppen egy asztali gép előtt ülnek vagy egy táblagépet használnak.

Emellett a Windows 8 új szintre emeli a táblagépek működésének eddig csak mintegy melléktermékeként megjelenő „alkalmazás-szinergiát", vagyis azt, hogy az eszköz igazi képességeit nemcsak maguk az alkalmazások adják, hanem ezek együttműködése is. Ez az első olyan asztali gépekre is telepíthető operációs rendszer, melyet eleve úgy alakítottak ki, hogy az egyes alkalmazások természetes, a rendszerbe és a felhasználói élménybe illeszkedő módon működhessenek együtt.

Két cél, két felület, egy operációs rendszer

A hagyományos asztali számítógépek általánosan elterjedt adatbeviteli eszközeit senkinek sem kell bemutatni. Évtizedek óta billentyűzetet és egeret használunk erre a feladatra. Ezek az eszközök nagyon lassan változtak csak, hiszen tökéletesen szolgálják feladatukat.

Az elmúlt években azonban a hardverek mérete és fogyasztása (és ára) elérte azt a pontot, amikor megjelenhettek újfajta, a korábbi hordozható számítógépektől független kategóriát képviselő céleszközök, a *slate*-ek, illetve elterjedtebb nevükön *tablet*ek, táblagépek. Ezek olyan céleszközök, melyek tervezésénél elsődleges szempont volt, hogy kézben tarthatóak – könnyűek, vékonyak – legyenek, illetve ennek köszönhetően ne legyen szükség extra hardverre a vezérléshez, vagyis a felhasználó közvetlenebb, természetesebb módon manipulálhassa a képernyőn lévő objektumokat.

Az érintésre optimalizált operációs rendszer azonban jóval többet jelent, mint hogy a képernyő érintésre érzékeny. A felhasználói felületet eleve úgy kell kialakítani, hogy azt a felhasználó az ujjaival könnyedén kezelhesse. Például az ikonoknak, menüpontoknak elég nagyoknak kell lenniük. Emellett nemcsak az egér funkciójának kiváltása a cél, hanem az, hogy a felület építsen a természetes interakció előnyeire. Erre szolgálnak a többérintéses támogatásnál kihasználható ujjmozdulatok (*gestures*), mint például a két ujjal történő nagyítás vagy elforgatás.

Adott tehát két világ, két felülettel. Az egyik oldalon a megszokott, hatékony tartalom-létrehozásra kihegyezett egérközpontú irányítás, a másik oldalon a tartalomfogyasztásra alkalmas természetes felhasználói felület.

A Windows 8 szakít azzal a megoldással, hogy egy operációs rendszer csak az egyik fajta felülettel rendelkezhet, ezzel mintegy előre megszabva, mire lehet kényelmesen használni! Egy csomagban kapjuk meg a hagyományos asztalos-ablakos felületet annak minden előnyével, illetve az új stílusú, látványos Windows 8 felületet, mely ugyanolyan könnyen használható egérrel, mint az ujjainkkal.

Adatbevitel Windows 8-on

A Windows 8 használata érintésvezérléssel

A természetes felhasználói felület általában intuitív módon tanulható. Elég kezünkbe venni egy Windows 8-cal érkező táblagépet, és egyszerűen megérzéseinkre hagyatkozva néhány perc alatt megtanulhatjuk az új Start képernyő és a Windows Store alkalmazásainak alapvető használatát. Vannak azonban elsőre talán rejtettebb képességek, amelyeket a hatékony használat érdekében nem árt ismernünk. Ezeket általában valamilyen speciálisabb ujjmozdulattal hívhatjuk elő. Az alábbiakban a Windows 8-on használható ujjmozdulatokat tekintjük át röviden, egyelőre elméletben, megkönnyítendő a fejezet későbbi részeinek értelmezését.

- **Érintés** (*Tap*): Egy objektum egyszeri, rövid megérintése.
- **Nyomva tartás** (*Press and Hold*): Egy objektum megérintése után az ujj képernyőn tartása körülbelül egy másodpercig.
- **Elhúzás** (*Slide*): Egy lista vagy egyéb képernyőről kilógó objektum látható részének módosítása egy ujj képernyőn történő elhúzásával.
- **Lehúzás** (*Swipe*): Egy objektum kijelölése az objektum megérintése után az ujj lefelé húzásával.
- **Csípés** (*Pinch*): Kép vagy egyéb objektum kicsinyítése, nagyítása (*semantic zoom*) két ujjnak az érintőképernyőn való közelítésével vagy távolításával.
- **Forgatás** (*Rotate*): Kép vagy egyéb objektum forgatása két ujjnak az érintőképernyőn való forgatásával.
- **Szélről behúzás** (*Swipe from the edge*): Egy ujjnak a képernyő széléről való behúzása. Jellemzően kapcsolódó lehetőségeket (pl. *AppBar*) vagy rendszerszintű lehetőségeket (pl. *Charm bar*) aktivál.

Egyéb adatbeviteli eszközök Windows 8-on

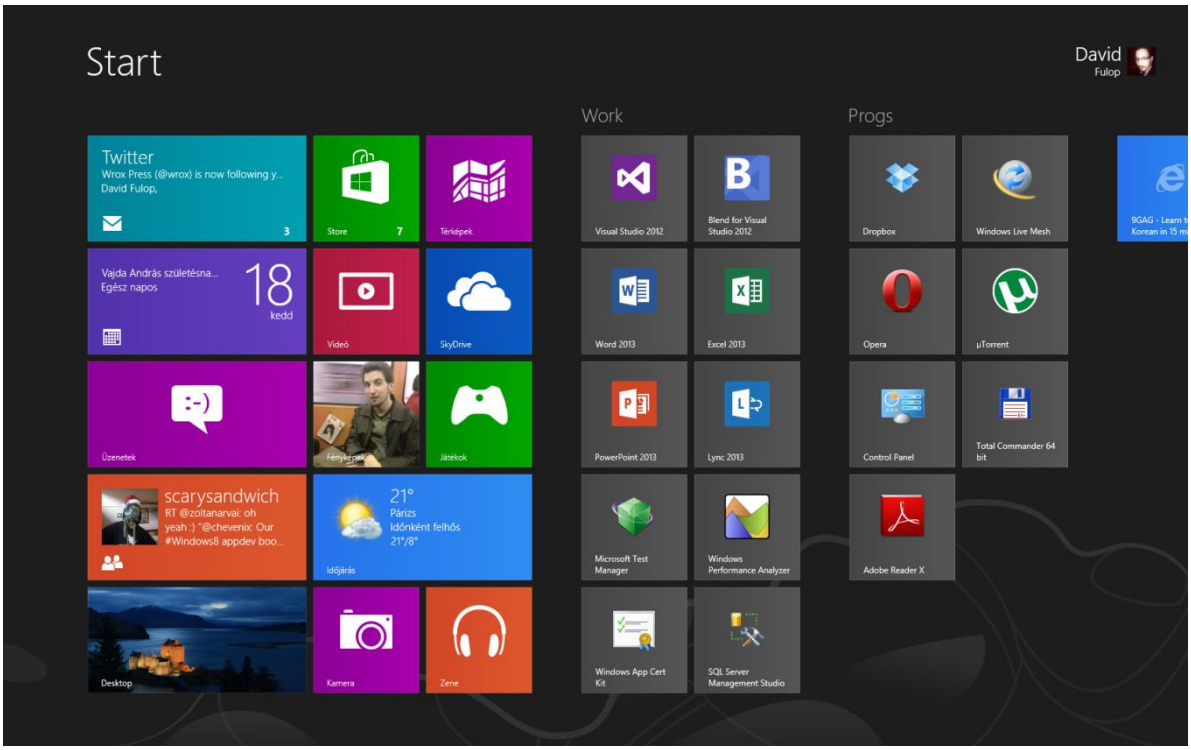
Mind felhasználóként, mind fejlesztőként érdemes figyelembe venni, hogy nemcsak a hagyományos értelemben vett interakciós eszközök (egér, billentyűzet, érintőképernyő) használhatók adatbevitelre, hanem a kevésbé előtérbe helyezettek is.

Ilyen például az íróvessző (*stylus*), mely a kézírás-felismeréssel együtt nagyban megkönnyítheti a szöveges adatbevitelt, és ilyenek például az érzékelők. A Windows Runtime – a Windows 8 programozási felülete – érzékelők egész sorához biztosít hozzáférést, ha az adott eszközt éppen ellátták valamelyikükkel: gyorsulásmérő, giroszkóp, magnetométer, környezeti fényérzékelő, illetve a helyzet-meghatározó rendszer. Egy olyan alkalmazásnál, mely akár táblagépeken is futhat, vagy azt kifejezetten táblagépre fejlesztették, elsődleges szempont lehet, hogy a felhasználó a lehető legkönnyebben tudja kezelni az alkalmazást akkor is, ha nincs a közelében hardveres billentyűzet és egér. Ugyan az érintőképernyő képes átvenni ezek helyét, sőt esetenként még kényelmesebb is lehet rajta keresztül végezni az interakciót, de érdemes kihasználni minden rendszer adta lehetőséget arra, hogy még könnyebbé tegyük a felhasználók dolgát. (Így nagyobb valószínűséggel költik a pénzüket az alkalmazásunkra.) Elég arra gondolni, hogy például mekkora előnyt jelent a beépített helyzet-meghatározó rendszer integrálása az alkalmazásunkba azzal szemben, mintha a felhasználónak magának kellene megadnia a tartózkodási címét.

A Start képernyő és az élő csempék

Talán a legszembeötlőbb változás és újdonság a Windows 8 felületével kapcsolatban, hogy eltűnt az „ikonikus” Start menü. Bejelentkezéskor teljesen új felület, a 2-1 ábrán látható Start képernyő (*Start Screen*) fogadja a felhasználót. Ez a Start menü utódának tekinthető: a korábbi Start menü által nyújtott

lehetőségek továbbra is mind elérhetők lesznek. De a Start képernyő nem pusztán teljes képernyőssé tett Start menü, annál sokkal többet nyújt!



2-1 ábra: A Start képernyő, a bal szélső csoportban élő csempékkel

Ahogy korábban is, a Start képernyőre kitűzhetjük a leggyakrabban használt programjainkat. Ezek azonban nemcsak statikus ikonokként tudnak megjelenni.

A Windows Store alkalmazásokhoz úgynevezett élő csempék (*Live Tiles*) tartoznak. Ha az alkalmazás támogatja, és nem tiltottuk le, az élő csempék folyamatosan információkat közölnek velünk a hozzájuk tartozó alkalmazással kapcsolatban. Nincs feltétlenül szükség az alkalmazások megnyitásához arra, hogy értesüljünk a minket érdeklő újdonságokról, elég, ha élő csempéjét rögzítjük a Start képernyőn. A hírolvasó alkalmazás csempéjén mindig a legfrissebb hír lesz olvasható, az időjárást jelző alkalmazásén a várható idő, a közösségi hálózatokat kezelő alkalmazásén pedig ismerőseink legutóbbi állapotfrissítései cserélődnek. Ilyen módon a Start képernyő nemcsak a gyakran használt alkalmazások listája lesz, hanem egy személyre szabott központi része a Windows nyújtotta felhasználói élménynek: elég egy pillantást vetni a Start képernyőre, és azonnal áttekinthetjük, hogy mi történt körülöttünk a világban, ismerőseinkkel, ahogy azt a 2-1 ábra is mutatja. Mindezt egy helyen anélkül, hogy több alkalmazást el kellett volna indítanunk. Ha pedig egy hírre, e-mailre, eseményre bővebben is kíváncsiak vagyunk, esetleg reagálni akarunk rá, elég megérinteni az élő csempét, és azonnal elindul a kapcsolódó alkalmazás.

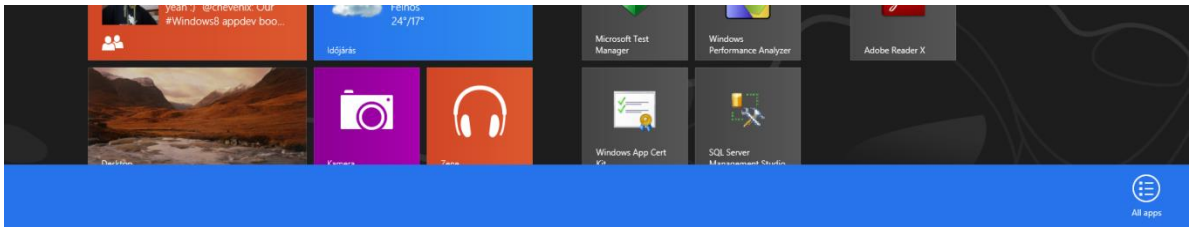
A csempék a Start képernyőn két-két csempe szélességű oszlopokban helyezkednek el – akárhány ilyen oszlopot létrehozhatunk, akár annyit is, hogy már ne férjenek el a kijelzőn. Ilyenkor a Start képernyő görgethető. Egeret használva egyszerűen az egérgörgő vagy a PageUp és PageDown gombok segítségével görgethetjük jobbra-balra, illetve a Start képernyő alján megjelenő görgetősávot is használhatjuk.

A csempék létrehozása

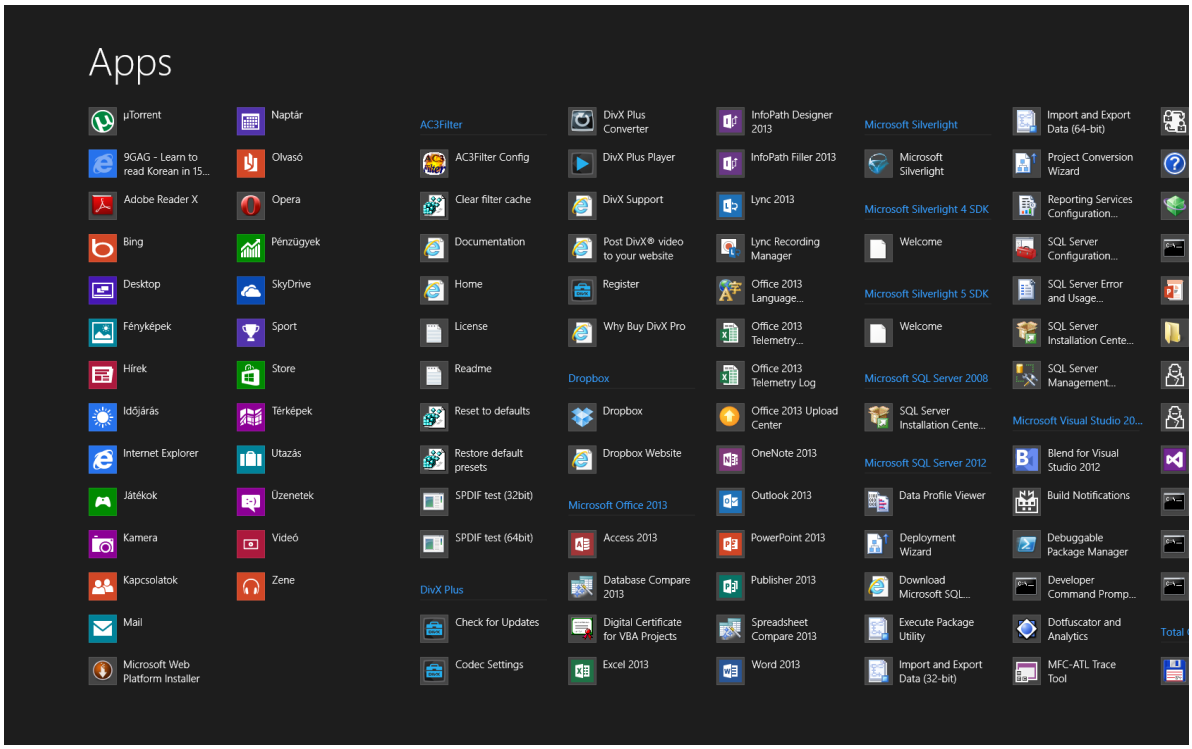
Csempét úgy tudunk létrehozni, hogy egy programot rögzítünk a Start képernyőn, vagy egy alkalmazást futtatva, azon az alkalmazáson keresztül rögzítjük.

Hogy áttekinthessük az összes telepített program listáját, a jobb egérgombbal kell kattintanunk a Start képernyő egy üres részére, vagy érintőképernyő esetén használjuk a képernyő széléről behúzás ujjmozdulatot a képernyő alján. Ekkor alul megjelenik az alkalmazássáv (*AppBar*), amelyen egyetlen gombot találunk, melynek felirata „All Apps” (Az összes alkalmazás), ahogy a 2-2 ábrán is látható. Ezt aktiválva a

Start képernyő átadja a helyét az összes telepített alkalmazást megjelenítő képernyőnek (*Apps*), ahogy a 2-3 ábra is szemlélteti.



2-2 ábra: A Start képernyő alkalmazássávja

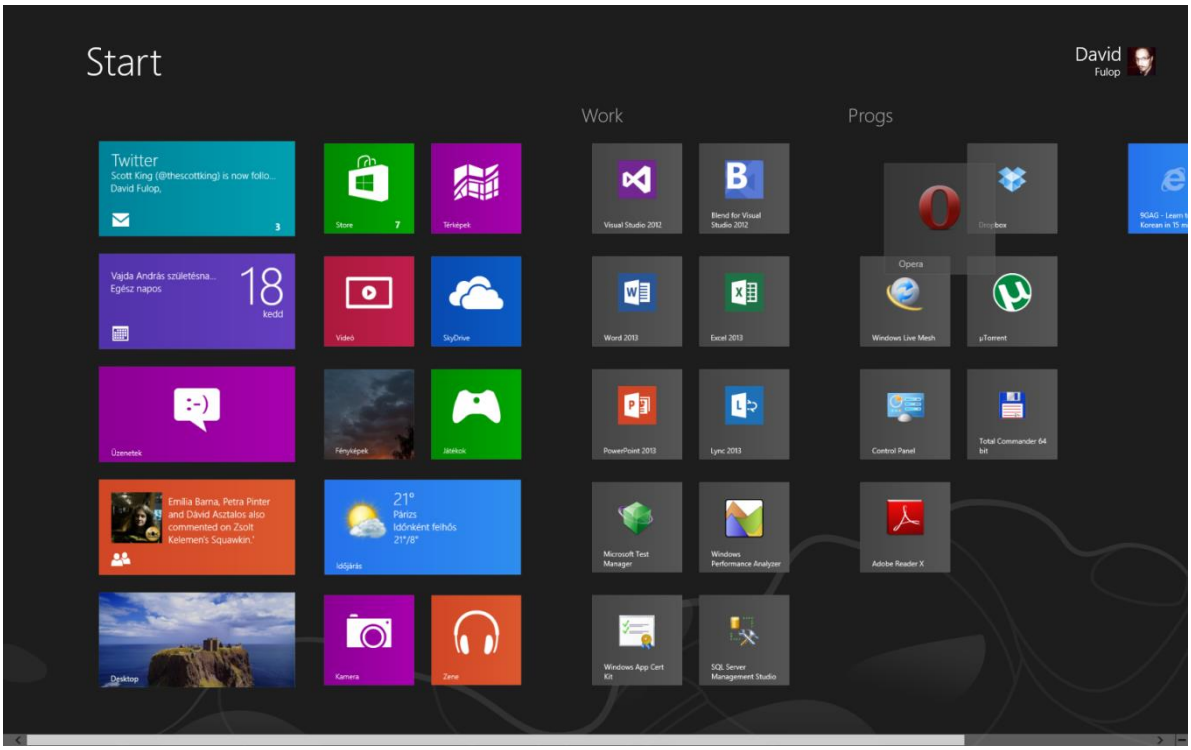


2-3 ábra: A telepített alkalmazások listája

Az Apps képernyőn jobb kattintással vagy a lehúzás (*swipe*) ujjmozdulattal kiválaszthatjuk az alkalmazást, amelynek csempéjét a Start képernyőn szeretnénk elhelyezni. Ekkor ismét az alkalmazássáv nyílik meg alul, és azon megtalálható a „Pin to Start" (Rögzítés a Start képernyőn) gomb – feltéve, hogy az alkalmazásnak még nincs ott csempéje.

A csempék átrendezése

A Start képernyő testreszabásának egyik módja, hogy a rajta rögzített csempéket – a hidegburkolóktól eltérően – könnyedén áthelyezhetjük. A csempék áthelyezésének módja teljesen intuitívnak mondható: az ismert fogd és vidd (*drag-and-drop*) technikát alkalmazhatjuk. Egér használata esetén a bal gombot nyomva tartva húzzuk el a csempét az eredeti helyéről. Érintőképernyőt használva annyi a különbség, hogy a csempét ujjunkkal lefelé húzva tudjuk kimozdítani helyéről. A többi csempe ilyenkor kisebb lesz, a háttérbe kerül, és ahogy az áthelyezendő csempét mozgatjuk az egérrel vagy ujjunkkal, „félreállnak az útjából", vagyis kiürítik a helyet, ahova a csempe kerülne, ha elereszténk. Ez a helyzet látható a 2-4 ábrán is. Amikor ténylegesen el is eresztjük a csempét, akkor rögzül új pozíciójában.



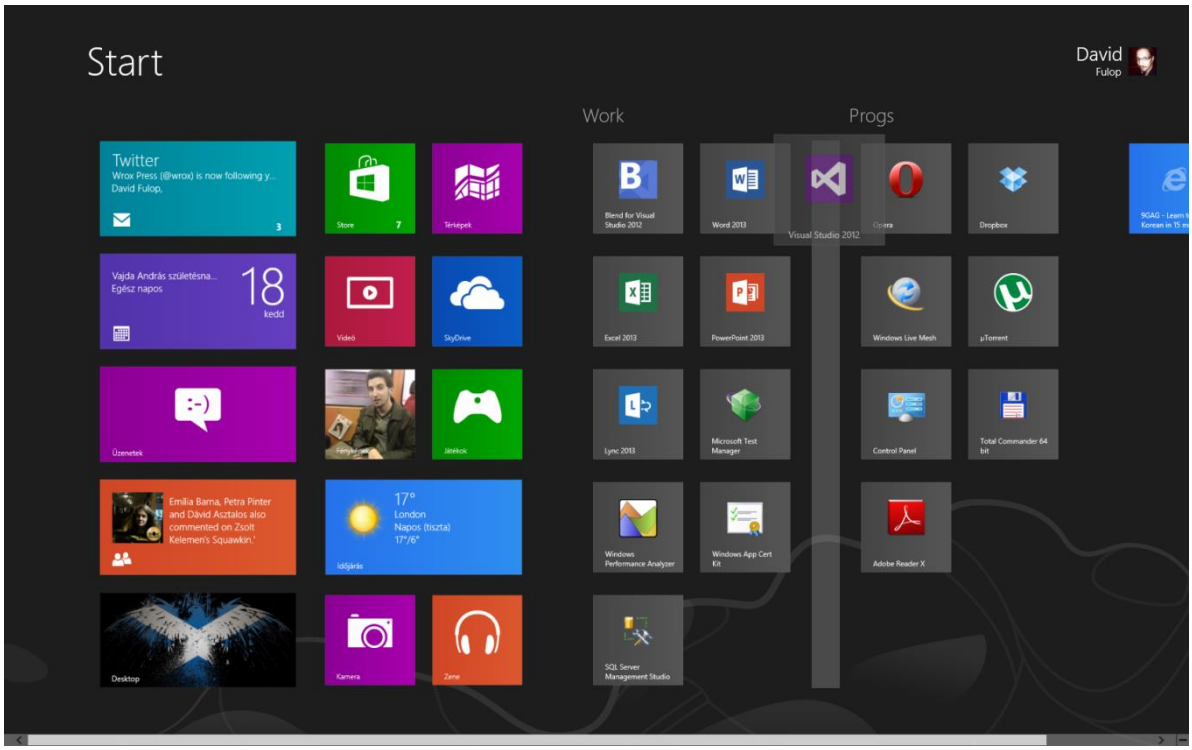
2-4 ábra: A Start képernyő csempeáthelyezés közben

A csempék csoportosítása

Bár a Start képernyőn elsősorban csak a gyakran használt alkalmazásainkat rögzítjük, így is előfordulhat, hogy szeretnénk átláthatóbbá tenni a több tucat csempéből álló képernyőt. E célból a csempéket akár csoportokba is rendezhetjük, és ezeknek a csoportoknak nevet is adhatunk. Lehetőségünk nyílik még gyorsan, a csoportok között navigálva ugrálni a Start képernyőn, így akkor is gyorsan eltalálunk a keresett csempéhez, ha egyébként több képernyőnyit kellene átgörgetnünk.

A csoportba rendezés is a fogd és vidd módszerrel történik: ha egy csempét egy másik csoportba húzunk, onnantól annak a csoportnak lesz eleme. Ha új csoportot szeretnénk létrehozni, akkor egy csempét két csoport közé (vagy a legelső elé, illetve legutolsó mögé) kell vonszolni, amíg meg nem jelenik alatta egy ujjnyi vastag világosszürke csík, ahogy a 2-5 ábrán is látható. Ha ilyenkor engedjük el, a csempe egy új csoport első eleme lesz.

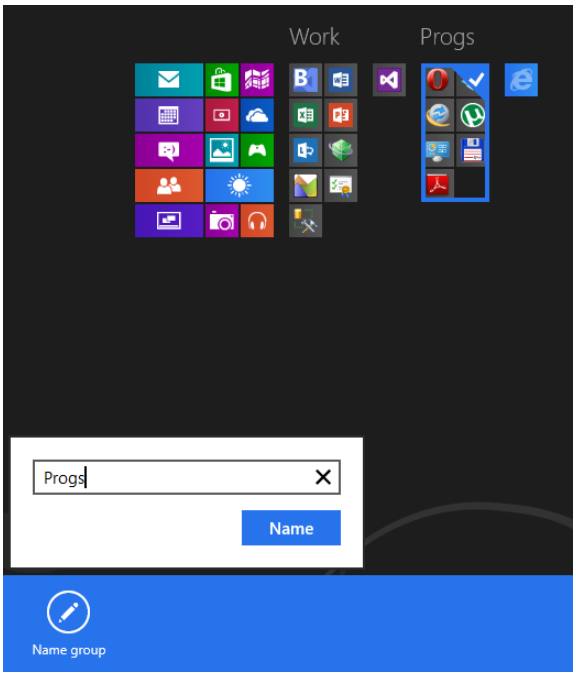
A csoportok sorrendjének megváltoztatására és a csoportok el- vagy átnevezésére normál nézetben nincs lehetőség, de többek között erre is szolgál az áttekintő vagy „madártávlat” nézet.



2-5 ábra: Új csempecsoport létrehozása

Ha a Start képernyőn a Ctrl nyomva tartása mellett húzzuk lefelé az egérgörgőt, vagy érintőképernyős eszköznél ha összecsippentjük két ujjunkat a képernyőn, az áttekintő (lekicsinyített) nézetbe jutunk. Alternatívaként a Start képernyő jobb alsó sarkában található gombbal válthatunk a normál és a áttekintő nézetek között. Itt madártávlatból vehetjük szemügyre a csempéket, és jobban szembetűnnek a csoportok. Egy csoportra kattintva visszakerülünk a normál nézetbe, ezzel tehát gyorsan ugrálhatunk a Start képernyő szélei között, ha nem akarunk görgetni.

A csoportok átrendezéséhez mindössze annyit kell tennünk, mint amit egy-egy csempe áthelyezésénél tennénk. Fogd és vidd módszerrel arrébb vonszolhatunk egy csoportot, és elereszthetjük ott, ahova helyezni szerettük volna.



2-6 ábra: Egy csempecsoport nevének megváltoztatása az áttekintő nézetben

De ebben a nézetben ki is jelölhetjük a csempecsoportokat jobb kattintással vagy a lehúzás ujjmozdulattal. Ha egy csempecsoportot kijelöltünk, a képernyő alján megjelenő menüben a „Name group” gombot aktiválva nevet adhatunk a csoportnak, vagy átírhatjuk meglévő nevét, ahogy azt a 2-6 ábra is szemlélteti.

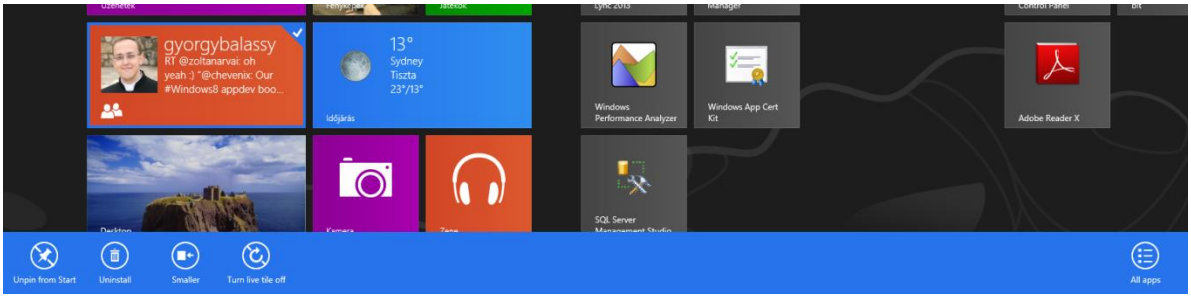
A normál nézethez való visszatéréshez elég, ha az egyik csoportra kattintunk, vagy az egér görgőjét, illetve a csípés ujjmozdulatot fordítva használjuk, vagy a jobb alsó sarokban lévő nézetváltó gombra kattinthatunk.

Műveletek a csempékkel

Csempéink tehát rendezgethetők, csoportosíthatók, és ha a mögöttük lévő alkalmazás támogatja, folyamatosan információkkal látnak el minket az alkalmazás futtatása nélkül is. Amennyiben rákattintunk egy csempére (vagy érintőképernyő használata esetén megérintjük ujjunkkal), azzal elindítjuk a mögötte lévő programot. De természetesen itt nem érnek véget a lehetőségeink!

Ha kijelölünk egy élő csempét, a 2-7 ábrán látható alkalmazássáv jelenik meg a képernyő alján, bár nem mindegyik gomb lesz elérhető minden alkalmazásnál. A gombok és funkcióik a következők:

- **Unpin from Start**: Eltünteti az élő csempét a Start képernyőről.
- **Uninstall**: Eltávolítja az alkalmazást a PC-ről.
- **Larger/Smaller**: Dupla szélességűvé teszi az élő csempét, vagy dupla szélességű esetén normál méretűvé változtatja azt. Ez nem egyszerűen felnagyítja a csempét: a széles csempén az alkalmazás több információt képes megjeleníteni.
- **Turn Live Tile off/Turn Live Tile on**: Kikapcsolja, illetve bekapcsolja az élő csempe frissítését.



2-7 ábra: Egy élő csempe alkalmazássávja

A nem Windows Store-alkalmazások (ide tartoznak például a .NET-es és Win32-es natív programok) is rendelkezhetnek csempékkel, azonban ezek nem „élő” csempék, nem jeleníthetnek meg frissülő tartalmat. Ennek leginkább technikai okai vannak: ezen alkalmazások fejlesztésekor még nem állt rendelkezésre az élő csempékhez szükséges API és infrastruktúra. Ha egy nem Windows Store alkalmazás csempéjét jelöljük ki, az alábbi lehetőségek lesznek elérhetők az alul megjelenő alkalmazássávon:

- **Unpin from Start**: Eltünteti az élő csempét a Start képernyőről.
- **Pin to Taskbar/Unpin from Taskbar**: A program indító ikonját a tálcán rögzíti, vagy eltávolítja azt.
- **Uninstall**: Megnyitja a „Programs and Features” ablakot, amelynek segítségével eltávolíthatjuk az alkalmazást a gépről.
- **Open new window**: Új ablakot nyit a programból (ha az már fut).
- **Run as Administrator**: rendszergazdai jogosultságokkal indítja el a programot. Természetesen ehhez a felhasználónak rendszergazdának kell lennie.
- **Open file location**: A Windows Explorerben megnyitja a csempéhez tartozó programot tartalmazó mappát.

De nemcsak alkalmazásokat tűzhetünk ki a Start képernyőre, hanem bármilyen fájlt vagy akár linket, amit az adott típusú erőforrást kezelő alkalmazás engedélyez. Ezek alkalmazássávján csak az „Unpin from Start” parancs érhető el.

A Windows Store alkalmazások használata

A Windows Store-on keresztül terjesztett újfajta alkalmazások sokban különböznek az eddig megszokott Windows programoktól. Fontos, hogy megismerkedjünk ezekkel a különbségekkel, és ne próbáljunk minél inkább a „régi” alkalmazásokra hasonlítókat létrehozni. A felhasználók egy Windows Store alkalmazástól nem azt fogják elvárni, hogy a régi ablakos alkalmazások közé illeszkedjen be, hanem hogy minél többet megragadjon az új, modern tervezési elvekből, és ezáltal lehetőleg minél intuitívabb legyen a kezelése.

Alkalmazások indítása és bezárása

Egy Windows Store alkalmazást egyszerűen a Start képernyőre rögzített csempéjére kattintással vagy annak megérintésével indíthatunk el. Egyes alkalmazások több (másodlagos) élő csempe létrehozását és Start képernyőre rögzítését is támogatják. Például egy tőzsdei árfolyamokat figyelő alkalmazás lehetőséget adhat arra, hogy egy-egy cég részvényárfolyamára külön élő csempét hozzunk létre, és így ezeket az árfolyamokat anélkül is követhessük, hogy meg kellene nyitnunk az alkalmazást.

A Windows Store alkalmazások a korábbi Windows alkalmazásoktól eltérően nem ablakban jelennek meg, hanem a kijelző minden egyes képpontját a fejlesztő rendelkezésére bocsátják, hogy maximalizálhassa a megjeleníthető tartalmat, és növelje az érintőfelületet. Ez az irányelv többek között azt is magával hozza, hogy a Windows Store alkalmazások mindig teljes képernyősek: olyannyira, hogy nincs megszokott keretük, sem pedig címsoruk. (Később bemutatjuk, hogy ennél árnyaltabb a kép, és nemcsak ténylegesen teljes képernyős lehet egy Windows Store alkalmazás.) Ennek megfelelően nincs olyan gomb sem az alkalmazás jobb felső sarkában, amellyel bezárhatnánk az alkalmazást, vagy ikon állapotúra kicsinyíthetnénk azt, hogy más alkalmazásokat használjunk.

Ehelyett egy Windows Store alkalmazást úgy zárhatunk be, ha az alkalmazás felső részét fogd és vidd módszerrel lehúzzuk a képernyő aljára. Vagyis egér használata esetén a kurzort a képernyő tetejére mozgatjuk (ilyenkor egy kis kézzé változik), majd a bal egérgombot lenyomva tartva a képernyő aljáig húzzuk, és ott felengedjük. Érintőképernyő használata esetén gyakorlatilag csak annyi a dolgunk, hogy ujjunkat a képernyő felső szélétől (a legfelső képponttól kell, tehát érdemes a képernyőn kívülről) a képernyő aljáig húzzuk. Ahogy az alkalmazás ablakát elkezdjük lefelé húzni, az alkalmazás képe összemegy jelezve, hogy tovább mozgatva lefelé az egeret bezárhatjuk az alkalmazást.

Ha bezárunk egy Windows Store alkalmazást, a Start képernyőre jutunk vissza.

Annak ellenére, hogy ez meglehetősen szokatlan annak a felhasználónak, aki először használ Windows Store alkalmazást, nem javasolt, hogy a fejlesztők manuálisan elhelyezzenek egy-egy „X” gombot az alkalmazások jobb felső sarkában. A felhasználók az imént vázolt viselkedést várják el egy Windows Store alkalmazástól, legfeljebb nagyon marginális esetekben igénylik egy „kikapcsoló gomb” elhelyezését.

Váltás a Windows Store alkalmazások között

Még a tartalomfogyasztásra optimalizált táblagépeken is ritka, hogy egy felhasználó csak egy alkalmazást akarna egyszerre használni. Attól, hogy valaki éppen az internetet böngészi, még szüksége lehet arra, hogy egy levelező alkalmazás a háttérben figyelje, érkezett-e új levele, és egy ilyen esemény bekövetkeztéről természetesen értesítse is a felhasználót.

Ahogy korábban írtuk, a tálcára kicsinyítés sem érhető el a Windows Store alkalmazásoknál – már csak azért sem, mert nincs tálca, amin megjelenhetne az ikon állapotúra kicsinyített alkalmazás. Ámde mégiscsak szükség van arra, hogy egy alkalmazást elindíthassunk és használhassunk anélkül, hogy az előzőleg használtat be kellene zárnunk.

Alkalmazás indítására elsősorban a Start képernyő szolgál, így tehát ahhoz kell visszajutni anélkül, hogy bezárnánk a futó alkalmazást. Erre több módszer is kínálkozik. Ha van Windows billentyű egy PC-hez kapcsolt billentyűzeten vagy egy táblagépen, akkor ennek megnyomásával mindig a Start képernyő és az aktuális program között válthatunk. Egy másik lehetőség, hogy az egérkurzort a képernyő bal alsó sarkába mozgatjuk: ekkor megjelenik a Start képernyő kicsinyített képe, és ha arra kattintunk, vissza is kerülünk oda.

A harmadik lehetőség az ún „Charm bar” közepén található Windows gomb használata. Ez elsősorban a táblagépek felhasználóinak lesz hasznos: ha ujjunkat az érintőképernyő jobb széléről húzzuk be, a jobb

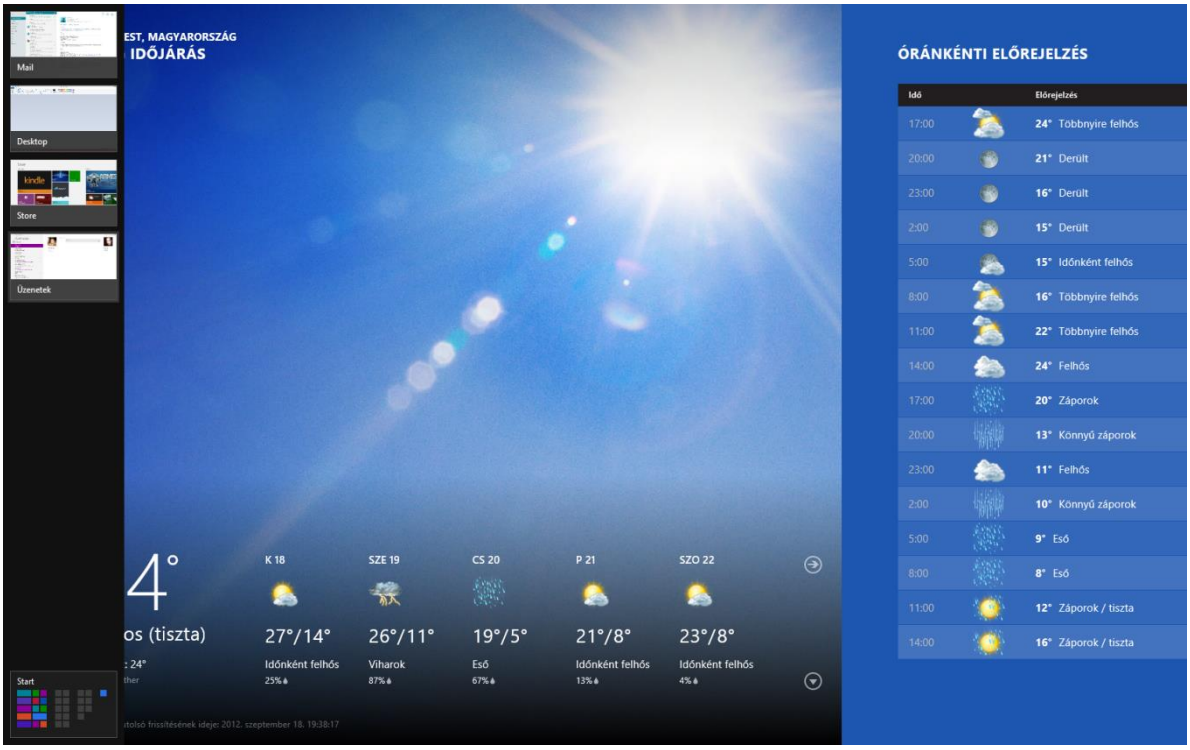
oldalon megjelenik a Charm bar, melyről később még bővebben is szót ejtünk. Ennek középső gombja egy Windows logo, melyet megérintve ismét a Start képernyőre kerülünk.

> *A könyv írásakor még nem állt rendelkezésünkre a Windows 8 magyar nyelvű terminológiája, amely kialakulása jelenleg is folyamatban van. Egyelőre még nincs jó fordításunk a „Charm bar" fogalmára.*

Ha visszatérünk a Start képernyőre, indíthatunk másik alkalmazást, és az imént használt alkalmazás is tovább fut. Munkánkat gyorsan ott folytathatjuk, ahol abbahagytuk. Persze ehhez az is szükséges, hogy valahonnan ismét előhívhassuk a már futó alkalmazást. Erre szolgál a képernyő bal oldalán előhívható „backstack", a futó Windows Store alkalmazások listája.

Bármely alkalmazás futtatása közben – vagy akár a Start képernyőn is – ha az egérkurzort a képernyő egyik bal sarkába mozgatjuk, majd a másik bal sarok irányába mozdítjuk el (ez sokkal egyszerűbb a gyakorlatban, mint amilyennek leírva tűnhet), a képernyő bal szélén megjelennek a futó Windows Store alkalmazások kicsinyített képei, ahogy a 2-8 ábrán látható. Ilyenkor csak rá kell kattintani a megnyitni kívánt alkalmazásra, és folytathatjuk is a munkát.

Ha egyszerűen az előzőleg futtatott alkalmazáshoz akarunk visszatérni, elég, ha a bal felső sarokba húzzuk az egérkurzort, és a megjelenő képre – amely az előző alkalmazás képe – kattintunk.



2-8 ábra: A futó alkalmazások oldalsávja

Érintőképernyő használata esetén kicsit más a helyzet, de ugyanezt a listát használhatjuk. Ha ujjunkat a képernyő bal széléről elkezdjük befelé húzni, megjelenik az aktuálisan futó alkalmazás előtt futtatott (és be nem zárt) alkalmazás kicsinyített képe. Ha ezt ujjunkkal továbbhúzzuk, és a képernyő közepe táján eleresztjük, az aktuálisan futó alkalmazás a háttérbe kerül, és átadja helyét annak, amelyet éppen behúzunk a képernyőre. Így tehát nagyon kényelmesen lehet váltogatni a futó alkalmazások között, egyszerűen csak bal hüvelykujjunkkal kell befelé „lökdösni" őket.

Ha ujjunkat nem húzzuk tovább, amikor megjelent alatta az előző alkalmazás, hanem ehelyett visszafelé húzzuk, ki a képernyőről, akkor megjelenik a korábban leírt lista, amelyen az összes éppen háttérben lévő alkalmazást szemügyre vehetjük, és egy érintéssel átválthatunk valamelyikre.

Több Windows Store alkalmazás egyidejű futtatása

A Windows Store alkalmazások keret nélkülisége magával vonzza azt is, hogy nem tudjuk átméretezni az alkalmazásokat. Így viszont egyszerre csak egy lehet a képernyőn, ami nagyban megnehezítené azoknak az eseteknek a kezelését, amikor ténylegesen két alkalmazást akar egyszerre használni valaki. Elég elképzelni egy olyan egyszerű szituációt, mint amikor mondjuk egy ismerősünkkel egy azonnali üzenetküldő alkalmazáson keresztül kommunikálva egyazon weboldalról vagy dokumentumról beszélgetünk. Meglehetősen kényelmetlen lenne, ha a dokumentum-nézegető alkalmazásból folyton-folyvást át kellene váltani az üzenetküldőbe, és egyszerre csak az egyik látszana.

Éppen ezért a Windows 8 valójában három méretben tudja megjeleníteni Windows Store alkalmazásainkat: a már ismert teljes képernyős (*full screen*) módon kívül az alkalmazás lehet még kisméretű (*snapped*) és nagyméretű (*filled*).

Kisméretű, vagyis „snapped" módban az alkalmazás a képernyő egyik szélén jelenik meg egy 320 képpont széles sávban; vertikálisan továbbra is teljesen kitölti a képernyőt. Nagyméretű, vagyis „filled" módban pedig a maradék helyet tölti ki, vagyis a képernyő teljes szélességét, kivéve az egyik oldalra rögzített, kisméretű alkalmazás által elfoglalt 320 képpontot. Alkalmazásaink fejlesztésekor külön-külön meghatározhatjuk, hogy milyen elrendezésű legyen az alkalmazás felülete a három módban.

Fontos azonban megjegyezni, hogy a Windows 8 minimális képernyőfelbontás-igénye 1024*768 képpont, vagyis ekkora felbontáson is elindul az operációs rendszer – viszont a fenti két mód, vagyis a kisméretű és a nagyméretű megjelenítés támogatása csak 1366*768 képpont felett támogatott. Tehát előfordulhat, hogy egyes felhasználók nem tudják kis- vagy nagyméretű módban használni alkalmazásunkat, csakis teljes képernyősen. Továbbá ez csak fekvő módban támogatott, vagyis amikor a képernyő hosszabbik oldala van vízszintesen.

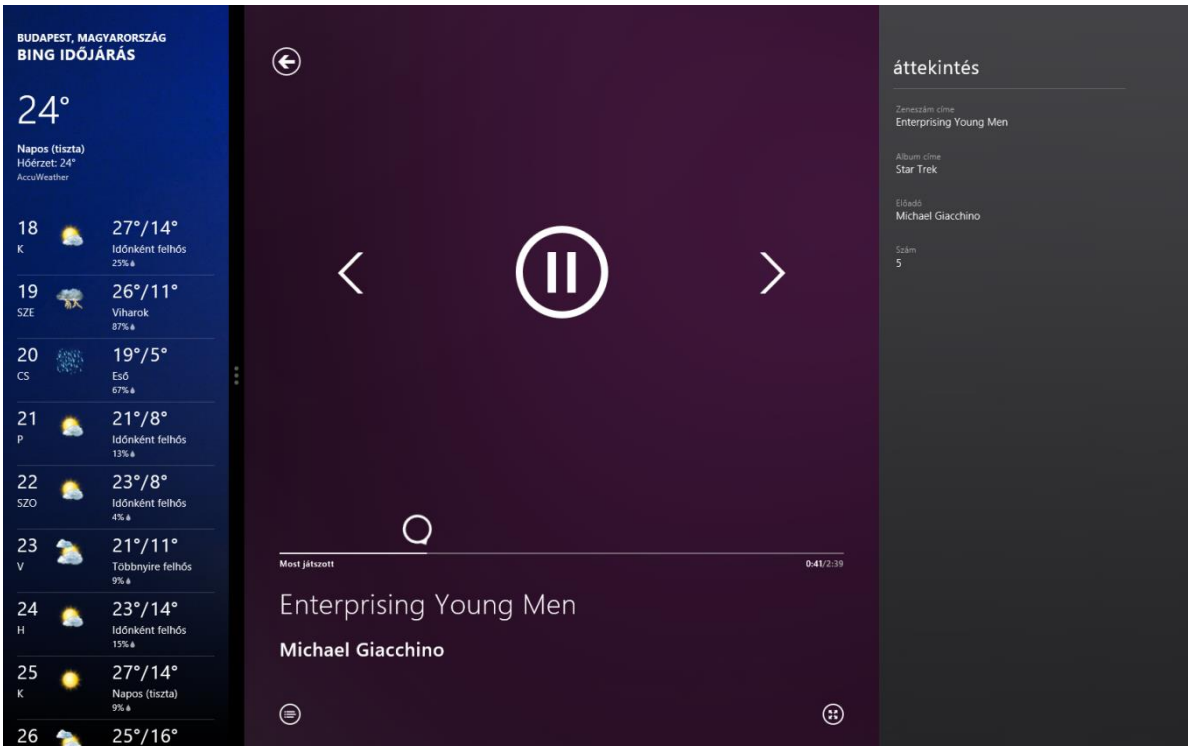
Az alkalmazások kisméretűvé váltására ismét egy egér- vagy ujjmozdulatot használhatunk. Egérrel vagy ujjunkkal ragadjuk meg az alkalmazás felső részét, mint amikor be akartuk zárni, és húzzuk lefelé addig, amíg a kicsinyített képe meg nem jelenik– pontosan úgy, mint amikor egy ujj- vagy egérmozdulattal bezártuk! Ezúttal azonban ne lefelé húzzuk, hanem a képernyő egyik oldala felé! Amikor ujjunk vagy az egérkurzor és az alatta vonszolt kép elérte a kisméretű állapothoz tartozó térrészt (vagyis a képernyő szélétől számított 320 képpontnyi távolságot), a vonszolt kép alatt megjelenik egy függőleges elválasztóvonal. Ha ilyenkor elengedjük az alkalmazást, kisméretű, „snapped" állapotba vált a képernyő kijelölt szélén.

Amikor egy alkalmazás kisméretű állapotban van a képernyő egyik szélén, a háttérbe került alkalmazások listájának egyik elemét kiválasztva az előtérbe kerülő alkalmazás nem teljes képernyős módban fut tovább, hanem nagyméretű („filled") állapotba kerül, ahogy azt a 2-9 ábra is szemlélteti.

Az egyszerre látszó kisméretű és nagyméretű alkalmazást egy keskeny fekete sáv választja el egymástól. Ennek segítségével válthatunk, hogy mely alkalmazás fusson kis méretben, és melyik nagyban. Egyszerűen csak meg kell ragadni a sávot a nyomva tartás ujjmozdulattal vagy a bal egérgomb lenyomásával, amikor a kurzor felette áll, majd elhúzni a nagyméretűként megjelenő alkalmazás irányában. Amikor ujjunk vagy a kurzor a képernyő másik oldalához közeledik, a nagyméretű alkalmazás kisméretűvé válik. Ilyenkor fel kell engedni ujjunkat az érintőképernyőről, vagy a bal egérgombról, és az elválasztóvonal átkerül a másik oldalra, az eddig kisméretű alkalmazás pedig nagyméretű megjelenítésre vált.

Ha az elválasztó vonalat nem a nagy, hanem a kisméretű alkalmazás felé húzzuk, majd elengedjük, a kisméretű alkalmazás a háttérbe kerül, az eddig nagyméretű pedig teljes képernyős megjelenítésre vált.

Az itt bemutatott módon tehát lehetséges két Windows Store alkalmazás egyidejű megjelenítése is a képernyőn. Lehet, hogy ez nem tűnik soknak, elvégre nem ritka, hogy tucatnyi alkalmazást futtattunk egyszerre eddig a Windows-on. Ne feledjük el azonban, hogy a Windows Store alkalmazásoknak nemcsak az asztali számítógépeken kell használhatóknak lenniük, hanem a teljesen más célú és megjelenésű táblagépeken is. Ha táblagépen szeretnénk kettőnél több alkalmazást egyidejűleg használni, elképzelhető, hogy nagyon kevés helyet adna egy-egy alkalmazásnak ahhoz, hogy azt kezelni tudjuk, főleg érintésvezérléssel, amely az egérnél kevésbé precíz, és éppen emiatt nagyobb vizuális objektumokat igényel.



2-9 ábra: A futó alkalmazások oldalsávja

A Charm bar

Ahogy azt már többször is kiemeltük, a Windows 8 az első olyan operációs rendszer, mely az egér- és billentyűzetvezérelt, ablakos alkalmazások világa és az elsősorban táblagépekre jellemző érintésvezérelt, tartalomfogyasztásra kiélezett világ közötti szakadékot próbálja áthidalni – egyetlen rendszerben egyesítve a két stílus előnyös sajátosságait. Ennek egyik folyománya viszont az is, hogy a táblagépeknél bonyolultabb asztali operációs rendszer fontos, gyakran használt funkcióit egyszerűbben elérhetővé kell tenni. Senki sem akar menük és ablakok mélyén turkálni azért, hogy felcsatlakozzon egy WiFi hálózatra vagy átállítsa a képernyő fényerejét! Sőt, még kevésbé, ha mindezt egér és billentyűzet nélkül kell tennie.

Ebből kifolyólag a Windows 8 tervezői úgy döntöttek, hogy a gyakran használt – rendszerszintűnek nevezhető – funkciókat egy közös, mindig egyszerűen elérhető helyen fogják össze. Ez lett a Charm bar, melyet a képernyő jobb oldalán jeleníthetünk meg.

A Charm bar megjelenítéséhez ugyanazt kell tennünk az egérrel, mint amit a futó Windows Store alkalmazások listájának megjelenítéséhez tettünk – csak nem a képernyő bal, hanem a jobb sarkainak egyikén. Vagyis mozgassuk az egeret a képernyő egyik jobb oldali sarkába, majd mozdítsuk el a másik jobb oldali sarok felé! (Tulajdonképpen elég az is, ha csak a sarokba mozgatjuk.) Érintőképernyő esetén használjuk a képernyő széléről behúzás ujjmozdulatot a képernyő jobb szélén!

Ekkor megjelenik a Charms bar öt ikonja, illetve a képernyő másik felén egy információs mező, ami az aktuális dátumot, időt, illetve a hálózati kapcsolatot és az akkumulátor (ha van) töltöttségét jelzi, ahogyan azt a 2-10 ábra is mutatja.

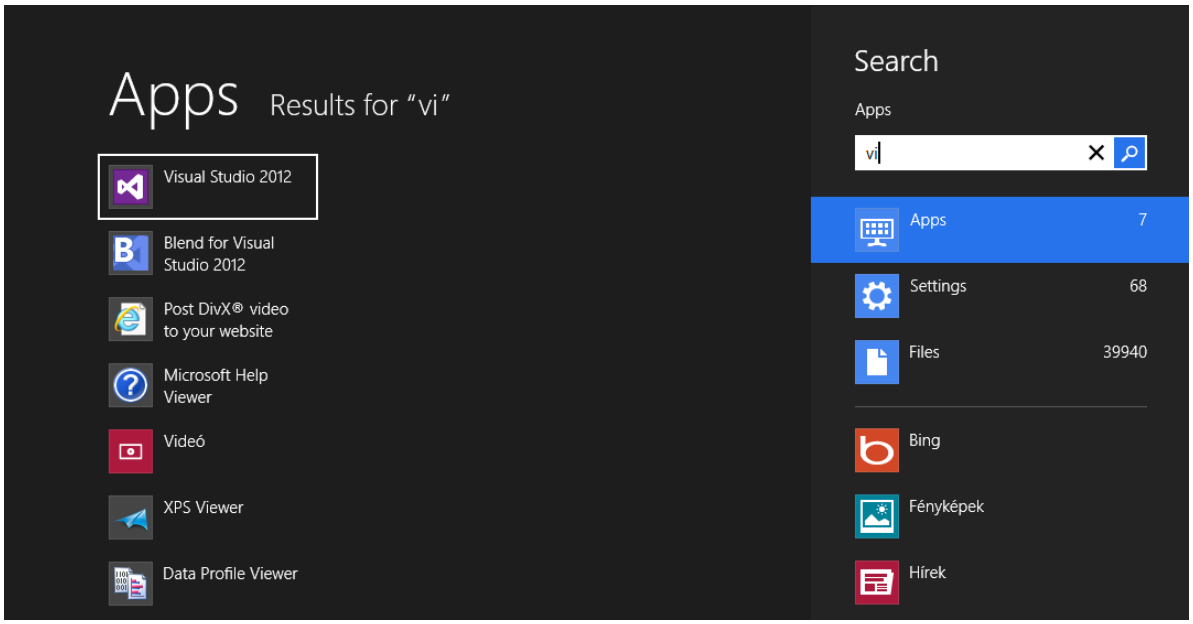
Az öt ikon közül a középső a Start gomb, amiről korábban már szó esett. Erre kattintva, illetve ezt megérintve a Start képernyőre kerülünk. De ezenkívül még négy másik gomb is található a Charm baron. Ezek fentről lefelé: *Search*, vagyis Keresés, *Share*, vagyis Megosztás, (ez után következik a Start, majd) *Devices*, azaz Eszközök és *Settings*, vagyis Beállítások.

2-10 ábra: A Start képernyő nyitott Charm barral

Keresés

A Search gombra kattintva vagy azt megérintve a képernyőn a telepített alkalmazások listája jelenik meg, illetve a képernyő jobb oldalán a keresőablak. Ennek tetején egy szövegdobozba rögtön begépelhetjük, hogy mire is szeretnénk rákeresni. Alapesetben a telepített alkalmazások között keresünk, de a kereső doboz alatti Settings, illetve Files gombokkal ezt megváltoztathatjuk, ha például fájlokat, dokumentumokat szeretnénk keresni a gépen.

Ahogy gépelünk tovább, a képernyő tartalma az alapján szűrődik, hogy mit írtunk be a kereső dobozba, ahogyan a 2-11 ábrán is látható.



2-11 ábra: A kereső képernyő a „vi" karaktersor begépelése után

A három nagy kategória (Apps, Settings, Files) alatt további ikonok jelennek meg. Ezek azon alkalmazások ikonjai, amelyek jelezték a Windows felé, hogy rendelkeznek saját keresési képességgel. Ilyen

alkalmazásokat mi is készíthetünk. Ha kijelöljük egy alkalmazás ikonját ebben az oldalsávban, akkor az alkalmazás aktiválódik, és az alapértelmezett kereső helyett a kiválasztott alkalmazás állítja össze a képernyő többi részét kitevő találatlistát. Ezenfelül egy alkalmazásnak még arra is lehetősége van, hogy javaslatokat jelenítsen meg a keresődoboz alatt, ahol egyébként a korábbi keresések kulcsszavai is megjelennek.

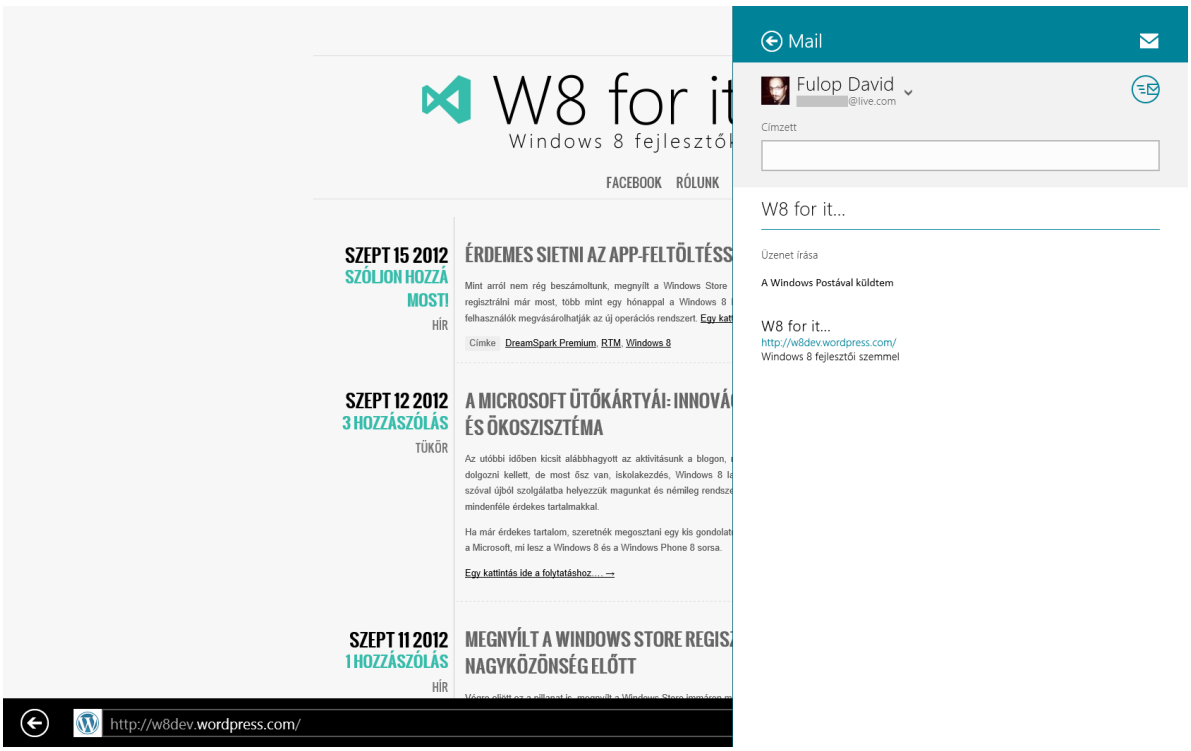
Érdemes tudni, hogy ha a Start képernyőn vagyunk, és leütünk egy alfanumerikus billentyűt, az rögtön a keresést aktiválja, és a billentyű karaktere bekerül a keresődobozba. Tehát ha hozzászoktunk, hogy egyes alkalmazásokat úgy indítunk, hogy lenyomjuk a Windows billentyűt, begépeljük az alkalmazás nevének egy részét, majd rögtön Entert ütünk, továbbra is pontosan így tehetjük meg.

Ha a képernyőn megjelent a keresett erőforrás (legyen az alkalmazás vagy bármi más), megnyithatjuk az ikonjára kattintással vagy érintéssel; ha pedig kijelöljük, az alul megjelenő alkalmazássávon többek között megjelenik a „Pin to Start" gomb, mellyel rögzíthetjük a csempét a Start képernyőn.

Megosztás

Előfordul, hogy adatokat szeretnénk egy alkalmazásból egy másikba juttatni. Például böngészünk egy weboldalt, és szeretnénk e-mailben elküldeni egy ismerősünknek a weblap URL-jét. Ilyenkor elvileg ki kell lépni a böngészőből, el kell indítanunk a levelező alkalmazást, beírni a címzettet, tárgyat, a linket, egyebeket, és küldhetjük a levelet – és ezután visszatérhetünk a böngészőhöz. Ennek a meglehetősen hosszú lépéssorozatnak a leegyszerűsítésére szolgál a Charm bar második gombja, a Share, vagyis Megosztás.

Amikor egy olyan alkalmazásban vagyunk, mely képes adatokat közölni más alkalmazásokkal, a Share gombot aktiválva egy listát kapunk azokról az alkalmazásokról, melyek képesek az éppen aktív alkalmazás által szolgáltatott adatot fogadni. (Az alkalmazások fejlesztésekor jelölhetjük meg, hogy milyen típusú adatot tudnak megosztani vagy fogadni.) Ha kiválasztjuk a célalkalmazást, akkor az aktiválódik, de nem normál üzemmódban, hanem értesül róla, hogy őt egy megosztás céljaként aktiválták. Ennek megfelelően egy külön erre a célra szolgáló megjelenést lehet adni az alkalmazásnak, amely nem teljes képernyős, és csak részben takarja ki a megosztás forrásaként funkcionáló alkalmazást, ahogy az a 2-12 ábrán is látható.



2-12 ábra: Megosztás az Internet Explorerből az e-mail-küldő alkalmazás felé

A célalkalmazás fogadja az adatot, és feldolgozza azt, majd ismét a háttérbe kerül, a felhasználó pedig folytathatja például a böngészést.

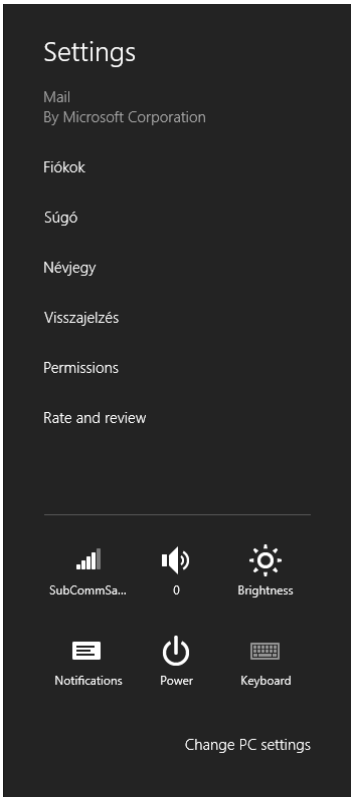
Ezzel a módszerrel a megosztás már nem alkalmazások közötti váltogatást eredményez, hanem egy egyszerű, rendszer szinten egységes (tehát a felhasználó számára könnyen megtanulható) rendszert, amelynek bármely Windows Store alkalmazás könnyedén a részévé válhat.

Eszközök és Beállítások

A Start gomb alatti két gomb a Devices („Eszközök”) és a Settings („Beállítások”).

A Devices gombbal a PC-hez vagy táblagéphez csatlakoztatott eszközökről kapunk egy listát; itt találhatjuk meg például a másodlagos monitor beállításait.

A Settingset aktiválva már több lehetőséget kapunk: megjelenik a Beállítások oldalsáv. Ennek pontos elemei változóak. Más jelenik meg, ha a Start képernyőről nyitjuk meg, más, ha az Asztalról (az Asztal eléréséről rövidesen szó lesz), és más elemek jelennek meg, ha úgy nyitjuk meg, hogy egy Windows Store alkalmazás volt az aktív. A 2-13 ábra a Mail alkalmazásból megnyitott beállítások oldalsávot mutatja.



2-13 ábra: A Mail alkalmazás beállításai

A beállítások oldalsáv alsó részének tartalma állandó. Itt az alábbi lehetőségeink vannak:

- **Hálózat**: Az aktuális számítógépes hálózatot (LAN/WLAN) mutatja, illetve annak beállításait, valamint a rendelkezésre álló hálózatok listáját érhetjük el rajta keresztül.
- **Hangerő**: A gép hangerejét állíthatjuk vele.
- **Fényerő**: A képernyő fényerejét állíthatjuk vele. (Nem mindig érhető el.)
- **Értesítések**: A háttérben futó alkalmazások értesítéseket dobhatnak fel a képernyőre, ha történt valami érdekes esemény. Ezeket az esetleg zavaró értesítéseket tilthatjuk le itt átmenetileg.
- **Állapotváltás**: Itt érhető el a számítógép kikapcsolása, újraindítása, alvó állapotba küldése stb.
- **Billentyűzet**: A szoftveres billentyűzet beállításait változtathatjuk meg. (Nem mindig érhető el.)
- **Beállítások megváltoztatása** (*Change PC Settings*): Megnyitja a PC Settings alkalmazást, mellyel a számítógép rengeteg beállítását módosíthatjuk. Amit a PC Settings alkalmazásban sem találunk, azt a Control Panelből (Vezérlőpult) érhetjük el.

A Settings oldalsáv felső része változó tartalommal bír. Ha az Asztalról nyitjuk meg, rögtön elérhető róla például a Control Panel, a Personalization (Témabeállítások), és egy kattintással megnézhetjük a gépinformációkat.

Ha a Start képernyőről nyitjuk meg, a beállítások felső részének fontosabbik pontja a „Tiles", vagyis csempék link. Ezt aktiválva beállíthatjuk, hogy szeretnénk-e, ha megjelennének csempeként a Start képernyőn a rendszergazdai eszközök, illetve itt törölhetünk minden személyes információt a csempékről.

A harmadik lehetséges eset, hogy egy Windows Store alkalmazás futtatása közben nyitjuk meg a beállítások oldalsávot. Ekkor nem a korábban leírt pontok jelennek meg, hanem teljes egészében az alkalmazás által beállítottak. Vagyis ha szeretnénk alkalmazásunkhoz egy olyan képernyőt biztosítani, ahol a felhasználók az alkalmazással kapcsolatos beállításokat módosíthatják, azt nem feltétlenül az alkalmazáson belül érdemes elhelyezni mint külön oldalt, hanem összeköthetjük a Windows által szolgáltatott lehetőséggel, hogy a Settings menüpont alatt jelenhessen meg, ahogy az korábban is látható volt a 2-13 ábrán.

Az Asztal

Amellett, hogy a Windows 8 modern felülete nagyon jól használható – nemcsak a táblagépeken, de még az asztali számítógépeken is élvezhetjük a letisztult felületek átláthatóságát –, nem minden feladatra alkalmasak a Windows Store alkalmazások és a modern tervezési elvek szempontjaira épülő felületek. Nem is beszélve arról a tényről, hogy a rengeteg már kész és jól működő Windows alkalmazás jelentős részéből várhatóan sohasem lesz Windows Store alkalmazás.

A Windows 8-nak nemcsak a megújulás a célja, hanem emellett az is, hogy megtartsa a korábbi verziók folytonosságát, és így negatív változástól mentesen tovább dolgozhasson az is, aki kifejezetten a korábbi alkalmazásokhoz kötődik.

Ehhez viszont nem árt, ha a felhasználó ismerős környezettel találkozik. A Windows gyakorlatilag első verziója óta rendelkezik egy „Asztallal" (Desktop), amelyen a különböző méretű ablakokban megjelenő alkalmazásokat elhelyezhetjük. Nincs ez másképp a Windows 8 esetében sem.

Most először a Windowsok történetében a felhasználó a bejelentkezése után nem az Asztallal találja magát szemben! Ehelyett a Start képernyő fogadja. Ez a komoly változás igazából nagyon hasznos döntés volt: a viszonylag statikus, ikonokkal teletűzdelt Asztal helyett a Start képernyőn lévő élő csempék azonnal összefoglalják a felhasználónak, hogy mi történt utolsó bejelentkezése óta. A levelező alkalmazás csempéjén megjelennek a legutóbbi levelek, a csevegő alkalmazás csempéje az érkezett azonnali üzeneteket mutatja és így tovább.

Átkapcsolás az Asztalra

Ha szeretnénk megnyitni az Asztalt, csak rá kell kattintanunk csempéjére (*Desktop*) a Start képernyőn, vagy meg kell érintenünk azt. Ha esetleg nem találnánk a csempét, a Keresés pont alatt leírt módon ezt is fellelhetjük az alkalmazások között, és újra kitűzhetjük a Start képernyőre. A csempe egyébként mindig az Asztal hátterét mutatja, tehát azt a képet, amit aktuálisan az Asztalon is látnánk.

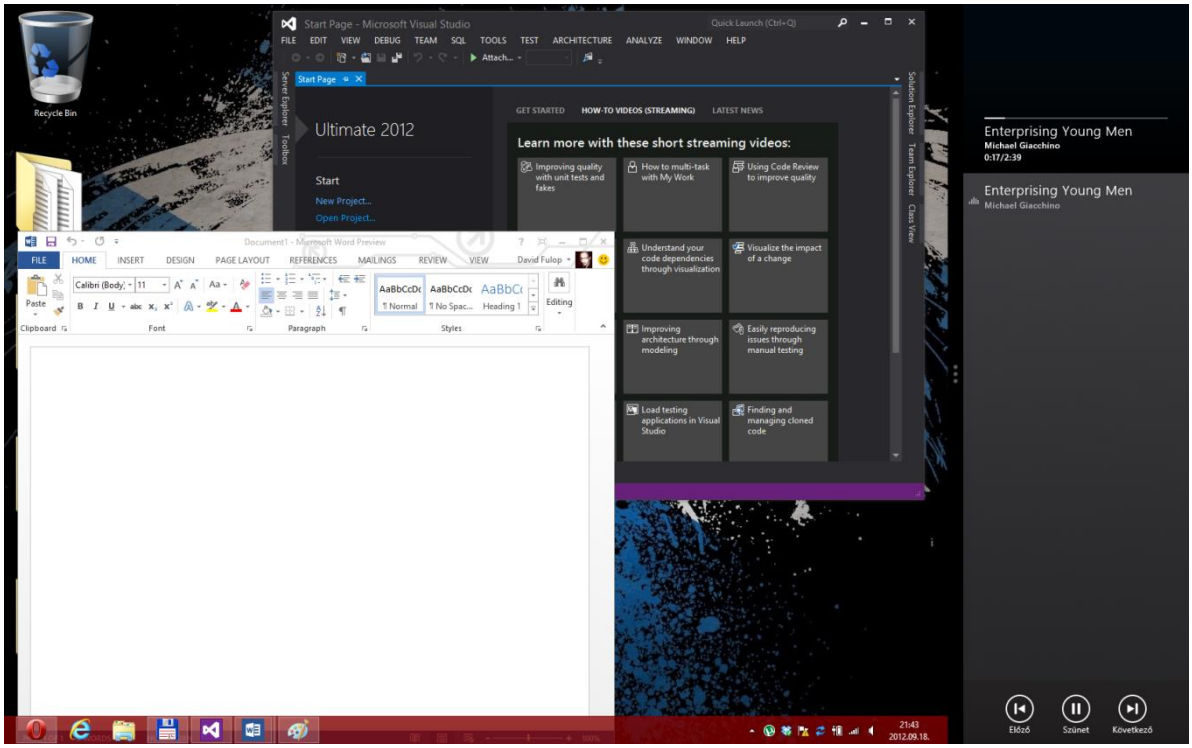
Ha egy olyan alkalmazást indítunk a Start képernyőről, mely nem Windows Store alkalmazás, a rendszer automatikusan megnyitja az Asztalt, és ott indítja el azt.

Az Asztal és az alkalmazások használata

Az első és legfontosabb dolog az itt részletezendő, elsőre talán furának tűnő lehetőségek megértéséhez: az Asztal nem különálló, a Windows 8 eddig bemutatott részeitől független világ, hanem egy Windows Store alkalmazásnak fogható fel. Természetesen valójában nem az, így véletlenül sem fordulhat elő, hogy eltávolítjuk a számítógépről.

Bár a Windows 8 Asztalának feladata, hogy elérhetővé tegye és megjelenítse a nem Windows Store alkalmazásokat, azonban sok dologban maga is hasonlít a Windows Store alkalmazásokra. Például ki lehet belőle lépni, mégpedig ugyanazzal a módszerrel, amivel a Windows Store alkalmazásokból: megragadjuk az Asztal tetejét, és lehúzzuk a képernyő aljára, majd ott elengedjük.

Emellett az Asztal is képes megosztani a képernyőt egy Windows Store alkalmazással. Nemcsak teljes képernyőn jelenhet meg, hanem ismeri a kisméretű (Snapped) és nagyméretű (Filled) módokat. Míg nagyméretű módban tulajdonképpen egy kisebb Asztalt kapunk, ahogy a 2-14 ábrán is látható, kisméretű módban a futó asztali alkalmazások már nem jelennek meg közvetlenül, helyettük csak a betekintési képük, egymás alatt.



2-14 ábra: „Filled” Asztal és egy „snapped” Windows Store alkalmazás egyszerre a képernyőn

Észrevehetjük, hogy a Start gomb eltűnt a bal alsó sarokból. Azonban ha a bal alsó sarokba visszük az egérkurzort, mint mindig, ezúttal is feltűnik a Start képernyő kicsinyített képe, amelyre kattintva az meg is nyílik.

A képernyő bal oldalán elérhető futó alkalmazások listáján az Asztal egyetlen alkalmazásként jelenik meg. A „benne” futó asztali alkalmazások ezen a listán nem láthatóak. Amikor az Asztal alkalmazás előtérben van, vagy a Tálca, vagy az Alt+Tab billentyűkombináció segítségével váltogathatunk az ott futó alkalmazások között, ahogyan a régebbi Windows változatokon eddig is megtehettük. A Windows+Tab billentyűkombináció működése viszont megváltozott: ezzel már a futó Windows Store alkalmazások listájának elemei között lépkedhetünk.

Az Asztal tulajdonképpen csak megjelenítő felület a Windows alkalmazásokhoz, azok semmilyen egyéb módon nem függenek tőle. Éppen ezért ha leállítjuk az Asztal (Desktop) alkalmazást, a futó alkalmazások nem állnak le, csupán addig nem látjuk őket, amíg újra meg nem nyitjuk az Asztalt.

Összegzés

A Windows megújult mind belsőleg, mind külsőleg. Ebben a fejezetben megismerkedhettünk a felhasználói felületen bekövetkezett változással, melynek fő oka és célja, hogy – egyedülálló módon – egyszerre alkalmazkodjon az érintőképernyőkön keresztüli kezeléshez, ugyanakkor továbbra is kényelmesen használható legyen egérrel és billentyűzettel is. Alkalmazásainkat érdemes úgy felépíteni, hogy az itt látott kezelési újdonságokat figyelembe vesszük, és ezáltal szorosabban integrálódunk a Windows Store alkalmazások ökoszisztémájába.

3. A Windows 8 architektúrája — a fejlesztő szemszögéből

Ebben a fejezetben az alábbi témákat ismerheted meg:

- A Windows 8 stílusú és a hagyományos asztali alkalmazásokhoz kapcsolódó technológiák komponensei, azok rétegződése
- A Windows Runtime szerepe a Windows 8 stílusú alkalmazások fejlesztésében
- Programozási nyelvek, amelyek hozzáférhetnek a Windows Runtime programozási felületéhez
- A .NET keretrendszer 4.5 változatának legfontosabb újdonságai
- A megfelelő programozási nyelv és technológiakészlet kiválasztása saját Windows 8-on futó alkalmazásaid elkészítéséhez

Amikor alkalmazásokat fejlesztesz, általában egymással együttműködő technológiák készletével dolgozol. A Windows fejlesztői platformja bőséges kínálattal rendelkezik fejlesztéshez használható eszközökből, módszerekből és technológiákból. Az elmúlt évtizedben ezeknek a komponenseknek a száma jelentősen megnőtt. A Windows 8 egy új alkalmazásfejlesztési modellt kínál egy új alkalmazástípuson keresztül – ezeket Windows 8 stílusú alkalmazásoknak nevezzük –, amely több programozási nyelvet is támogat (C++, C#, Visual Basic, JavaScript), és alacsonyan tartja a használatához szükséges technológiai komponensek számát.

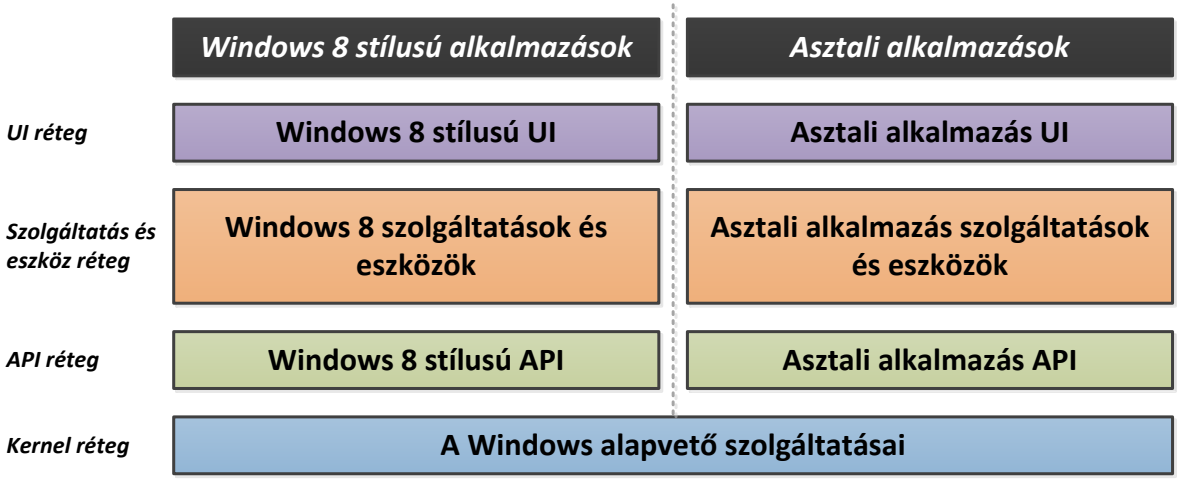
Ebben a fejezetben a Windows 8 stílusú alkalmazásokhoz szükséges komponenseket, ezek architektúráját ismerheted meg. Először a hagyományos asztali alkalmazások és a Windows 8 stílusú alkalmazások fejlesztési technológiái közötti különbségeket tekintheted át. Ahogyan azt meg fogod tanulni, a Windows 8 stílusú alkalmazások alapvető komponense a Windows Runtime, és a fejezet végére tisztában leszel ennek felépítésével és hasznosságával. A megfelelő technológia kiválasztása egy adott alkalmazáshoz nem egyszerű dolog. Ez a fejezet egy olyan résszel zárul, amely segít téged a döntés kialakításában.

A Windows 8 fejlesztői architektúrája

Amint azt már tudod, a Windows 8 lehetővé teszi egy új alkalmazástípus – a Windows 8 stílusú alkalmazások – fejlesztését, és a hagyományos asztali alkalmazások továbbra is futnak a Windows-on. Sőt, a kedvenc eszközeidet és technológiáidat továbbra is használhatod a Windows 8 stílusú alkalmazások fejlesztéséhez.

A Windows 8 architektúráért felelős csapata fontos döntést hozott, amikor új, önálló komponenskészletet alkottak meg a Windows 8 stílusú alkalmazásokhoz. A Windows 8-ban két alapvető készletet találsz (egyet a Windows 8 stílusú alkalmazásokhoz, egyet pedig az asztali alkalmazásokhoz), és ezek egymás mellett használhatók, amint azt a 3-1 ábra is mutatja.

> *Architektúra rétegnek nevezzük az azonos szerepkörben lévő komponenseket (például azokat, amelyek az operációs rendszer szolgáltatásaival kommunikálnak). Komponenskészletnek nevezzük az architektúra rétegek egymásra épülését, amelyek azt tükrözik, hogy az egyes szolgáltatások miként használják egymást.*



3-1 ábra: A Windows 8 két komponenskészlete

Bár a Windows 8 stílusú és asztali alkalmazások ugyanazt az operációs rendszer kernelt használják, az alkalmazásfejlesztés során néhány fontos dolgot figyelembe kell venned. A 3-1 táblázat ezt a két alkalmazástípust hasonlítja össze.

3-1 táblázat: Az asztali és a Windows 8 stílusú alkalmazások összehasonlítása

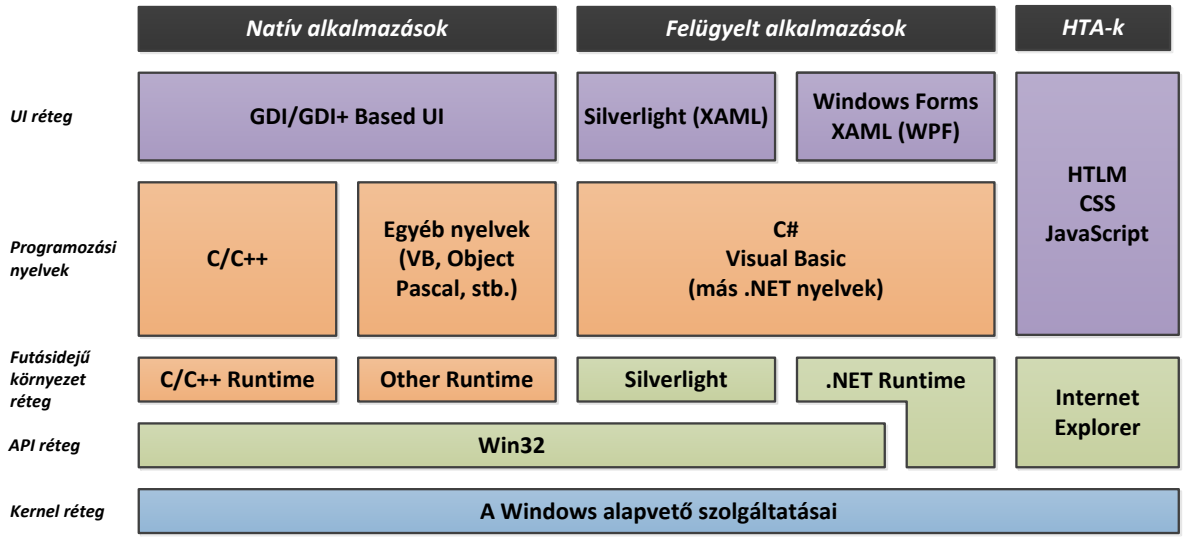
Asztali alkalmazások	Windows 8 stílusú alkalmazások
Az asztali alkalmazások vagy a Win32 API elérésével (ez 1995-től érhető el, illetve néhány szolgáltatása egészen 1991-ig nyúlik vissza) programozhatók, vagy a .NET keretrendszerrel, amely felügyelt típusokra épít. Amint azt korábban megtanulhattad, több programozási nyelvet, eszközt és technológiát is használhatsz. Bár a választási lehetőségek gazdagsága nagyszerű dolog, alkalmazásfejlesztési szempontból hátrányos is lehet. Attól függően, hogy mi az alkalmazásod célja – illetve mi a szándékod vele –, technológiák és nyelvek keverékével kerülhetsz szembe, hogy leprogramozd az elképzelt funkcionalitást.	A Windows 8 stílusú alkalmazások új megközelítésmódot jelentenek. Első osztályú felhasználói élményt kínálnak többpontos érintéssel, szenzorokkal, és az előtérbe helyezik a felhasználói felület érzékenységét, a felhasználó akcióira való azonnali reagálást. Képzelj el egy olyan alkalmazást, ahol az ujjaddal egy tárgyat vontathatsz keresztül a képernyőn elakadások nélkül, ahelyett, hogy az darabosan mozogna, miközben az egérrel mozgatod! A tökéletes élmény érdekében a Windows 8 stílusú alkalmazások ezt az érzékenységet natív módon biztosítják. Természetesen a Windows 8 stílusú alkalmazások továbbra is tökéletes billentyűzet- és egérintegrációt biztosítanak.
A legtöbb asztali alkalmazás ablakokat és dialógusokat jelenít meg a képernyőn felhasználói inputra, válaszra, illetve megerősítésre várva. Néha ezek az alkalmazások több dialógust is megjelenítenek, amelyek között a felhasználó kapcsolgathat. Mialatt a felhasználó ezekkel az ablakokkal dolgozik, gyakran különböző feladatokra fókuszál, akár a tudata alatt.	A Windows 8 stílusú alkalmazások olyan intuitív felhasználói felületet kínálnak, ahol az alkalmazás az egész képernyőt birtokolja. Ez a megközelítési mód nem a képernyőn felugró dialógusok megjelenítésére bátorítja a fejlesztőket, hanem inkább a több alkalmazáslap használatára, amint azt a böngészőben futó webes alkalmazások is teszik. A Windows 8 stílusú alkalmazások a Windows Store-on keresztül telepíthetők, és meg kell felelniük azoknak a felhasználói felülethez és élményhez kapcsolódó irányelveknek, amelyeket a Microsoft azért publikál, hogy az alkalmazások átmehessenek a minőségi vizsgálatokon.

Asztali alkalmazások	Windows 8 stílusú alkalmazások
Az asztali alkalmazások általában olyan telepítési eljárásokat használnak, amelyek néhány lépésben végigvezetik a felhasználót ezen a folyamaton. A teljes eljárás akár néhány percet (esetleg hosszabb időt is) igénybe vehet, hogy az összes fájl és egyéb erőforrás a számítógépre kerüljön a szükséges beállításokkal együtt. Az alkalmazások eltávolítása szintén hasonló szabványos eljárással történik, amelyhez a felhasználónak el kell indítania a Vezérlőpultot.	A Windows 8 stílusú alkalmazások fogyasztóknak – átlagembereknek – készültek, akik nincsenek feltétlenül tisztában olyan fogalmakkal, mint fájl, regisztrációs adatbázis, telepítési eljárás. Ezek a felhasználók azt várják el, hogy egyszerűen adhassanak hozzá egy alkalmazást a már meglévőkhöz, egyetlen érintéssel vagy egérkattintással, és minden szükséges lépést végezzen el a rendszer automatikusan. Ugyanezek a felhasználók azt is elvárják, hogy az alkalmazások eltávolítása hasonlóan egyszerű legyen – nem érdekli őket, a Windows hogyan is valósítja meg ezt a kulisszák mögött.

Amint azt láthatod, a fogyasztóközpontú megközelítésmód eredményeként a Windows 8 stílusú alkalmazásokkal szemben megfogalmazott elvárások jelentősen különböznek az asztali alkalmazásokétól. A Microsoft architektúráért felelős csapata arra az elhatározásra jutott, hogy egy különálló alrendszert készít a Windows 8 stílusú alkalmazásokhoz, és egy másik API-t az alkalmazások építéséhez.

Az asztali alkalmazások rétegei

A hagyományos asztali alkalmazások készítéséhez különböző alkalmazástípusok és a hozzájuk tartozó komponenskészletek állnak a fejlesztők rendelkezésére. Mulatságos (de egyáltalán nem meglepő) az, hogy a „hagyományos" szónak ma már teljesen más jelentése van, mint néhány évvel (mondjuk, talán öt évvel) ezelőtt volt. Mindazonáltal a Windows 8 megjelenésével a 3-2 ábrán bemutatott komponenskészleteket ma már „hagyományosnak" nevezheted. A felügyelt alkalmazások 2002-ben viszonylag újnak számítottak, de a legújabb, a Silverlight is elérhető 2008 óta.



3-2 ábra: Asztali alkalmazások komponenskészlete

Ne ijedj meg, ha nem ismersz minden technológiát, amelyeket a 3-2 ábra nevesít! Ennek célja az, hogy megmutassa az asztali alkalmazásokhoz kapcsolódó technológiai komponensek széles választékát. Ha bármelyikük iránt érdeklődsz, használd az MSDN-t (`http://msdn.microsoft.com`), és keress rá azokra nevük alapján!

Figyeld meg a 3-2 ábrán a programozási nyelvek rétege és a UI réteg közötti kapcsolatot! Amint azt láthatod, a kiválasztott programozási nyelv alapvetően meghatározza azoknak a technológiáknak a körét, amelyeket az asztali alkalmazások elkészítéséhez használhatsz.

A natív alkalmazásokhoz a C vagy C++ nyelveket választva – vagy más natív nyelveket (mint a Visual Basic, Object Pascal és így tovább), amelyeket a fordítóprogram közvetlenül CPU-specifikus kódra fordít – a Graphics Device Interface (GDI) vagy GDI+ technológiákat választhatod a felhasználói felület számára.

A felügyelt alkalmazások esetében is még mindig választhatod a GDI-t vagy a GDI+-ot. A .NET-ben ezek a Windows Forms fedőnév alatt használhatók. Modernebb és hatékonyabb választást jelent a Windows Presentation Foundation (WPF), amely az Extensible Application Markup Language (XAML) jelölésre épül, illetve annak fiatalabb, de nem kevésbé modern testvére, a Silverlight.

> *Míg a natív alkalmazások közvetlenül CPU-specifikus kódra fordulnak, addig a felügyelt alkalmazások egy közbenső nyelvre. Amikor egy felügyelt alkalmazást futtatsz, ezt a közbenső nyelvet az ún. just-in-time (JIT) fordító CPU-specifikus kódra alakítja.*

Bár nem népszerű asztali alkalmazásokat HTML-lel és a kapcsolódó technológiákkal létrehozni, a HTML Applications (HTA-k) modellel erre is lehetőséged van.

A nyelvi preferenciád egyúttal azokat a futásidejű könyvtárakat és környezeteket is meghatározza, amelyeket a programozás során használhatsz (a futásidejű környezet réteg mutatja ezeket a 3-2 ábrán). Ezek a könyvtárak olyan műveleteket tartalmaznak, amelyeket a programozási nyelv közvetlenül használhat az operációs rendszer szolgáltatásainak elérésére, mint például értékek megjelenítésére, fájlok kezelésére vagy éppen hálózati kommunikációra. A natív alkalmazások esetében minden programozási nyelvnek megvan a saját futásidejű könyvtára, mint pl. a C és C++ nyelvek esetében a Microsoft Foundation Classes (MFC) vagy az Active Template Library (ATL), a Visual Basic (a jó öreg Basic nyelv, amely még a .NET megjelenése előtt jött létre) esetében a VB Runtime, a Delphi (Object Pascal) kapcsán pedig a Visual Component Library (VCL).

A .NET eliminálja ezt a nyelvi függőséget a saját Base Class Library komponensének bevezetésével, amely az összes .NET nyelvből elérhető, beleértve a C# és Visual Basic nyelveket éppen úgy, mint az egyéb népszerű .NET nyelveket, az F#-ot, az IronPythont vagy az IronRuby-t. Ezt az egységes képet némileg elhomályosította a Silverlight megjelenése, amely egy újabb .NET futásidejű környezetet teremtett, amely egyaránt használható a böngészőben (Internet Explorer, Firefox, Safari és néhány más) és az asztalon futó alkalmazásokban. A Silverlight Base Class Library-ja csupán egy részét teszi elérhetővé a .NET keretrendszerben található típusoknak.

A natív alkalmazások futásidejű könyvtárai a Win32 API-ra épülnek (ezt a 3-2 ábra API rétege mutatja), amely egy sima API több tízezer belépési ponttal, amelyek a Windows kernel alapvető szolgáltatásait érik el. Ezzel szemben a .NET saját futásidejű környezettel rendelkezik – ezt Common Language Runtime-nak (CLR) nevezzük –, és ez a Win32 API fölött egy sokkal jobb absztrakciót biztosít, ahol a szolgáltatásokat újrahasznosítható típusok adatstruktúráiba és műveleteibe csomagolva érhetjük el.

Képzeld el, hogy te vagy az a szoftver építész, akinek egy komponenskészletet kell létrehoznia a Windows 8 stílusú alkalmazások számára! Egybefésülnéd ezeket a már létező asztali technológiák készletével? Lehetséges. Mindazonáltal a Microsoft architektúráért felelős csapata úgy döntött, hogy egy új, önálló API-készletet hoz létre elkerülendő a technológiák további töredezettségét.

A Windows 8 stílusú alkalmazások rétegei

A Windows 8 operációs rendszer fogyasztóközpontú szemléletével a Microsoft új kihívásokkal szembesül. Az intuitív, többpontos érintést biztosító, a felhasználó beavatkozásaira azonnal reagáló felhasználói felület létrehozása csak egyike ezeknek. Ennél sokkal nagyobb feladatot jelent olyan fejlesztői platform kialakítása, amely a megfelelő módon támogatja a fejlesztőket, lehetővé teszi, hogy a már ismert eszközöket és technológiákat használják, és ami a legfontosabb, produktívak lehessenek.

A kihívás

A Microsoft erős Windows fejlesztői közösséggel rendelkezik. Nagyszámú fejlesztő az IT fogyasztó-központúvá alakulását a webes alkalmazások fejlesztése kapcsán ismerte meg. Azonban a fejlesztőknek csak igen kis része ismeri a Windows-alapú fogyasztói eszközöket, egészen pontosan a Windows Phone fejlesztők. Hosszú ideig a Windows Phone volt az egyetlen valódi, fogyasztókat megcélzó eszköz a Microsofttól.

A világon rengeteg fejlesztő és hobbiként programozó – számuk nagyobb, mint a Microsoft közösségi táborában lévőké – már megismerkedett a fogyasztói eszközökre való alkalmazások fejlesztésével az Android (Google) vagy iOS (Apple) platformokon. A Windows 8-at a Microsoft olyan operációs rendszernek szánja, amely nemcsak a professzionális Windows fejlesztőket vonzza, hanem azokat is, akik nem kötődnek közelebbről a Microsoft közösségéhez.

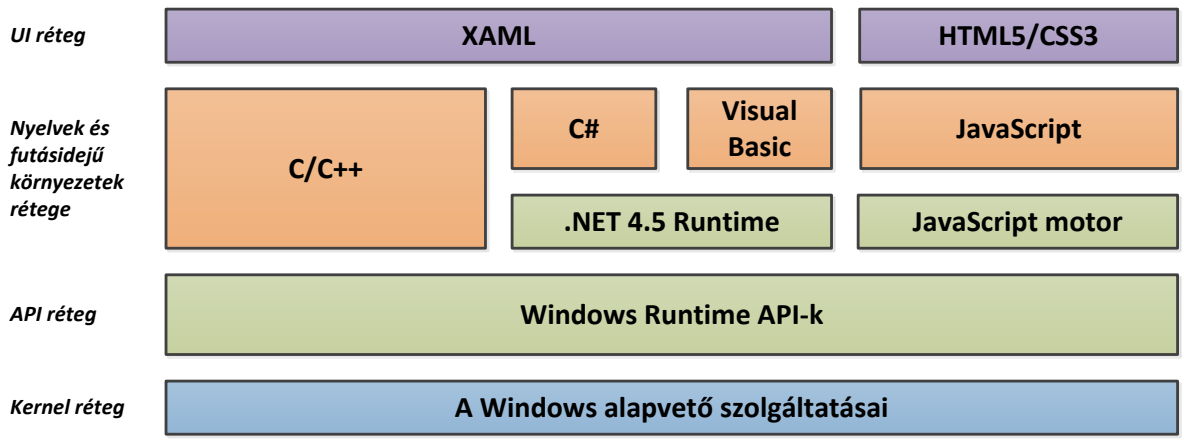
A Microsoft láthatóan későn kapcsolódott be a fogyasztók megnyeréséért való versenybe. Új eszközök és a rajtuk futó operációs rendszer megalkotása csak egy fontos lépés ebben a versenyben, de megfelelő és kimagaslóan jó fejlesztői platform és stratégia nélkül nem lehet olyan minőségi alkalmazásokat biztosítani, amelyek elég erősek a Windows megerősítésében, és valódi, minőségi alternatívát kínálnak a fogyasztók számára.

A Microsoftnak mindig jó fejlesztőeszközei voltak, amelyek az idő előrehaladásával folyamatosan fejlődtek, olyan eszközök, amelyek magas produktivitást tettek lehetővé. Ahelyett, hogy egyetlen programozási nyelvre alapozta volna a platformját (ahogyan ezt az Apple és a Google teszi), a Microsoft több programozási nyelvvel vette körbe magát, lehetővé téve, hogy a fejlesztők saját maguk választhassák ki a céljaiknak legjobban megfelelőt. A legjobb platform biztosítása fogyasztókat megcélzó alkalmazások fejlesztéséhez határozottan a kulcstényező lehet a jobb piaci pozíció eléréséhez.

A Microsoft erre a kihívásra a Windows 8 stílusú alkalmazások fejlesztői platformjával válaszolt.

Az architektúra rétegek áttekintése

Egy független Windows 8 stílusú komponenskészlet létrehozásával a Microsoft új koncepciót vezetett be, amelyek a hagyományos asztali alkalmazásfejlesztés számos nehézségétől megszabadul, újraértelmezi a Windows API és a programozási nyelvek futásidejű környezetének fogalmát. Ez az új koncepció több nyelv egyidejű támogatását valósítja meg egységes programozási felület felett, amint azt a 3-3 ábra mutatja be.



3-3 ábra: A Windows 8 stílusú alkalmazások technológiai rétegei

A 3-3 ábrából több következetést is levonhatsz, beleértve az alábbiakat is:

Minden Windows 8 stílusú alkalmazás egy közös API réteget –a Windows Runtime-ot – használja az alapvető Windows szolgáltatások elérésére, és nincs semmilyen más API, amelyekre a programoknak szükségük volna. A használt programozási nyelvtől függetlenül minden szolgáltatás korlátozások nélkül elérhető, vagyis mindazt, amit C++-ból el tudsz érni, ugyanúgy elérheted C#-ból, Visual Basic-ből és JavaScriptből is.

A HTML-t és a CSS-t fejlesztők milliói használják világszerte webes alkalmazások és weboldalak létrehozásához. Ezek a fejlesztők JavaScriptet használnak a weboldalak (néha komplex) logikájának a leírásához. A Microsoft lehetővé tette, hogy Windows 8 stílusú alkalmazások modelljében a fejlesztők a JavaScriptet is használhassák a Windows Runtime által felkínált API-k eléréséhez.

A HTML legújabb szabványa (HTML5) a Cascading Style Sheets 3-mal (CSS3) kombinálva jóval erőteljesebb, mint elődjei. A HTML5 natív módon képes médiaállományokat lejátszani, illetve vektoros grafikát megjeleníteni – a számítógépben található grafikus kártya hardveres gyorsítási lehetőségeinek használatával. Az elmúlt néhány évben ezernyi weboldal készült HTML5-tel – a felhasználói élmény egy új korszakának jeleként. Az összes ehhez kapcsolódó tudást fel tudják használni a HTML és JavaScript technológiában jártas fejlesztők Windows 8 stílusú alkalmazások készítéséhez.

A Windows 8-ban a XAML-alapú WPF és Silverlight technológiák magja az operációs rendszer része lett, mégpedig natív kódban megvalósítva. A C++, C# és Visual Basic alkalmazások felhasználói felülete XAML-ben definiálható. Ugyanaz a XAML jelölés ugyanazt a felületet jeleníti meg minden egyes programozási nyelvben, korlátozások nélkül. Mivel a felhasználói felület és az operációs rendszerhez való hozzáférést megvalósító API is egységesek, a C++, C# és Visual Basic ugyanazt az alkalmazásmodellt használják.

Ha összehasonlítod az asztali alkalmazások 3-2 ábrán látható rétegeit a Windows 8 stílusú alkalmazások 3-3 ábrán látható rétegeivel, azonnal észreveheted, hogy az utóbbi egyszerűbb a futásidejű környezetek, API-k és UI technológiák tekintetében. Attól függetlenül, hogy web fejlesztő vagy-e, C++ hívő vagy .NET programozó, a Windows 8 stílusú alkalmazások belépési korlátja alacsonyabban van, mint az asztali alkalmazások fejlesztéséhez szükségeseké.

Az általad kedvelt programozási nyelv csak azt a UI technológiát határozza meg, amelyet a programok készítése során kell felhasználnod. A JavaScript a HTML-hez kötődik, míg a többi programozási nyelv a XAML-höz. Általában a fejlesztők több programozási nyelvet és megjelenítési nyelvet használnak, szóval hozzá vannak szokva a többnyelvűséghez. Egy új nyelv megtanulása általában motivál, nem pedig visszatart, azonban több különböző futásidejű környezettel foglalkozni egyidejűleg meglehetősen fárasztó. A Windows 8 stílusú alkalmazások túllépnek ezen a kérdésen. A programozási nyelv választásától függetlenül csak néhány egyszerű API-t kell megtanulnod – azokat, amelyeket a Windows Runtime biztosít.

Kétségtelen, hogy a Windows Runtime a Windows 8 stílusú alkalmazások architektúrájának sarokköve. Hatalmas ugrást jelent a programozási modell fejlődésében, ahhoz hasonlót, amit a .NET keretrendszer kapcsán tapasztalhattunk 2002-ben. A Microsoft architektúráért felelős csapata ezt a ragyogó komponenst egyszerűen úgy írja le, hogy „a Windows 8 stílusú alkalmazások szilárd és hatékony alapja”.

Az 1. fejezetben már megtanultad, hogy a Win32 API műveletek és adatstruktúrák kiterített halmazát tartalmazza. A .NET előtt használt nyelvi futtatókörnyezetek olyan könyvtárakat biztosítottak, amelyek a legtöbb API műveletet elrejtették, és olyan egyszerű objektum- és függvényhalmazokat jelenítettek meg, amelyek egyszerűbbé tették az alapvető programozási feladatokat. A .NET keretrendszer ezekhez az objektumokhoz futásidejű környezetet biztosít olyan extra képességekkel, mint például a memóriakezelés (garbage collection), kivételkezelés, alkalmazások közötti kommunikáció és még rengeteg más.

Bár a nyelvi futásidejű könyvtárak és a .NET keretrendszer már hosszú ideje folyamatosan fejlődnek, mégsem jelenítenek meg minden Win32 API adatstruktúrát és műveletet. Ennek a helyzetnek az elsődleges oka az, hogy a Win32-t használó futásidejű komponenseket az operációs rendszer szolgáltatásaitól külön hozzák létre.

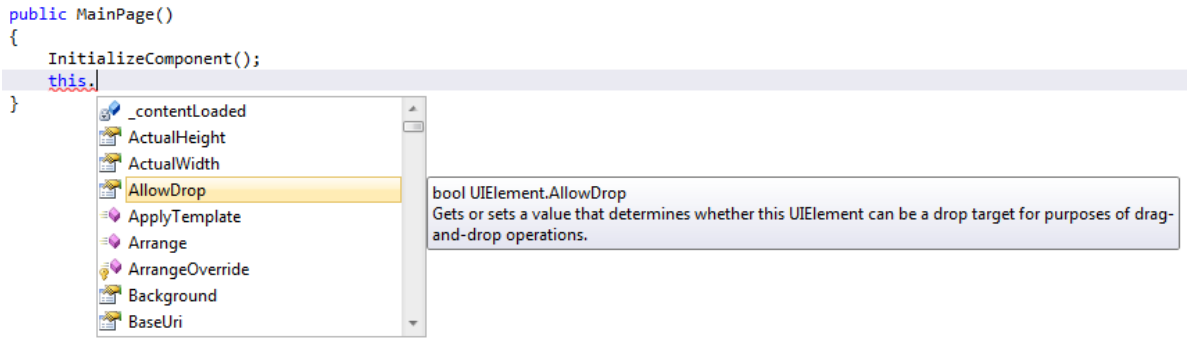
Ez gyökeresen megváltozott a Windows 8 megjelenésével! A Windows Runtime szerves része az operációs rendszernek. Nem egyszerűen hozzáadott komponens, amelyet külön kell telepíteni, mint például a Windows alkalmazásfejlesztő készletet (Windows SDK). Minden alkalommal, amikor a Windows 8 forrásállományai lefordításra kerülnek, a Windows Runtime-hoz tartozó API-k is az operációs rendszer részeként fordulnak. Nézzük meg közelebbről ezt a nagyszerű komponenst!

A Windows Runtime architektúrájának áttekintése

Amint a Windows 8-at a felhasználói élmény előtérbe helyezésével tervezték meg, a Windows Runtime tervezése a fejlesztői élményt helyezte a fókuszba. A modern programozási környezetek – mint például a Visual Studio – nagyszerű eszközöket kínálnak, amelyek a fejlesztőket termelékennyé teszik.

A Windows Runtime tervezési alapelvei

Az IntelliSense kitűnő példája a hatékonyságot javító eszközöknek. Ez folyamatosan figyeli a kontextust, ahogyan a kód részleteit gépeled be, és automatikusan felkínál egy listát az adott kontextusban lehetséges folytatásokról. Amint azt a 3-4 ábra mutatja, az IntelliSense olyan kifejezések listáját kínálja fel, amelyek a **`this`** kulcsszót követhetik a **`MainPage`** metódusban. Az IntelliSense nem csak megjeleníti az elérhető azonosítók listáját, de egyúttal egy rövid magyarázatot is biztosít a kiválasztotthoz (**`AllowDrop`**). A Windows Runtime megvalósításáért felelős csapat folyamatosan szem előtt tartotta azt, hogy egy új API-nak nagyon egyszerűen felhasználhatónak kell lennie az olyan hatékonyságjavító eszközökből, mint amilyen az IntelliSense is.



3-4 ábra: Az IntelliSense termelékenyebbé teszi a fejlesztőket

A Windows 8 felhasználói élményét csak olyan alkalmazások közvetíthetik, amelyek felhasználói felülete azonnal reagál a felhasználó akcióira. A múltban a Windows egyik legnagyobb problémája az alkalmazások és az operációs rendszer lassú és akadozó viselkedése volt. Például, amikor a felhasználó egy objektumot vonszolt a képernyőn, az gyakran darabosan mozgott annak ellenére, hogy a felhasználó az egeret folyamatosan mozgatta.

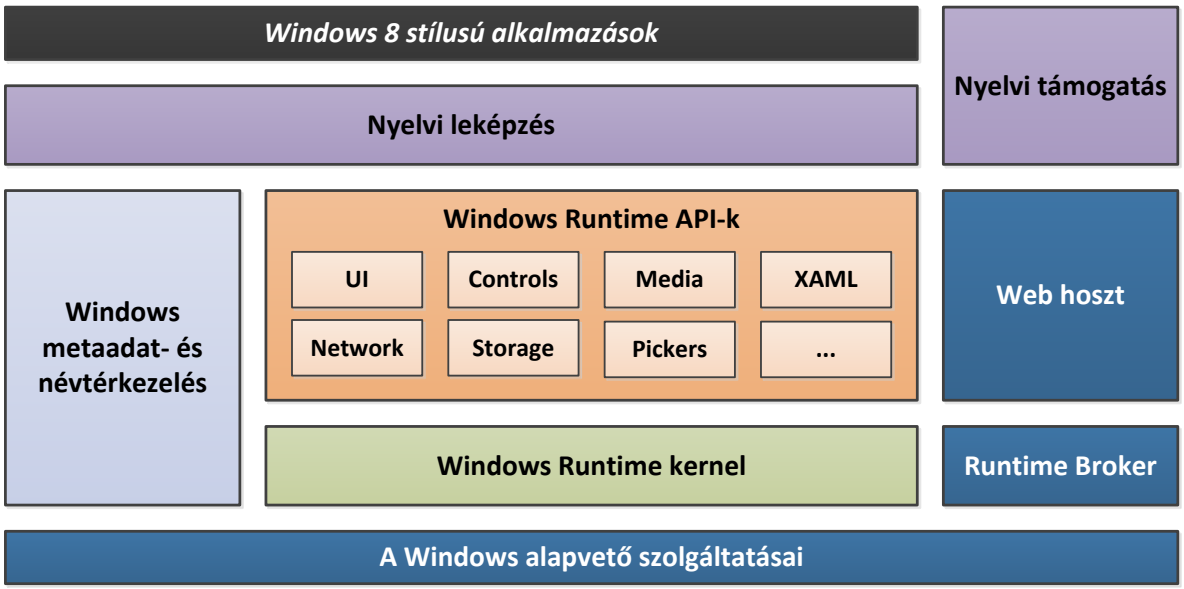
Az aszinkron programozási modell remek gyógyírt jelent erre a problémára! Ez a modell nem blokkolja a felhasználói felületet miközben a háttérben hosszú és erőforrás-igényes számítási műveletek zajlanak, a Windows Runtime felületét az aszinkron modell szem előtt tartásával tervezték meg. A tervezőcsapat egy kemény döntést hozott: minden olyan műveletet, amely 50 ezredmásodpercnél hosszabb ideig tarthat aszinkron műveletként valósítottak meg. A C# és Visual Basic programozási nyelvek új változatai— amelyek a Visual Studio 2012-vel érkeznek –, közvetlenül támogatják ezt a modellt, amint azt az 4. fejezetben megismerhetted.

> *Ne ijedj meg az olyan ismeretlenek tűnő fogalmaktól, mint az „aszinkron programozási modell" vagy a „felhasználói felület blokkolása"! A következő fejezet részletes áttekintést nyújt ezekről. Addig is, tekintsd úgy, hogy az aszinkron végrehajtás lehetővé teszi az időigényes műveletek háttérbe mozgatását, miközben a felület folyamatosan képes a felhasználó akcióira reagálni.*

A Windows csapat mindig is rengeteg munkát fordított az alkalmazások visszamenőleges kompatibilitásának megőrzésére minden egyes új Windows változatban. A Windows Runtime-ot úgy tervezték és készítették, hogy az alkalmazások az új Windows változatokkal is együtt tudjanak futni. Amennyiben egy API megváltozna az operációs rendszer valamelyik újabb változatában, a régebbi műveletváltozatok még mindig elérhetőek maradnak az új változattal párhuzamosan, vagyis azok egymás mellett használhatók. Minden alkalmazás azt a műveletváltozatot használja, amellyel létrejött.

A Windows Runtime építőelemei

A Microsoft tervezési elveit tükrözve a Windows Runtime nem egyszerűen API-k készlete. Ez egy valósi futásidejű környezet, amely az operációs rendszer szolgáltatásait elérhetővé teszi a rá épülő Windows 8 stílusú alkalmazások számára, függetlenül attól, hogy az adott alkalmazást milyen programozási nyelv is valósítja meg. Ez a környezet néhány építőelemből áll össze, amint azt a 3-5 ábra is bemutatja.



3-5 ábra: A Windows Runtime építőelemei

A Windows alapvető szolgáltatásait a rá épülő alkalmazások a Windows Runtime kernelen keresztül érhetik el, amely (mint a neve is elárulja) a futásidejű környezet alapvető komponense. Ez a központi elem burkolja be az alacsony szintű szolgáltatásokat típusokba és interfészekbe. Erre a magra rengeteg API épül, amelyek mindegyike egy adott típusú operációs rendszer feladatkörért felelős. Bár a 3-5 ábra csak néhányat mutat be ezek közül, több mint kéttucatnyi API érhető el.

Nincs szükség arra, hogy minden típusra, interfészre vagy API-ra fejből emlékezz, tudd azt, hogy melyik művelet hol található, mert a Windows Runtime saját metaadat-rendszere leírja ezeket. Sőt, az elemi típusok hierarchikus névterekbe vannak csoportosítva – hasonlóan a .NET keretrendszer vagy a Java futásidejű környezetének objektumaihoz –, és így nagyon egyszerű azokat megtalálni.

Az egyes programozási nyelvek különböző jellegzetességekkel bírnak, és eltérő konvenciókat használnak. Az API-k és a metaadatok kezeléséért felelős blokkok felett egy vékony réteg, a Nyelvi leképzés rétege található, amely az egyes API-k nyelvekre specifikus megjelenítéséért felelős. A Windows 8 stílusú alkalmazások a Windows Runtime API-jait ezen a rétegen keresztül érik el.

Néhány egyéb komponens teszi teljessé a 3-5 ábrán ábrázolt környezetet, amelyek segítségével a Windows 8 stílusú alkalmazások az operációs rendszer minden képességét kihasználhatják a Windows Runtime-on keresztül.

- Az alkalmazások olyan erőforrásokat használhatnak (ilyen például a webkamera, a mikrofon vagy az Internet), amelyek kapcsán a felhasználó engedélyére van szükség. Ezek az erőforrások a Runtime Broker komponensen keresztül érhetők el, amely megkérdezi a felhasználót, hogy ténylegesen engedélyezi-e az erőforráshoz való hozzáférést – a legelső alkalommal, amikor egy művelet megpróbálja azt használatba venni. Például, ha egy olyan alkalmazásról van szó, amely a felhasználóról egy fotót készít a webkamerával, a Runtime Broker nem engedi a rendszernek ezt a fotót elkészíteni mindaddig, amíg a felhasználó azt explicit módon nem engedélyezi.
- Amint azt már megismerted, a HTML5 és a JavaScript is használható a Windows 8 alkalmazások létrehozásához. Ezek az alkalmazások – a webes természetük kapcsán – egy Web hosztban futnak.
- Minden nyelv kapcsán szükség van valamilyen specifikus futásidejű támogatás biztosítására. Például olyan könyvtárakra, amelyek az adott nyelv működéséhez kapcsolódnak, amint a **`printf`** C++ metódus vagy az **`append`** JavaScript függvény, esetleg a C#-ban és Visual Basic-ben elérhető **`String.Split`** művelet. Ezek a Nyelvi támogatás blokkban találhatók, amely a Nyelvi leképzéssel működik együtt.

A 3-5 ábrán feltüntetett néhány építőelem kiemelten fontos szerepet játszik a mindennapi programozási tevékenységek során. A következő fejezetrészekben több részletet is megismerhetsz ezekről.

Metaadatok a Windows Runtime-ban

Az, hogy a Windows Runtime lehetőséget biztosít metaadatok használatára, igen fontos képesség a Windows 8 stílusú alkalmazások fejlesztése kapcsán. A metaadatok nélkülözhetetlenek az architektúra robusztusságához, az API képességeinek felfedezéséhez, és így hosszú távon a fejlesztők produktivitásához. Hát, a Windows különböző technológiái bizony meglehetősen szétszórtak a felhasznált metaadatok formátumának tekintetében!

- A Win32 API adatstruktúrák és műveletek gyűjteménye, amelyeket DLL-eken keresztül lehet elérni. Ezek a DLL-ek a **`System32`** mappában vannak a Windows telepítési könyvtárában. Közöttük található a **`kernel32.ddl`**, a **`user32.dll`**, a **`netapi32.dll`**, **`avicap32.dll`** és még sok másik. A DLL-eknek sok belépési pontjuk van. Azonban ha csak egyszerű **`.dll`** fájljaid vannak, nem tudod megmondani, hogy azok milyen műveleteket tartalmaznak, ugyanis ezek a fájlok nem önleírók.
- A COM (Component Object Model), amelyet a Microsoft 1993-ban vezetett be egy bináris szabvány. Ez lehetővé teszi, hogy a fejlesztők az objektumok interfészeit IDL (Interface Definition Language) nyelven írják le. Egy **`.idl`** fájl ún. Type Library fájllá (**`.tbl`**) fordítható és ez a COM objektumok metaadatait jeleníti meg.
- A .NET keretrendszer a típusokhoz tartozó metaadatokat első osztályú konstrukcióként kezeli. A .NET-ben a típusok azok az objektumok, amelyek a felhasználható műveleteket tartalmazzák, és ezeket a műveleteket a metaadatok írják le. A szerelvények (assembly), amelyek valamelyik .NET nyelvi forráskód lefordításával állnak elő, önleírók. A típusokat és műveleteiket leíró metaadatokat a szerelvények foglalják magukba.

A .NET nagyszerű módon kezeli a metaadatokat! Sajnos, ezek a metaadatok csak felügyelt típusokhoz kapcsolhatók. Rengeteg Win32 művelethez nincs azt beburkoló .NET-es típus! Ha ezeket felügyelt kódból kell használni, olyan definícióra van szükség, amely ezeket a hiányzó metaadatokat pótolja. Ezeknek a definícióknak a leírása munkaigényes, és mindenképpen szükség van a művelet kapcsán olyan dokumentációra, amely leírja annak használatát. Ilyen példa az **`avicap32.ddl`**-ben található **`capCreateCaptureWindows`** művelet definíciója:

```
[DllImport("avicap32.dll", EntryPoint="capCreateCaptureWindow")]
static extern int capCreateCaptureWindow(
  string lpszWindowName, int dwStyle,
  int X, int Y, int nWidth, int nHeight,
  int hwndParent, int nID);
```

Ha rosszul építed fel a definíciót (például **`double X`**-et használsz **`int X`** helyett), a forráskód lefordul, de futásidőben hiba keletkezik, és rengeteg erőfeszítésedbe kerülhet a probléma feltárása és elhárítása.

A metaadatok formátuma

A Windows Runtime tervezőit a .NET keretrendszer inspirálta, amikor megalkották a metaadatokat kezelő alrendszer architektúráját. Úgy döntöttek, hogy a .NET szerelvények metaatadatainak formátumát fogják használni, mert az az elmúlt tíz évben már bizonyította alkalmasságát. A tervezők a .NET keretrendszer 4.5 változatának formátuma mellett döntöttek. Ez a .NET keretrendszer legújabb változata, amely a Windows 8-cal együtt jelent meg.

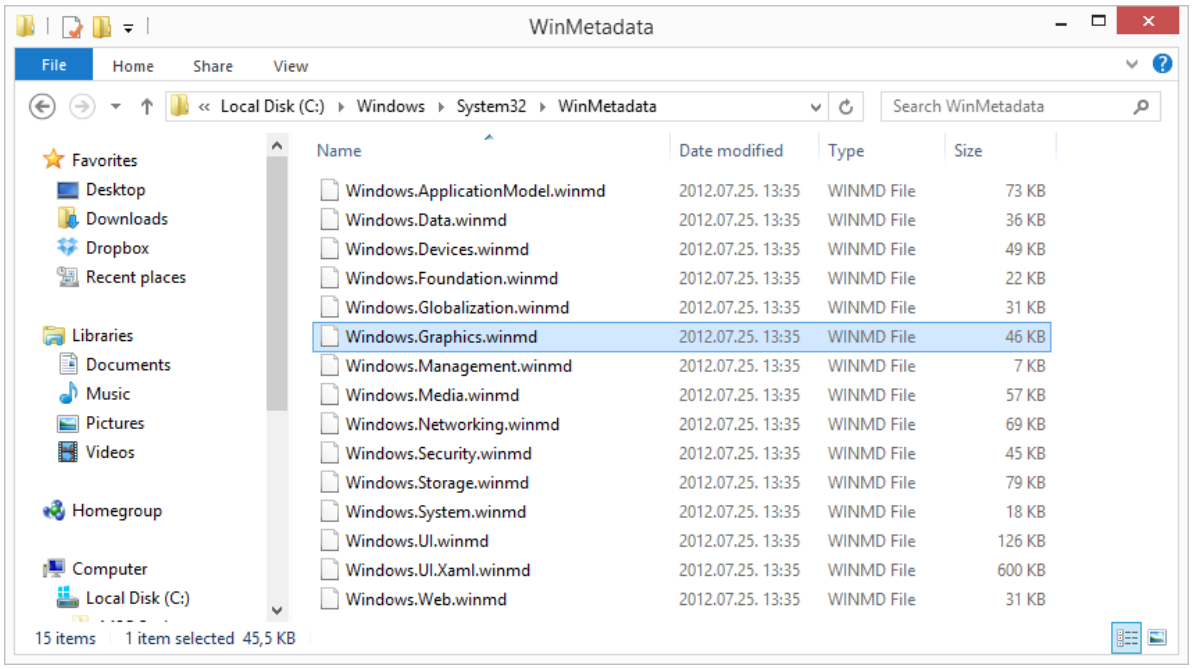
A metaadatfájlok minden egyes – a Windows Runtime-ban elérhető – API-ról teljes körű információt biztosítanak. Ezek a Windows 8-cal együtt települnek, és így azokat minden egyes Windows 8-at tartalmazó számítógépen megtalálhatjuk függetlenül attól, hogy azt fejlesztésre vagy teljesen más célra használják. Ezeket a fájlokat programok is olvashatják, értelmezhetik, sőt te is egyszerűen megvizsgálhatod a tartalmukat, amint azt a következő gyakorlatból megtanulhatod.

***Gyakorlat**: A Windows metaadatok megtekintése*

A Windows Runtime metaadatfájljai a Windows telepítési könyvtárában helyezkednek el. Kiterjesztésük .winmd, és tartalmukat az ILDASM segédprogrammal vizsgálhatod meg.

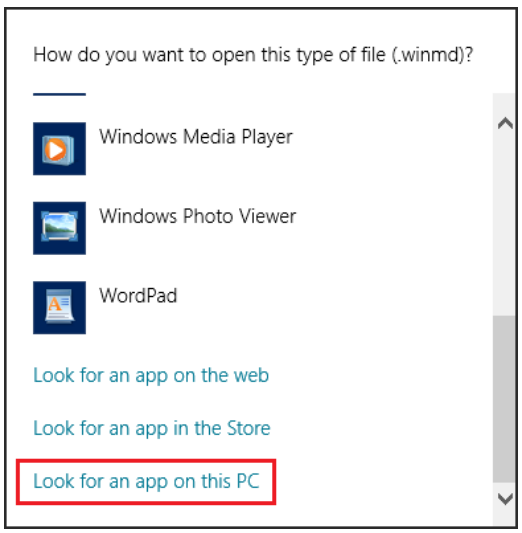
A metaadatfájlok megtekintéséhez kövesd az alábbi lépéseket:

1. A Start képernyőn kattints a Desktop lapkára, majd a tálcán kattints a Windows Explorer-re!
2. Navigálj a **`System32\WinMetadata`** mappába, amely a Windows telepítési könyvtára alatt található! Ha az operációs rendszer az alapértelmezett helyére van telepítve, akkor ezt a mappát a **`C:\Windows`** alatt találhatod meg. Amennyiben a Windows telepítéskor más mappát használtál, válaszd azt!
3. Ebben a mappában rengeteg fájl található, mindegyik neve Windows-zal kezdődik, és a kiterjesztésük **`.winmd`**, amint az a 3-6 ábrán is látható. Ezek a fájlok reprezentálják a Windows Runtime API-k metaadatait.



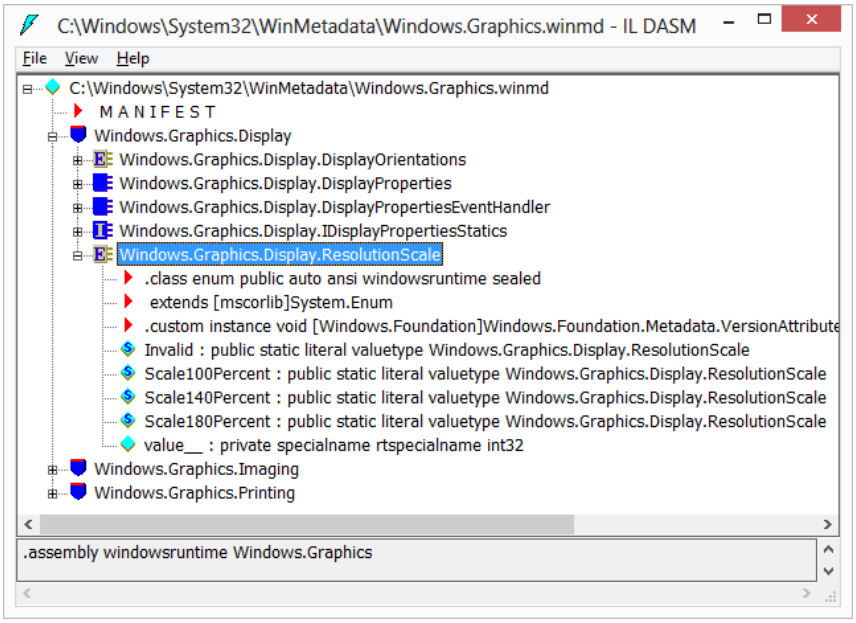
3-6 ábra: A Windows Runtime metaadatfájljai a System\WinMetadata mappában

4. Görgesd le a listát a **`Windows.Graphisc.winmd`** fájlig! A jobb egérgombbal kattints rá, és válaszd az Open With parancsot a gyorsmenüből! Egy felugró ablak jelenik meg a képernyőn, ahol egy alkalmazást rendelhetsz a **`.winmd`** fájltípushoz. Görgesd le a listát az aljáig, és válaszd a „Look for an app on this PC” parancsot, ahogyan azt a 3-7 ábra is mutatja!



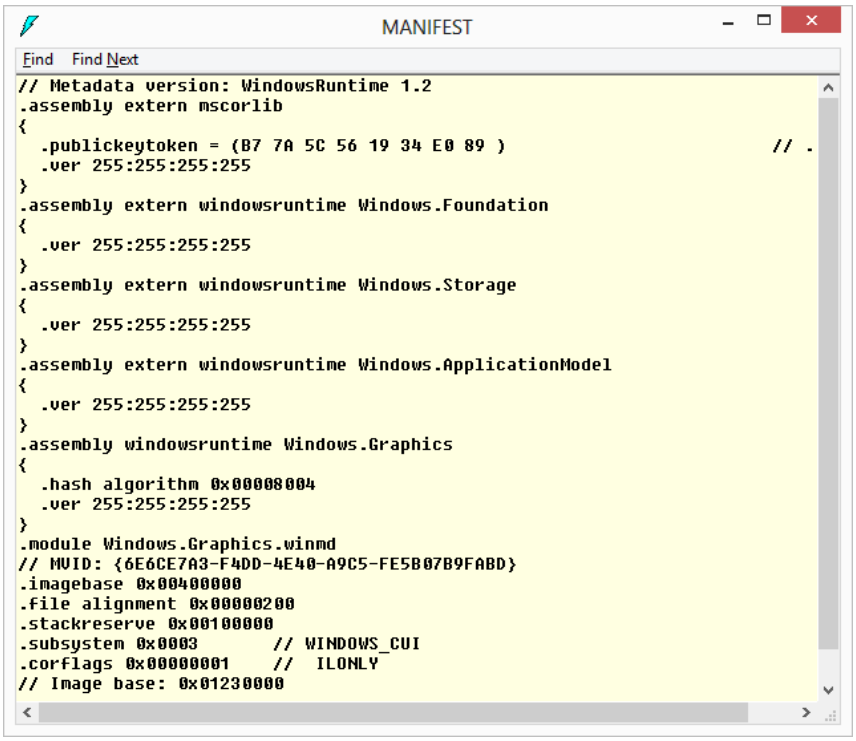
3-7 ábra: Egy alkalmazás hozzárendelése a `.winmd` fájltípushoz

5. Az Open With dialógus megjelenik a képernyőn. A File Name mezőbe írd be a **`C:\Program Files (x86)\Microsoft SDKs\Windows\v8.0A\bin\NETFX 4.0 Tools`** útvonalat (vagy navigálj oda)! A dialógus feltöltődik a mappában található fájlokkal. Görgesd le a listát az **`ildasm`** fájlig, válaszd ki azt, és kattints az Open gombra!
6. Az ILDASM megnyílik, és megjeleníti a **`Windows.Graphics.winmd`** fájl metaadat-hierarchiáját. Nyisd ki a **`Windows.Graphics.Display`** elemet, majd a **`Windows.Graphics.Display.ResulotionScale`**-t! Felfedezheted az ebben a fájlban definiált összes típust és a **`ResolutionScale`** felsorolt típus értékeit is, amint azt a 3-8 ábra mutatja.



3-8 ábra: A **`Windows.Graphisc.Display`** metaadat információja az ILDASM segédprogramban

7. Az ILDASM alkalmazás ablakában duplán kattints a MANIFEST elemre! Egy új ablak nyílik meg, amely a metaadatfájl ún. manifesztum információját jeleníti meg, amint azt a 3-9 ábra is mutatja.



3-9 ábra: A **`Windows.Graphics.Display`** fájl manifesztuma

A metaadatfájl manifesztuma fontos információt hordoz a fájlról és az abban leírt típusokról. A 3-9 ábrán láthatod, hogy a fájl egy **.module Windows.Graphics.winmd** bejegyzés, amelyet néhány további sor követ. Ezek a bejegyzések mind a metaadatfájlra vonatkoznak. Három bejegyzést is megfigyelhetsz, amelyek az **.assembly extern** definícióval kezdődnek, és ezek más metaadat információra hivatkoznak, amelyeket a **Windows.Graphics.Display** metaadatfájlban lévő definíciók használnak. Az első referencia az **mscorlib**, és ez a .NET CLR fő rendszerkomponense. A másik kettő, a **Windows.Foundation** és a **Windows.Storage** két másik **.windmd** fájlra hivatkozik, amint ez a **windowsruntime** módosító tagból kikövetkeztethető.

8. Nyiss meg más metaadatfájlokat is a **System32\WinMetadata** mappában, és vizsgáld meg a tartalmukat! Amikor végeztél, zárd be az összes nyitott ILDASM alkalmazáspéldányt!

Hogyan működik?

A **System\WinMetadata** mappa a rendszer működése szempontjából alapvető **.winmd** fájlokat tartalmaz. Az 5. lépésben a **.winmd** kiterjesztéshez hozzárendelted az ILDASM segédprogramot. A 6. lépésben belenéztél a **Windows.Graphics.Display.ResolutionScale** típus metaadataiba. A 7. lépésben megvizsgáltad, hogy a **.winmd** fájl manifesztuma hogyan hivatkozik külső függőségekre.

Névterek

Amint azt az előző gyakorlatban is megfigyelhetted, a Windows Runtime típusai hierarchikus neveket használnak. Például, a képernyőfelbontás skálaértékeit felsoroló típus teljes neve **Windows.Graphics.Display.ResolutionScale**. A név utolsó része (**ResolutionScale**) a típus egyszerű neve, az előtte álló névrészek pedig a névtér-hierarchiát alkotják. A hierarchia legfelső szintjén lévő névtér a Windows. Ez egy másik névteret, a **Graphics** névteret foglalja magába, amely pedig egy továbbit, a **Display** névteret ágyazza be.

> *A .NET, Java és C++ programozók számára már ismerős a névterek fogalma. A Windows Runtime pontosan ugyanazzal a szemantikai értelmezéssel használja a névtereket, mint ahogyan azt a .NET teszi.*

A névterek koncepciója nélkülözhetetlen azokban az esetekben, amikor alkalmazások készítése közben nagyon sok típust használsz fel. Ha ezeket a neveket egyszerűen egy nagy készletbe hajítanád be, nagyon nehéz lenne megtalálni azokat, és rájönni, hogy melyik típust is kell használni egy adott feladatra.

Egy másik probléma az elnevezésekből eredhet. Nem tudod garantálni, hogy senki más nem használja pontosan ugyanazt a típusnevet, mint te. Például, ha típusodat **Circle**-nek nevezed el, igen nagy esély van arra, hogy valaki más is használja ugyanezt a nevet. Ha egy UI komponenskészletet vásárolsz, az szintén használhat **Circle** nevű típust. Hogyan fogja az alkalmazásod tudni, hogy a forráskód egy adott helyén vajon a saját **Circle** típusodat szeretnéd használni vagy pedig a megvásároltat?

A névterek remek konstrukciót biztosítanak az objektumaid kategóriákba csoportosításához. Jól megtervezett névtér-hierarchiák használatával termelékenyebbé válhatsz, mert könnyebben megtalálhatod egy adott feladathoz az arra legjobban illeszkedő típusokat. Például, ha éppen képeket szeretnél megjeleníteni az alkalmazásban, valószínűleg először a **Windows.Graphics.Imaging** névteret fogod átvizsgálni, mert ennek neve azt sugallja, hogy itt létezik erre alkalmas típus.

A névterek segítenek az elnevezési konfliktusok elkerülésében is. Ha a saját típusaidat saját névtereidben helyezed el (például a **Circle** típust a **MyCompany.Shapes** névtérben), akkor azok nem fognak ütközni más programozók vagy más cégek saját típusaival.

A típusok teljes neve gyakran nagyon hosszú is lehet. Szerencsére a legtöbb nyelv, amely támogatja a névterek koncepcióját, egyúttal valamilyen megoldást is biztosít arra, hogy csak a típusok egyszerű nevét kelljen használnod.

A C# például a **using** direktívát használja a névterek feloldásának könnyítésére:

```
using Windows.UI.Xaml;


namespace MyAppNamespace
{
  class MyClass: UserControl
  {
    public MyClass()
    {
      // ...
      selectedItem = this.FindName(itemName) as ListBoxItem;
      // ...
    }
  }
}
```

A Visual Basic hasonló konstrukciót kínál az **Imports** kulcsszó segítségével:

```
Imports Windows.UI.Xaml

Namespace MyAppNamespace

  Class MyClass
    Inherits UserControl

    Public Sub New()
      ' ...
      selectedItem = TryCast(Me.FindName(itemName), ListBoxItem)
      ' ...
    End Sub
  End Class
End Namespace
```

Talán nem meglepő, hogy a C++ is hasonló koncepciót használ a **using namespace** direktívával:

```
using namespace Windows::UI::Xaml;

namespace MyAppNamespace
{
  class MyClass: UserControl
  {
    public MyClass()
    {
      // ...
      selectedItem = dynamic_cast<ListBoxItem^>(this->FindName(itemName);
      // ...
    }
  }
}
```

A **using**, **Imports** és **using namespace** konstrukciók a C#-ban, Visual Basic-ben és C++-ban arra utasítják a fordítóprogramot, hogy a neveket a megadott névterekben kell ellenőrizni. Ez a mechanizmus lehetővé teszi, hogy csak a **ListBoxItem** típusnevet kell leírni a programjaidban, mert a fordítóprogram ellenőrizni fogja a **Windows.UI.Xaml** névteret is. Ezek nélkül a konstrukciók nélkül a típus teljes nevét – **Windows.UI.Xaml.ListBoxItem** – le kellene írnod.

> *Természetesen a Windows Runtime névterek a JavaScript programokból is elérhetők. A JavaScript másfajta mechanizmust használ, amivel a 9. fejezetben ismerkedhetsz meg.*

Nyelvi leképzés

Minden egyes programozási nyelvnek megvannak a saját jellemzői. Az egyes nyelvek hívei általában az adott nyelv jellegzetességei miatt szeretnek azokkal dolgozni, mint például az olvashatóság, nagy teljesítmény, funkcionalitás, alacsony szintaktikai zaj és így tovább. Amikor Windows 8 stílusú alkalmazást készítesz, néhány nyelv áll a rendelkezésedre – különböző szintaxissal és szemantikai megközelítéssel. A Windows Runtime tervezői számára komoly kihívás volt lehetővé tenni a szolgáltatások elérését ennyire különböző nyelvekből.

A Nyelvi leképzés rétege felelős a Windows Runtime API-k adott nyelvi alakjára transzformálásáért. Ezeknek a transzformációknak köszönhetően a programozók az API-kat úgy használhatják, mintha azok az adott nyelv natív futásidejű könyvtárának részei lennének.

A Nyelvi leképzés működésének megértéséhez a legjobb mód néhány egyszerű mintapélda megtekintése. A **`Windows.Storage.Pickers`** névtérnek van egy **`FileOpenPicker`** típusa, amely lehetővé teszi egy fájl kiválasztását egy mappából. A C# programozási nyelv használatával a következő kódot használhatod a fájlok kiválasztására szolgáló dialógus konfigurálásához:

```
using Windows.Storage.Pickers;
// ...
FileOpenPicker openPicker = new FileOpenPicker();
openPicker.ViewMode = PickerViewMode.Thumbnail;
openPicker.SuggestedStartLocation = PickerLocationId.PicturesLibrary;
openPicker.FileTypeFilter.Add(".jpg");
openPicker.FileTypeFilter.Add(".jpeg");
openPicker.FileTypeFilter.Add(".png");
```

Ugyanezt a feladatot a JavaScriptben az alábbi kódrészlettel lehet leírni:

```
var openPicker = new Windows.Storage.Pickers.FileOpenPicker();
openPicker.viewMode = Windows.Storage.Pickers.PickerViewMode.thumbnail;
openPicker.suggestedStartLocation =
    Windows.Storage.Pickers.PickerLocationId.picturesLibrary;
openPicker.fileTypeFilter.replaceAll([".png", ".jpg", ".jpeg"]);
```

A félkövér betűkkel megjelölt azonosítók a Windows Runtime API típusokat jelölik. Ezek mindegyike nagybetűvel kezdődik, és az ún. Pascal-case stílust használják, a C#-ban és a JavaScriptben egyaránt. A dőlt betűvel szedett azonosítók a Windows Runtime típusok tagjai. Amint láthatod, a C# nyelvben ezek Pascal-case stílust használnak, a JavaScriptben pedig Camel-case stílust. Ezt a transzformációt a Nyelvi leképzés rétege végzi! Míg a C# nyelvi leképzése a tagok nevét Pascal-case stílusban jeleníti meg, a JavaScript nyelvi leképzése a Camel-case stílust használja – a JavaScript konvenciónak megfelelően.

> *A Pascal-case és a Camel-case stílusok használata azt jelenti, hogy az azonosítókat, amelyben eredetileg a szavakat szóköz választja el, úgy kapcsoljuk össze, hogy a szavak kezdőbetűit nagybetűvel írjuk az összetett azonosítókban. Míg a Pascal-case az első betűt is naggyal írja (lásd a „**`P`**”-t a „**`PictureLibrary`**”-ben), addig a Camel-case kisbetűvel indít (lásd a „**`v`**”-t a „**`viewMode`**”-ban).*

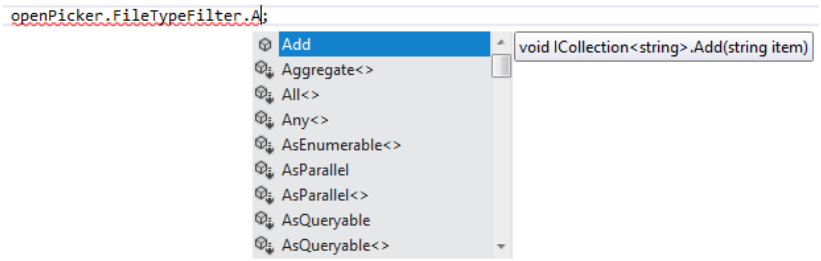
A Nyelvi leképzés réteg feladata nem ér véget egyszerűen ennél a szintaktikai konverziónál! A Windows Runtime API-ban a **`FileOpenPicker`** típus **`FileFilterType`** tulajdonsága **`string`** elemek egy vektora (tömbje). A C# nyelvi leképzése ezt a tulajdonságot **`List<string>`** típusra transzformálja, és így a fájlok kiterjesztésének bővítése a **`List<string>`** típus **`Add`** műveletével végezhető el:

```
openPicker.FileTypeFilter.Add(".jpg");
openPicker.FileTypeFilter.Add(".jpeg");
openPicker.FileTypeFilter.Add(".png");
```

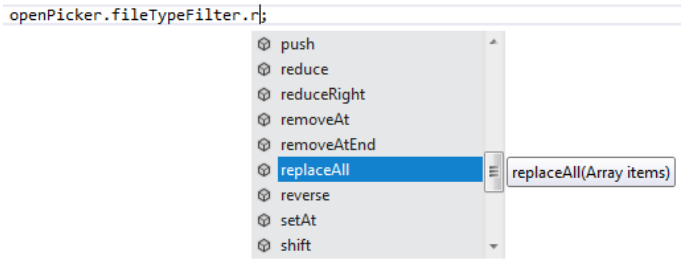
A JavaScript nyelvi leképzése a **`fileFilterType`** tulajdonságot **`string`** elemek tömbjére képzi le, és így lehetőség van a **`replaceAll`** függvény használatára a tömb tartalmának definiálására:

```
openPicker.fileTypeFilter.replaceAll([".png", ".jpg", ".jpeg"]);
```

A Visual Studio szintén épít a Nyelvi leképzés réteg képességeire. Mint azt a 3-10 ábra és a 3-11 ábra mutatják, az IntelliSense a C#-ban és a JavaScriptben más-más folytatási lehetőségeket kínál fel a nyelvtől függően.



3-10 ábra: Az IntelliSense a `List<string>` tagjait kínálja fel a C#-ban



3-11 ábra: Az IntelliSense a tömbműveleteket kínálja fel a JavaScriptben

A nyelvi leképzés segítségével nincs szükséged arra, hogy különböző eszközöket használj fel a különböző programozási nyelvek kapcsán az operációs rendszer szolgáltatásainak elérésére! A Windows Runtime szolgáltatásait minden nyelv esetében úgy érzékeled, mintha azok a kiválasztott programozási nyelv futásidejű környezetének részei lennének.

A Windows Runtime hasznossága

Amint azt már megtanultad, a Windows Runtime egy új modern API a Windows alapvető szolgáltatásai fölött, és a régi API-knál sokkal természetesebben lehet használni több programozási nyelvből is, úgymint C++-ból, C#-ból, Visual Basic-ből (.NET) és JavaScriptből. Ez a modernizálás lehetővé tette, hogy a Windows Runtime-on dolgozó mérnökök újragondolják, hogyan is kellene egy jelenkori operációs rendszernek az alkalmazásfejlesztést támogatni:

- A Windows Runtime egyszerű hozzáférést biztosít a hardver eszközökhöz, beleértve a GPS-t, a szenzorokat, a kamerát és más modern eszközöket – mindezt néhány kódsor leírásával. Míg a múltban néhány tucat kódsort le kellett írnod ahhoz, hogy az első „Hello, World” programodat létrehozd C nyelven a Windows 3.1 alatt, a Windows Runtime lehetővé teszi, hogy kb. öt kódsor leírásával képet készíthess a rendszeredhez kapcsolt kamerával.
- Az alkalmazások egy „homokozóban” (sandbox) futnak. Ez azt jelenti, hogy csak azok a műveletek hajthatók végre, amelyek az aktuális biztonsági kontextusban biztonságosnak tekinthetők. Például, ha egy alkalmazás szeretné bekapcsolni a mikrofont, ezt mindaddig nem tekinti a rendszer biztonságosnak, amíg a felhasználó explicit módon meg nem erősíti, hogy engedélyezi a mikrofon használatát.

- A régi Win32 API egy az operációs rendszertől független réteg volt. A Windows Runtime integráns része az operációs rendszernek, amely a fejlesztői élményre is fókuszál. A Windows Runtime nemcsak könnyebben használható, mint a Win32 API, de egyúttal sokkal stabilabb és fejlettebb is, gyorsabb memóriakezelés mellett kevesebb memóriát használ.
- A modern hardver eszközök és az azonnali reagálásra kész felhasználói felület nem működhetne az aszinkron programozási modell nélkül. A Windows Runtime natív módon támogatja az aszinkron műveleteket.

Ami nem része a Windows Runtime-nak

Mostanra már megtanultad, hogy a Windows Runtime a Windows 8 stílusú alkalmazások fejlesztésének sarokköve. Az operációs rendszer minden szolgáltatása elérhető a Windows Runtime felületén a C+, C#, Visual Basic és JavaScript nyelvekből. Mielőtt azonban egyenlőségjelet tennél a Windows 8 stílusú alkalmazások és a Windows 8 közé, tudnod kell, hogy a Windows 8 stílusú alkalmazások olyan más komponensekre is támaszkodhatnak, amelyek nem érhetők el a Windows Runtime-on keresztül!

A C és C++ nyelven írt alkalmazásokat a fordítóprogram közvetlenül CPU-specifikus gépi kódra fordítja, és ez közvetlenül végrehajtható a CPU-n. Az ilyen alkalmazások közvetlenül elérhetik azokat az operációs rendszer komponenseket, amelyek a felhasználói felület megrajzolásáért, az input eszközök és a szenzorok kezeléséért, a GPU-val való kommunikációért és még számtalan fontos dologért felelősek. A legtöbb ilyen komponens extra értéket adhat a Windows 8 stílusú alkalmazásokhoz. Például a játékprogramozók támaszkodhatnak a DirectX API-k (Direct2D, Direct3D, DirectWrite, XAudio2, XInput) nagy teljesítményű grafikus és audió képességeire.

> *A Microsoft DirectX multimédia és játékprogramozási API-k gyűjteménye. Ezek az API-k jó néhány addicionális réteget helyettesítenek az alkalmazás és a nagy teljesítményt kívánó multimédia- és játékalkalmazások funkcióit megvalósító hardver között. A Direct2D egy kétdimenziós grafikus API, a Direct3D a háromdimenziós játékfejlesztés eszköze. A DirectWrite egy szövegmegjelenítő technológia, amely először a Windows Vistában jelent meg, és a Windows 8-ban is elérhető.*
>
> *Amikor a Microsoft a saját játékkonzolja kifejlesztéséhez kezdett hozzá, az „X"-et az Xbox névben annak jelzésére alkalmazta, hogy a konzol a DirectX technológiát használta.*

Ezek az API-k csak egy vékony réteget jelentenek az alkalmazás és az általa elért hardver között, és alacsony szintű adatstruktúrákat használnak az alkalmazás és a hardver közötti információk mozgatására. Bár a Windows Runtime-on keresztül elérhető szolgáltatások biztonságosak a felhasználói felület válaszképességét illetően, a DirectX API-k a teljesítményre lettek hangolva, és nagyobb felügyeletet kívánnak az alkalmazáson belül, mint a Windows Runtime szolgáltatásai.

A C++ használatával olyan Windows 8 stílusú alkalmazásokat készíthetsz, amelyek fel tudják használni a DirectX képességeit. Például a DirectWrite segítségével nagyszerű kalligrafikus szöveget készíthetsz, amint azt a 3-12 ábra is mutatja.

3-12 ábra: Egy Windows 8 stílusú alkalmazás a DirectWrite API-t használja

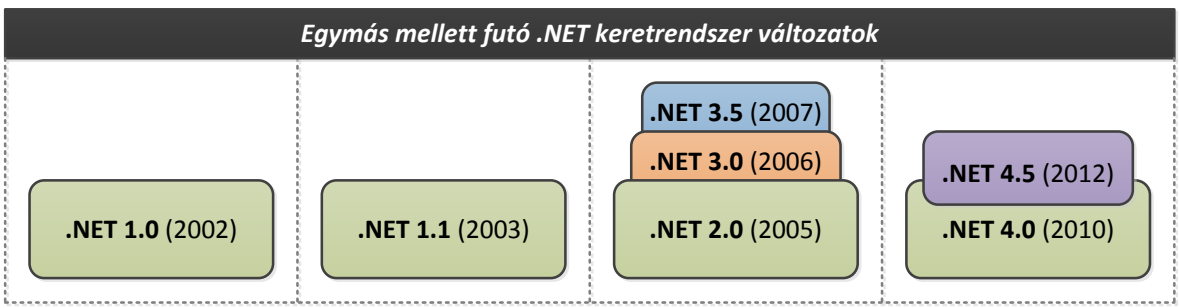
Bár a DirectX komponensek nem érhetők el a Windows Runtime-on keresztül, és közvetlenül csak a C++ nyelvből férsz hozzájuk, a C# és Visual Basic programok felügyelt kódból felhasználhatják ezeket az API-kat. A könyv írásakor már megjelent néhány nyílt forráskódú kezdeményezés a CodePlex-en (http://www.codeplex.com), amelyek a Windows 8-as DirectX elérhetőségét célozták meg C#-ból. Bár ezek a projektek még csak kezdeti állapotban vannak, néhány hónap alatt rengeteget fejlődhetnek.

A .NET keretrendszer 4.5-ös változata

Két programozási nyelv, amely támogatja a Windows 8 stílusú alkalmazásokat – a C# és a Visual Basic – a .NET keretrendszer fölött futnak. A Windows 8 a .NET keretrendszer egy új változatával, a 4.5-tel együtt jelent meg. A fejlesztői platformot leíró kép nem volna teljes a .NET újdonságainak, illetve a Windows Runtime-mal való együttműködésének megértése nélkül.

A .NET keretrendszer 4.5 változatának telepítési modellje

2002 óta (amikor is a .NET 1.0 megjelent) jó néhány módja volt annak, ahogyan a .NET keretrendszer kibocsátott változatai a korábbi verziókkal együttműködtek, amint azt a 3-13 ábra is illusztrálja. Bár az első három változat (1.0, 1.1, 2.0) különböző CLR-ekkel jelent meg, amelyek egymás mellett működtek ugyanazon a számítógépen, az ezt követő változatok már más utat követtek.



3-13 ábra: A .NET keretrendszer változatainak kapcsolata

Amikor 2006-ban a .NET 3.0-s változatát kibocsátották, az nem telepített saját CLR-t. Egyszerűen egy kiegészítés volt a 2.0-hoz, és annak futásidejű magját használta. Ugyanez történt a .NET 3.5-tel 2007-ben, az egy kiegészítés volt a .NET 3.0-hoz, és még mindig a .NET 2.0-val kibocsátott CLR-t használta.

2010-ben a .NET 4.0 újra saját CLR-rel jelent meg, beágyazva a .NET 3.5 BCL egy modernebb változatát. Elvileg 2010-ben akár négy különböző, egymás mellett élő CLR változatot is telepíthettél a számítógépre: a CLR 1.0-t, a CLR 1.1-et, a CLR 2.0-t és a CLR 4.0-t. A .NET keretrendszer legfrissebb változata, a 4.5 még egy kicsit csavart a korábban használt telepítési modellen. Ahelyett, hogy egy új CLR-t adott volna a gépen lévőkhöz, vagy egy új BCL-t épített volna a 4.0 tetejére, a .NET 4.5 egy beépülő módosítás (nemcsak új fájlokat ad a keretrendszerhez, hanem át is ír meglévőket). Ez magát a CLR 4.0-t is megváltoztatja, és egyúttal ki is bővíti a BCL korábbi változatát.

Egy az operációs rendszerbe vagy a :NET CLR-be beépülő módosítás mindig nagy kockázatot jelent a kompatibilitás szempontjából. A .NET keretrendszer 4.5-ös változatának kibocsátásakor a Microsoft felvállalta ezt a kockázatot, és rengeteg energiát ölt a potenciális kompatibilitási problémák feltárásába és kiküszöbölésébe. Egy kompatibilitási laboratóriumot állítottak fel, végignézték az összes korábbi hibajavítást, és változtatások nélkül lefuttatták a korábbi teszteseteket.

Windows Runtime integráció

A .NET CLR mindig is tartalmazott régebbi technológiákhoz kapcsolódó integrációs képességeket, mint például a COM és a P/Invoke (hozzáférés a Win32 API-hoz). A .NET keretrendszer 4.5-ös változatában a CLR natív módon integrálódik a Windows Runtime-mal. A C# és Visual Basic programozók már ugyanazzal a könnyedséggel használhatják a Windows Runtime típusokat, mintha azok .NET típusok lennének.

A Windows Runtime típusai natív típusok. Valójában azok a COM egy új, modern változatát használják, amely már nem követeli meg az objektumregisztrációt, de a műveleteik közvetlenül nem hívhatók felügyelt kódból. Mindazonáltal a .NET CLR és az egyes nyelvek fordítóprogramjai mindent elintéznek a kulisszák mögött, és így nem kell a natív Windows Runtime objektumok felügyelt kódból való meghívásával bajlódnod.

Ha visszalapozol a 3-2 és 3-3 ábrákhoz, láthatod, hogy az asztali és a Windows 8 stílusú alkalmazások rétegei egyaránt tartalmaznak egy .NET Runtime dobozt. Ebből esetleg azt gondolhatod, hogy két különböző .NET keretrendszer is létezik, egy az asztali, egy másik pedig a Windows 8 stílusú alkalmazásokhoz. Szerencsére, csak egy van.

A Windows 8 stílusú alkalmazások (Windows Runtime alkalmazások) számára elérhető BCL azokra a szolgáltatásokra van korlátozva, amelyek „biztonságosnak" tekinthetők. Mikor egy már meglévő .NET forráskódot fordítasz, annak egyes részei esetleg nem fordulnak le, mert olyan .NET BCL műveleteket próbálsz használni, amelyek nem érhetők el a Windows 8 stílusú alkalmazásokban, vagy nem tekinthetők „biztonságosnak".

Általánosságban, minden művelet biztonságosnak tekinthető, amely nem veszélyezteti a felhasználói felület azonnali reagálásának képességét.

Aszinkron műveletek támogatása

Amint azt már többször is olvashattad, az aszinkron programozás alapvető technika a felhasználói felület válaszképességének biztosításához, és ennek megfelelően a Windows Runtime-ot az aszinkron modell figyelembevételével tervezték. A Windows Runtime tervezési eredményei inspirálták a .NET keretrendszer csapatát, és a keretrendszer rengeteg komponensét kiegészítették az aszinkron műveletek támogatásával. A .NET BCL-je több száz ilyen változtatást és továbbfejlesztést tartalmaz, ezek közül talán az alábbiak a legjelentősebbek:

- A BCL alapvető interfészei (pl. az I/O műveletekhez kapcsolódók) támogatják az aszinkron műveleteket.
- A ADO.NET adatelérési eljárásai és a hálózati kommunikációs komponensek a WCF-ben szintén alapvető műveletekként kezelik az aszinkron konstrukciókat.

- Az ASP.NET is támogatja az aszinkron feldolgozási sorokat a kérések értelmezésénél és a válaszüzenetek létrehozásánál. Az ASP.NET MVC 4 aszinkron kontrollerek létrehozását is lehetővé teszi.
- A Task Parallel Library – amelyet a .NET 4.0 vezetett be – már eleve az aszinkron programozási modell támogatásával született. A .NET 4.5-ben ennek szálkezelését továbbfejlesztették új szinkronizációs típusokkal, illetve olyanokkal, amelyek az időtúllépés különböző forgatókönyveit kezelik.

Bár az aszinkron programozási minta használata a felhasználói felület válaszképességét jelentősen növeli, és a műveletek teljesítményét is fokozza, igen nehéz lehet a használata, és sok hibázási lehetőséget is hordoz magában. A .NET 4.5-ben a C# és Visual Basic fordítóprogramok két új kulcsszót (**`async`** és **`await`** a C#-ban, **`Async`** és **`Await`** a Visual Basic-ben) is biztosítanak, amely leveszi ezeket a nehézségeket a programozó válláról.

Egyéb újdonságok

Bár a .NET keretrendszer 4.5-ös változatának legfontosabb újdonsága a Windows Runtime integráció és az aszinkron programozás támogatása, számtalan apró továbbfejlesztés teszi a fejlesztőket produktívabbakká. A .NET 4.5-ben rengeteg újdonság van, amelyek ismertetése meghaladná ennek a könyvnek a kereteit. Amennyiben további információhoz szeretnél ezek kapcsán jutni, indulj el a .NET Framework Development Center honlapjáról az MSDN-en (http://msdn.microsoft.com/netframework)!

A projektekre illeszkedő technológia kiválasztása

Ebben a fejezetben Windows 8 alkalmazások fejlesztésére alkalmas programozási nyelvek és technológiákat ismerhettél meg. Néha a legnehezebb döntés egy projekt kapcsán a megfelelő programozási nyelv és technológia kiválasztása. A fejezetnek ez a része az optimális döntés kialakításában igyekszik segítséget nyújtani neked.

A programozási nyelvek és a fejlesztési technológiák mellett van egy másik – korábban alig említett – szempont is. A Windows 8 nemcsak a felhasználók alkalmazáskezelési szokásait változtatja meg, hanem azt is, hogyan fedezik fel, telepítik és távolítják el az alkalmazásokat. Ez a modell arra ösztönözhet téged, hogy Windows 8 stílusú alkalmazásokat hozz létre az asztali alkalmazások helyett, szóval épp itt az idő, hogy megismerkedj vele.

A Windows Store

Az előző Windows operációs rendszerek használata során telepítőkészleteket kellett létrehoznod, és azokat a cél számítógépeken lefuttatni, hogy alkalmazásaid ott használhatók legyenek. A felhasználóknak ehhez tisztában kellett lenniük a telepítési folyamat néhány lépésével, mint például a célmappa kiválasztásával, a szükséges előfeltételek telepítésével, parancsikonok hozzáadásával és így tovább. Gyakran a telepítési folyamat egyúttal a félelem forrása is volt – vajon nem ír-e át valamit a telepítőeszköz, amitől a korábban futó alkalmazások egyszer csak megszűnnek helyesen működni? Az időközben szükségtelenné vált alkalmazásokat el kellett távolítani, és gyakran segédprogramokra volt szükség az alkalmazások után maradt „szemét” eltávolítására.

A fogyasztócentrikus megközelítés nem működhet az alkalmazások telepítésének és eltávolításának radikális leegyszerűsödése nélkül! A Windows 8 stílusú alkalmazásokat a Windows Store-ból (ez online szolgáltatás) lehet telepíteni. Fejlesztőként feltöltheted alkalmazásodat a Windows Store-ba, ahol az egy jóváhagyási folyamaton megy keresztül, mielőtt a felhasználók megtalálhatják és telepíthetik. Felhasználóként a Windows Store-ban alkalmazásokat kereshetsz, majd a megvásárlásuk után (esetleg az ingyenes kipróbálásuk után) az operációs rendszer azt automatikusan telepíti. Amennyiben már nincs szükséged az adott alkalmazásra, a Windows 8 eltávolítja azt, és gondoskodik a takarításról, vagyis minden az alkalmazás által használt erőforrást (fájlok, bejegyzések stb.) felszabadít.

> *Természetesen a Windows Store-ban találhatsz ingyenesen letölthető alkalmazásokat is.*

A Windows Store használata a fogyasztók számára rendkívül egyszerű. Azonban a fejlesztőknek sokkal több információra van szükségük alkalmazásaik feltöltéséhez és ahhoz, hogy azokkal ténylegesen pénzt is kereshessenek. A 14. fejezet teljes egészében ezt a témát tárgyalja.

Windows 8 stílusú vagy asztali alkalmazások?

Az első dolog, amelyről döntést kell hoznod az, hogy vajon Windows 8 stílusú vagy asztali alkalmazást kívánsz létrehozni? A 3-2 táblázat néhány fontos szempontot tartalmaz, amely segíthet ennek a kérdésnek a megválaszolásában.

3-2 táblázat: Szempontok a Windows 8 stílusú és asztali alkalmazások közötti választáshoz

Használj Windows 8 stílusú alkalmazást, ha…	Használj asztali alkalmazást, ha…
Kevés gyakorlati tapasztalatod van a Windows programozásában.	Nagyobb tapasztalatod van a Windows programozásában, sőt, tapasztalt Windows programozó vagy.
Olyan alkalmazások létrehozására fókuszálsz, ahol a felhasználói élménynek kiemelt szerepe van.	Olyan alkalmazásokat igyekszel létrehozni, amelyek fejlesztésénél a már ismert UI technológiákra (pl. Windows Forms, WPF, Silverlight) támaszkodhatsz.
Kevés a már kész és újrahasznosítható UI-specifikus kódbázisod.	Nagy mennyiségű, a felhasználói felülethez kapcsolódó kódod van, és az abban rejlő tudást szeretnéd újrahasznosítani.
Az alkalmazásod egyetlen számítógépen fut. Elsődleges fókusza, hogy felhasználói felületet biztosítson interneten (vagy a céges hálózaton) keresztül elérhető szolgáltatásokhoz és távoli komponenseken.	Az alkalmazásod néhány komponensre van szétosztva, beleértve a felhasználói felületet, az üzleti szolgáltatásokat, adatbázisokat. Ezek a komponensek megörökölt, gyártó-specifikus, esetleg régimódi kommunikációs technológiákat használnak egymás szolgáltatásainak elérésére.
Az alkalmazásod mobil eszközök, táblagépek speciális képességeit szándékozik kihasználni, hogy kiemelkedő felhasználói élményt nyújtson.	Az alkalmazásod elsődlegesen asztali számítógépeken fut, ott lévő alkalmazásokkal integrálódik.
Az alkalmazásod szeretné a Windows Store által kínált egyszerű telepítési modellt kihasználni.	A Windows Store-ban lévő telepítési modellnél sokkal bonyolultabbra van szükséged.
A C++, C#, Visual Basic vagy JavaScript programozási nyelveket szeretnéd az alkalmazás elkészítéséhez használni.	A C++, C#, Visual Basic és JavaScript nyelvek mellett alkalmazásaid erősen támaszkodnak más programozási nyelvre is.

Nyilvánvalóan, amikor egy komplex projektben veszel részt, nem tudod a projekt egészén a Windows 8 stílusú alkalmazások technológiakészletét használni – gyakran szerveroldali fejlesztésre is szükséged van. Mindazonáltal megéri megvizsgálni rendszered felhasználói felület rétegéhez tartozó komponenseit, elképzelhető, hogy azokat meg tudod valósítani Windows 8 stílusú alkalmazásokként.

Programozási nyelv választása

Amint azt korábban már számtalanszor említettük, a Windows 8 stílusú alkalmazáshoz a C++, C#, Visual Basic és JavaScript nyelveket használhatod. Ha bármelyiküket is előnyben részesíted a többivel szemben, ne habozz, kezdd el azt a nyelvet használni!

Amennyiben nincs vagy nagyon kevés tapasztalatod van csak a Windows programozásban, esetleg bizonytalan vagy abban, hogy melyik nyelvet a legkönnyebb elsajátítanod, itt van néhány tipp:

- Ha már van weboldal tervezési és készítési gyakorlatod, valószínűleg ismered a HTML-t és tapasztalatod is van a CSS-sel és a JavaScripttel. Kezdd el Windows 8 stílusú alkalmazásaid programozását JavaScripttel!

- Ha korábban használtad már a makrókat a Microsoft Wordben vagy az Excelben, segíthet, ha a Visual Basic nyelvvel indítasz.
- Ha tapasztalatod inkább a programozási algoritmusok terén van, és kevésbé a felhasználói felülethez kapcsolódik, akkor a C++ vagy a C# lehet a legjobb választás. A C#-ot valószínűleg könnyebb megtanulni, a C++ pedig több kontrollt biztosít az alacsony szintű konstrukcióin keresztül.

Az egyik legjobb dolog a Windows 8 stílusú alkalmazások készítésében, hogy nem szükséges csak egyetlen nyelvhez ragaszkodnod! Akár több programozási nyelvet is használhatsz egy alkalmazáson belül! Például, ha már van webes programozási tapasztalatod a JavaScripttel, a felhasználói felületet elkészítheted HTML-lel és JavaScripttel, de mégis felhasználhatsz bonyolultabb alkalmazáslogikát, amelyet C#-ban írtál meg, illetve C++-ban készített nagy teljesítményű algoritmusokat.

Még akkor is, ha ezek közül csak egyetlen programozási nyelvvel is kezdesz hozzá az alkalmazásfejlesztéshez, megéri másik nyelveket is megtanulni, hiszen mindegyiknek megvan a saját erőssége. Ha általában szoros határidőket kell tartanod, illetve nyitottabbnak kell lenned más platformok irányába, több nyelv ismerete, azok célszerű vegyítése felbecsülhetetlen értékű lehet.

Összegzés

A Windows 8 új alkalmazásfejlesztési modellt kínál, amelyet Windows 8 stílusú alkalmazásmodellnek nevezünk. Az új stílus mellett asztali alkalmazások készítésére továbbra is használhatod a korábbi Windows változatoknál megismert technológiakészletet.

A Windows 8 stílusú alkalmazások saját technológiakészlettel rendelkeznek. Négy programozási nyelv – C++, C#, Visual Basic és JavaScript – használható a készítésükhöz. A választott nyelvtől függetlenül az operációs rendszer minden szolgáltatása, amelyekre ezeknek az alkalmazásoknak szüksége lehet, elérhető egy új komponensen, a Windows Runtime-on keresztül – és ebből a szemszögből ezek a programozási nyelvek egyenrangúak.

A Windows Runtime modern programozási felület az alapvető Windows szolgáltatások felett. Bár natív kódban van megvalósítva, névtereket, metaadatokat és aszinkron műveleteket biztosít. Minden programozási nyelvhez kapcsolódik egy Nyelvi leképzés réteg, amely úgy viselkedik, mintha a Windows Runtime az adott nyelven valósulna meg.

A C# és a Visual Basic a .NET keretrendszer 4.5-ös változatát használják, amely a Windows Runtime komponenseket natív módon integrálja, és így az ott található típusok ugyanolyan egyszerűen használhatók, mint azok, amelyek a .NET BCL-ben vannak.

4. Bevezetés a Windows 8 aszinkron programozásába

Ebben a fejezetben az alábbi témákat ismerheted meg:

- A párhuzamosítás és az aszinkronitás közötti különbség
- A szinkron és aszinkron műveletek futtatása közötti különbség
- Az aszinkron műveletek alkalmazási lehetőségei
- A Task Parallel Library alapvető képességei

A szinkron modell

Az objektum-orientált programok, ha végletekig lecsupaszítjuk őket, voltaképpen metódushívások sorozatai. Még az olyan programozási konstrukciók is, mint az eseménykezelés, mely látszólag kilóg ebből az általánosításból, visszavezethetők arra, hogy „valaki" – egy objektum – meghív egy metódust, mely azután további metódusokat hív meg.

Egy-egy ilyen metódushívásnál a futtatókörnyezet megállítja a hívó metódust, amely addig várakozik, míg a meghívott metódus vissza nem tér, azaz le nem futtatta saját utasításait. Ez a korábbi programozási paradigmákból átvett konstrukció biztosítja, hogy programjainkat utasításszekvenciák átlátható csoportjaiba rendezhetjük.

Ennek viszont az a következménye, hogy amíg egy meghívott metódus dolgozik, az őt meghívó metódus várakozni kényszerül. Első pillantásra ez nem tűnik problémának: amíg várunk a végeredményre, úgysem tudnánk tovább dolgozni. Szinkronban kell maradnia a hívó és a hívott metódusnak, különben a program futása közben érvénytelen állapotú objektumokat hozhatna létre. Azonban nem mindig ez a helyzet!

Vegyük a következő példát: egy grafikus felhasználói felülettel (a továbbiakban: GUI) rendelkező alkalmazást építünk, mely egy gombnyomás hatására egy vagy több képet tölt le az internetről, majd ezeket a helyi számítógépen feldolgozza.

Az első kérés elküldése után a program várakozni kezd arra, hogy választ kapjon a webkiszolgálótól. Ez ahhoz vezet, hogy az alkalmazás válaszkészsége nullára csökken (közkeletű kifejezéssel élve: a program megfagy) egészen addig, míg meg nem kapta a választ a szervertől. Ha ebben az állapotában megpróbálunk új utasításokat adni az alkalmazásnak – például megnyomunk egy gombot, áthelyezzük vagy átméretezzük az ablakot –, az utasításokat az csak a szerver válasza után fogja végrehajtani, gyors egymásutánban. Vagyis az alkalmazás továbbra is figyeli, hogy mit tesz a felhasználó, de a GUI reagálni csak akkor tud a felgyülemlett utasítások halmazára, amikor az alkalmazást blokkoló – szinkron módon meghívott – metódus visszatért.

Mi a helyzet akkor, ha az alkalmazás nemcsak egy, hanem több képet is letöltene és feldolgozna?

Az alkalmazás elindítja az egyik letöltést, majd várakozik. Amikor megérkezett a kért kép, feldolgozza. Ezután elindítja a következő letöltést, és megint várni kezd, amikor pedig megérkezett a válasz, feldolgozza azt is.

Talán érezhető, hogy itt már nemcsak a GUI válaszképességének csökkenése a probléma, hanem a skálázhatóság teljes hiánya is. Miért nem indítja el az alkalmazás egyszerre az összes kép letöltését, miért várja meg, míg az előző kép megjött a szerverről és átesett a feldolgozáson? Sokkal logikusabb, időtakarékosabb lenne, ha „egyszerre" jelezne az alkalmazás mindegyik szervernek, hogy küldjék a képeket, majd azokat a beérkezés sorrendjében dolgozná fel! Ha szerveroldali kódban gondolkodunk, ahol egyetlen

alkalmazásnak kellene kiszolgálnia több ügyfelet, komoly problémát jelent az, hogy a program egyes részei idejük nagy részét felesleges várakozással töltik.

Amennyiben tudjuk, hogy mind a letöltés, mind a feldolgozás másodperceket vesz igénybe, miért nem próbáljuk úgy végrehajtani őket, hogy

- egyrészről ne befolyásolják a GUI válaszképességét (vagyis továbbra is azonnal elfogadja és végrehajtsa például a gombnyomásokat az alkalmazás),
- másrészt pedig ne rontsák a skálázhatóságot (vagyis ha egy kép letöltése t időt vesz igénybe, akkor **n** darab képnél ez ne **n*t** legyen, hanem kevesebb, lehetőleg **max(t)** körüli)?

A probléma nem új, már régóta létezik, éppen ezért már számos megoldás született rá. A .NET keretrendszer – mely szegről-végről a Windows Runtime idősebb testvérének tekinthető – alapvetően a szál (*thread*) fogalmának bevezetésével oldotta meg. Programjaink felhasználói felületét egy szál futtatja, ha ezen egy metódust hívunk, az blokkolni fogja a GUI-t. Viszont lehetőségünk van arra, hogy más szálakat is létrehozzunk, amelyek időosztással, párhuzamosan futnak a GUI-t vezérlő szállal. Ha egy metódust egy ilyen különálló szálon futtatunk, az nem fog érzékelhető módon kihatni a felhasználói felület válaszkészségére. Ezzel a problémát megoldottuk!

A szálak közvetlen használata viszont több szempontból hátrányos. Létrehozásuk időigényes, futásukkor a processzor folyton váltani kényszerül közöttük (időosztás), emiatt kevesebb idő jut a tényleges üzleti logika végrehajtására, vagyis összességében lassulhat a program. Használatuk pedig más technikákhoz képest nehézkes lehet, és könnyű velük hibázni – elég, ha a szinkronizálásra, illetve a GUI-t futtató szálra történő visszahívásra gondolunk. (A saját szálakról nem módosíthatjuk közvetlenül a felhasználói felület elemeit.) Éppen ezért a modern alkalmazásfejlesztői keretrendszerek módot adnak arra is, hogy más, specializáltabb módszerekkel érjük el a többszálúságot vagy *aszinkronitást*.

Az egyik ilyen lehetőség a Task objektumok használata, amelyet a későbbiekben ez a fejezet is bemutat. Egy másik lehetőség az *aszinkron programozási modell* alkalmazása. Az alábbiakban áttekintjük, hogy ez hogyan is működött a C# 5.0 megjelenése előtt.

> *A szálak közvetlenül nem érhetőek el a Windows Runtime keretrendszerben, ezért ezekkel itt bővebben nem foglalkozunk.*

Az aszinkron programozási modell áttekintése

A .NET keretrendszerben az aszinkron programozási modellhez kapcsolódva két programozási mintát szokás megkülönböztetni. Az egyik többféle várakozási módszerrel is kompatibilis (ezekről a későbbiekben szó lesz), tehát valamivel flexibilisebbnek tekinthető. A másik, később elterjedt módszer pedig valamivel egyszerűbb: csak egy várakozási módszert tesz elérhetővé.

Közös bennük, hogy mindegyik egy-egy szállal oldja fel a szinkron hívások kényszerű várakozásának problémáját, és egyúttal előnyt is jelent a közvetlen szálhasználatoz képest. Először is a szálat a program indulásakor létrejövő szálak közül választja, így csökken az indulási költség – gyorsabban elindulhat a végrehajtás. Másodszor: a szinkronizálás kevésbé problémás; kérhetünk visszajelzést arról, hogy befejeződött-e a végrehajtás, és a megfelelő pillanatban azonnal folytathatjuk a munkát az akkor már rendelkezésre álló adatokkal. Harmadszor: a fejlesztő mentesül a szálak kezelésének nehézségei alól, minden „automatikusan" történik a háttérben.

Aszinkron programozás az APM segítségével

Az APM (*Asynchronous Programming Model*) a két modell közül a korábbi. Tekintsük az alábbi egyszerű, szinkron hívható metódust:

```
string DoSomething(int i)
{
    //…
}
```

Az APM használatakor ezt a metódust két másikra bontjuk szét, az alábbiak szerint:

```
IAsyncResult BeginDoSomething(int i, AsyncCallback callback, object userState)
{
    //…
}

string EndDoSomething(IAsyncResult result)
{
    //…
}
```

A szabályok:

- Az eredeti metódus neve elé a **`Begin`** és **`End`** prefixeket tesszük.
- A **`Begin`** metódus visszatérési típusa mindig egy **`IAsyncResult`**, paraméterlistájának eleje pedig megegyezik az eredeti metóduséval. Két plusz paramétert kap: egy **`AsyncCallback`** és egy **`object`** típusút.
- Az **`End`** metódus visszatérési típusa megegyezik az eredeti metódus visszatérési típusával. Paraméterlistáján egyetlen **`IAsyncResult`** típusú paramétert fogad.

A **`Begin`** metódus hívásával indíthatjuk el a feldolgozást, ami a háttérben fut. A **`Begin`** metódus azonnal visszatér, tehát a hívó metódus folytathatja a munkát, amíg az elindított feldolgozás a háttérben fut. Az **`End`** metódus hívásakor pedig megkapjuk a feldolgozás eredményét. Ha az **`End`** metódust a végrehajtás befejezése előtt hívjuk meg, az blokkolja a hívó metódust a feladat elvégzéséig.

Arra, hogy mikor hívjuk az **`End`** metódust, három bevárási módot (idegen kifejezéssel *rendezvous technique*) alkalmazhatunk. Az első és legegyszerűbb a „várakozás a befejezésig” (*wait-until-done*) modell. Ebben a **`Begin`** metódus hívását a későbbiekben követi az **`End`** metódus hívása. A két hívás közötti utasítások a **`Begin`** által indított feldolgozással párhuzamosan futnak.

A második módszer a folyamatos lekérdezés (*polling*). Ezzel a módszerrel a feldolgozás elindítása után ciklikusan ellenőrizzük, hogy fut-e még a háttérben a feldolgozás. Előnye az előző módszerrel szemben, hogy addig dolgozhatunk, amíg elő nem állt az eredmény, biztos nem fog blokkolni az **`End`** metódus hívása.

A harmadik, talán legelterjedtebben használt módszer pedig a visszahívás (*callback*). Ennél a **`Begin`** metódus híváskor megadunk annak egy delegáltba csomagolt másik metódust, illetve opcionálisan egy objektumot saját használatra – itt nyer értelmet a **`Begin`** metódus utolsó két paramétere. Amikor előállt a végeredmény, a **`Begin`** implementációja automatikusan meghívja (ez a visszahívás) az átadott metódust, és ennek az átadott metódusnak a törzsében hívhatjuk meg a **`Begin`**hez tartozó **`End`** metódust. Látható, hogy ez a megoldás nyújtja a legnagyobb flexibilitást, hiszen nem egy merev szekvencia szerint hajtjuk végre a programot, hanem automatikusan férünk hozzá az eredményhez.

Aszinkron programozás az eseményalapú modell segítségével

Az APM mellett egy másik modell, az eseményalapú aszinkron programozás (*Event-based Asynchronous Programming*) nevű terjedt el a .NET-ben és a később megjelent Microsoft-keretrendszerekben. A modell nagyon hasonlít az APM visszahívásos technikájához, itt azonban nem két metódussal találkozhatunk, hanem egy indító metódussal és egy olyan eseménnyel, mely a háttérben futó feldolgozás befejezéséről tájékoztat. Az eredményt az esemény második argumentuma hordozza.

A korábbi példával élve a szinkron metódus így alakítható át EAP-nak megfelelő aszinkron metódussá:

```
void DoSomethingAsync(int i, object userState)
{
    //…
}

event EventHandler<DoSomethingEventArgs> DoSomethingCompleted;
```

Ehhez természetesen egy **DoSomethingEventArgs** nevű osztálynak is léteznie kell.

Ennek a mintának a szabályait így lehet összefoglalni:

- A metódus nevét úgy kapjuk, hogy az eredeti metódus neve mögé az **Async** végződést tesszük.
- A metódus visszatérési típusa **void**, vagyis nem ad vissza semmit sem. Paraméterlistája egy **object** típusú paraméterrel egészül ki: ebben utaztathatunk bármilyen saját adatot az indító metódustól az eseményig.
- Az esemény nevét úgy kapjuk meg, hogy az eredeti metódus neve mögé a **Completed** végződést tesszük.
- Az esemény második argumentumába csomagolva kapjuk vissza az eredeti metódus visszatérési értékét.

A .NET keretrendszer 4.0-ás változatának 2010-es megjelenése után kialakult egy harmadik aszinkron programozási minta, a taszk-alapú aszinkron programozási modell (*Task-based Asynchronous Programming model* – TAP). A fejezet második felét e programozási minta bemutatásának szenteljük.

Az APM használata

A következőkben egy élő példán szemléltetjük, hogyan készíthető szinkronból aszinkron kód az APM segítségével. A tömörség megőrzése érdekében a téma szempontjából lényegtelen részleteket elhagyjuk.

> *A példa nem a Windows Runtime-ot használja fel, az a .NET keretrendszer 4.5-ös verzióján, azon belül pedig a WPF-en fut. Erre azért van szükség, mert a Windows Runtime-ot eleve aszinkron használatra készítették, ezért rengeteg helyen nincs is szinkron párja az aszinkron metódusoknak, így ott nem lehetne az APM mintát bemutatni.*

Az alábbi kód három szinkron metódus szekvenciális hívását szemlélteti, valamint minden metódus lefutása után frissíti az alkalmazás felhasználói felületét. Az első metódus letölt egy RSS feedet egy megadott URL-ről (**GetFeedString**), a második értelmezi a visszakapott karakterláncot, és CLR-objektumokká alakítja azt (**ParseFeedString**), a harmadik pedig megjeleníti az objektumokat az alkalmazás felületén (**AddItems**). Az **mgr** nevű tag egy **FeedManager** típusú objektum, ez végzi a feladatok oroszlánrészét.

```
private void DoWork(string url)
{
    string host = new Uri(url).Host;
    tbkStatus.Text = host + " letöltése...";
    var str = mgr.GetFeedString(url);
    tbkStatus.Text = host + " feldolgozása...";
    var feeditems = mgr.ParseFeedString(str);
    tbkStatus.Text = host + " listázása...";
    AddItems(feeditems);
    tbkStatus.Text = host + " kész.";
}
```

Ha ezt a kódot a UI szálon futtatjuk, meglehetősen nagy az esélye, hogy a felület több másodpercig nem reagál majd, és még a **tbkStatus** vezérlő **Text** tulajdonságának változásait sem látjuk (csak a legutolsót). Ez azért van, mert a UI szál először is megvárja, míg választ kap a szervertől a **GetFeedString** kérésére, majd megvárja, míg a processzor értelmezi, átalakítja a választ objektumokká, illetve megjeleníti a felületen.

Amennyiben a fenti három metódusból az első kettőhöz rendelkezésre áll **Begin**/**End** metóduspár, a szinkron kód átalakítható aszinkron futásúvá, ezzel levéve a terhet, illetve a felesleges várakozást a UI-t futtató szálról. Helyette egy (vagy a pontos működéstől függően akár több) háttérszál végzi a munkát, a UI-szál feladata kimerül a felhasználói interakció kezelésében, amelyre eleve rendeltetett:

```
        string host;

        private void BeginDoWork(string url)
        {
            host = new Uri(url).Host;
            tbkStatus.Text = host + " letöltése...";
            mgr.BeginGetFeedString(url, BeginGetFeedStringCompleted, null);
        }

        private void BeginGetFeedStringCompleted(IAsyncResult ar)
        {
            var str = mgr.EndGetFeedString(ar);
            tbkStatus.Text = host + " feldolgozása...";
            mgr.BeginParseFeedString(str, BeginParseFeedStringCompleted, null);
        }

        private void BeginParseFeedStringCompleted(IAsyncResult ar)
        {
            var feeditems = mgr.EndParseFeedString(ar);
            tbkStatus.Text = host + " listázása...";
            AddItems(feeditems);
            tbkStatus.Text = host + " kész.";
        }
```

Hosszabb lett a kód, viszont innentől fogva ha elindítjuk a **BeginDoWork** metódust, a végrehajtás az utolsó utasításnál (**BeginGetFeedString**) átadódik egy háttérszálnak, és a **BeginDoWork** visszatér – vagyis a feldolgozás már a háttérben folyik, nem blokkolja a UI működését.

A **host** nevű lokális változót vagy az osztály adattagjává kell tenni, vagy pedig a **Begin** metódusok utolsó paramétereként (ami itt most **null** volt) kell átadni. Utóbbi esetben minden **End** metódus elején egy plusz utasítás lenne, amelyben létrehozzuk a lokális változót, és átadjuk neki az **IAsyncResult** paraméter **AsyncState** tulajdonságának értékét. A fenti megoldás tehát valamelyest egyszerűsíti a kódot, ugyanakkor elsőre talán nem látható problémákat is felvet.

Ha egyszerre többször is elindítjuk a **BeginDoWork** metódus végrehajtását, a több szálon futó metódusok gyors egymásutánban módosíthatják a **host** értékét, és ennek következtében a callback metódusok (a **BeginGetFeedStringCompleted** és a **BeginParseFeedStringCompleted**) esetleg nem az eredetileg hozzájuk rendelt értéket látják, hanem egy másik **BeginDoWork** által beállítottat.

Viszont ha elindítjuk a programot a fenti metódusokat használva, azzal szembesülhetünk, hogy a UI-hoz hozzáférő utasítások (például a **tbkStatus.Text** frissítése) kivételt dobnak a visszahívott metódusokban! Ennek az az oka, hogy a visszahívott metódusok azon a szálon futnak, melyen maga a feldolgozás, vagyis egy háttérszálon. A vezérlőkhöz pedig csak a UI-szálon lehet hozzáférni, tehát kivételt fogunk kapni. Ezt a problémát úgy tudjuk megoldani, ha azoknál az utasításoknál, melyeknek a UI-hoz kell hozzáférniük, utasításba adjuk a keretrendszernek, hogy az utasítást vagy utasításokat küldjék át a UI-szálra. A kód, amely megoldja ezt, WPF esetében az alábbi lesz (a **BeginDoWork** metódus nem változott, csak a teljesség kedvéért szerepel újra):

```
private void BeginDoWork(string url)
{
    host = new Uri(url).Host;
    tbkStatus.Text = host + " letöltése...";
    mgr.BeginGetFeedString(url, BeginGetFeedStringCompleted, null);
}

private void BeginGetFeedStringCompleted(IAsyncResult ar)
{
    var str = mgr.EndGetFeedString(ar);
    Dispatcher.Invoke(() => tbkStatus.Text = host + " feldolgozása...");
    mgr.BeginParseFeedString(str, BeginParseFeedStringCompleted, null);
}
```

```
private void BeginParseFeedStringCompleted(IAsyncResult ar)
{
    var feeditems = mgr.EndParseFeedString(ar);
    Dispatcher.Invoke(() =>
        {
            tbkStatus.Text = host + " listázása...";
            AddItems(feeditems);
            tbkStatus.Text = host + " kész.";
        });
}
```

Itt gyakorlatilag az történik, hogy a kódot tovább bontjuk: azok a részek, melyeknek a UI-hoz kell hozzáférniük, de *feltehetően* háttérszálon fognak futni, visszakerülnek a UI-ra.

Innentől a kód már késznek tekinthető; vagy legalábbis működőképesnek. Nagyjából így néz ki tehát az eredeti, körülbelül tízsoros kód APM segítségével létrehozott aszinkron változata.

Jól látható, hogy a szükséges kód mérete tovább nőtt – a nagyobb gond, hogy átláthatósága, olvashatósága, valamint a karbantarthatósága jelentősen csökkent. Az egyetlen, átlátható logikájú metódust felbontottuk három részre, ráadásul felmerültek többszálúságból adódó problémák is. Az átalakítás egyetlen pozitív vonzata, hogy a UI a feldolgozás alatt nem blokkolódik.

A fenti hátrányokat sajnos nem lehet kikerülni. Egy metódusnak mindenképpen le kell futnia, mielőtt visszaadhatná a vezérlést az őt hívónak, tehát vagy végrehajt minden feladatot, és ebben az esetben a hívó metódus várakozni kényszerül, vagy azonnal visszatér, de a feladatot egy háttérszál hajtja végre, függetlenül a hívótól. Ebben az esetben pedig a koordinációról nekünk kell gondoskodni.

Nyelvi szintű aszinkronitás a C# 5.0-ban

A C# 5.0 legnagyobb újdonsága, hogy a fordítóprogram képes levenni a fejlesztő válláról a terhet. Egy nyelvi újítás segítségével úgy írhatunk aszinkron módon futó kódot, hogy közben a szinkron kódhasználat szinte összes előnyét is megkapjuk.

Ha egy metódus tartalmazza a logikát, akkor továbbra is egyetlen, átlátható metódusunk lesz, nem bomlik fel a szekvencia – legalábbis látszólag. Ennek következtében a koordinációról sem nekünk kell gondoskodni, az automatikusan megtörténik a háttérben. Továbbá, ha a UI-ról indítottuk a végrehajtást, már amiatt sem kell extra kódot írnunk, hogy a UI-interakció problémamentes legyen. A háttérben, rejtetten futó visszahívott metódusok automatikusan az eredeti szálra térnek vissza.

Emellett nincs szükség több metódusra vagy eseményre. Egyetlen metódusban tudunk megoldani mindent, ahogy azt a szinkron esetben is láthattuk!

Mindehhez két új kulcsszó és néhány szabály ismeretére van csak szükségünk.

Aszinkron metódusok készítése az async módosítóval

Az új **`async`** módosítót egy metódusra alkalmazva jelezhetjük a fordítónak, hogy ez a metódus aszinkron metódushívásokat tartalmazhat. A fordítóprogram ebből tudni fogja, hogy ha a metóduson belül találkozik az **`await`** operátorral (erről kicsit később), akkor az utána következő metódushívás működését módosítania kell.

Az **`async`** módosítóval jelölt metódusokra a következő szabályok, illetve ajánlások érvényesek:

- A metódus nevét az **`Async`** szóval zárjuk, kivéve, ha már volt ilyen nevű metódus, pl. egy EAP-ból származó; ebben az esetben a **`TaskAsync`** szóval zárjuk. Tehát ha volt egy normál metódus **`DoWork`** névvel, akkor **`DoWorkAsync`** vagy **`DoWorkTaskAsync`** lesz az új neve.
- A metódus paraméterlistájának nem kell változnia.
- A metódus visszatérési típusa csak **`void`**, **`Task`** vagy **`Task<T>`** lehet.
  - Amennyiben olyan metódusról van szó, melyet csak el akarunk indítani, de nem várunk tőle eredményt (jó példa erre gyakorlatilag az összes UI-eseménykezelő), és nem vagyunk kíváncsiak arra, hogy mikor fejeződött be, akkor **`void`** lehet a visszatérési típusa.

- Ha nem várunk eredményt a metódustól, de a feldolgozás során szükségünk van arra, hogy a befejezéséhez szinkronizáljunk más részfeladatokat, akkor annak **`Task`** típusú objektumot kell visszaadnia.
- Ha konkrét adatot várunk a metódustól (mert például az eredeti, szinkron metódus is visszaadott valamit), akkor **`Task<T>`** legyen a visszatérési típusa, ahol **`T`** a várt eredmény típusa!

Itt egy példa egy szinkron metódusra, és annak C# 5.0-ás aszinkron párjára:

```
string DoSomething(int i)
{
    //…
}

async Task<string> DoSomethingAsync(int i)
{
    //…
}
```

A **`Task`** és **`Task<T>`** osztályok működése a későbbiekben bővebben terítékre kerül. Most egyelőre elég, ha annyit tudunk róluk, hogy létrehozásukkor minden **`Task`** egy metódust vár, és példányaikat a keretrendszer egy saját maga által választott háttérszálon hajtja végre. Vagyis azok segítségével más szálakra oszthatjuk a program egyes részeinek végrehajtását, ezáltal tehermentesítve a Task objektumot eredetileg létrehozó szálat. Amennyiben egy Task objektumon belül hozunk létre egy másik **`Task`**ot, akkor természetesen előfordulhat, hogy ugyanaz a szál futtatja a szülőt és a gyermeket is.

Érdemes kiemelni, hogy névtelen metódusok és lambda kifejezések is megjelölhetők az **`async`** módosítóval! Ennek értelmében a nyelvi szintű aszinkronitás nem korlátozódik a „hagyományos” metódusokra.

Az async módosítóval megjelölt metódusok hívásának módosítása az await operátorral

Az **`async`** módosító önmagában még semmit nem módosít a metóduson. Ha meghívjuk, ugyanúgy, szinkron módon fog futni, mint bármely normál metódus. Pusztán azt jeleztük vele a fordítónak, hogy lehet, hogy használni fogjuk benne az **`await`** operátort (ha megtesszük, akkor módosul ténylegesen a metódus szerkezete a fordításkor), illetve hogy máshol, ennek a metódusnak a hívásakor is alkalmazható az **`await`**.

Ha egy **`async`** metódust az **`await`** kulcsszó használata nélkül hívunk meg, egy teljesen átlagos metódushívást kapunk. Az alábbi példában a **`GetFeedString`** metódust alakítjuk át úgy, hogy string helyett egy **`Task<string>`** példányt adjon vissza:

```
public string GetFeedString(string url)
{
    WebClient wc = new WebClient();
    string feedResp = wc.DownloadString(url);
    return feedResp;
}

public Task<string> GetFeedStringAsync(string url)
{
    WebClient wc = new WebClient();
    Task<string> feedResp = wc.DownloadStringTaskAsync(new Uri(url, UriKind.Absolute));
    return feedResp;
}
```

Látható, hogy csak a visszatérési típus és a hívott metódus változott. A **`DownloadStringTaskAsync`** metódustól egy **`Task`** példányt kapunk vissza, ez jelzi, hogy a végrehajtás a háttérben, egy másik szálon történik. A **`DownloadStringTaskAsync`** azonnal visszatér, ahogy a Task létrejött, így a **`GetFeedStringAsync`** metódus is. Azonban egy **`Task`** bevárása szinkron módon nem túl elegáns, elveszítjük vele azt az előnyt, amit annak használata ad, vagyis a párhuzamos, aszinkron végrehajtást. Éppen ezért lehetőségünk van arra,

hogy egy futó **Task**hoz hozzárendeljünk egy folytatást: egy másik **Task**ot, mely automatikusan elindul, amint az első Task eredménye előáll. Az alábbi kód szemlélteti, hogyan néz ki, ha a **Task**okra, illetve azokat visszaadó metódusokra építjük az aszinkronitást:

```
private void DoWork(string url)
{
    string host = new Uri(url).Host;
    tbkStatus.Text = host + " letöltése...";
    mgr.GetFeedStringAsync(url).ContinueWith(taskStr =>
    {
        Dispatcher.Invoke(() => tbkStatus.Text = host + " feldolgozása...");
        var feeditems = mgr.ParseFeedStringAsync(taskStr.Result);
        feeditems.ContinueWith(taskItems =>
        {
            Dispatcher.Invoke(() =>
                {
                    tbkStatus.Text = host + " listázása...";
                    AddItems(taskItems.Result);
                    tbkStatus.Text = host + " kész.";
                });
        });
    });
}
```

A fenti kódban látható, hogy ezúttal mindenáron egyetlen metódusban akartuk leírni a logikát, és ennek érdekében névtelen függvényeket (lambda kifejezéseket) használtunk. De hiába ez az erőfeszítés, mert így oda jutottunk, hogy egy metóduson belül van három másik, és ez ismét az olvashatóság, tesztelhetőség és karbantarthatóság rovására megy.

Változtassuk meg a **GetFeedStringAsync** metódust úgy, hogy megjelöljük azt az **async** módosítóval! Azt fogjuk tapasztalni, hogy a kód nem fordul le. A Visual Studio meglehetősen viccesen azt jelzi, hogy a **Task<string>**-et visszaadó metódusnak nem szabad **Task<string>**-et visszaadnia! Mindaddig nem fordul a kód, míg át nem írjuk az utolsó utasítást, hogy a **Task** helyett a **Task** eredményét adja vissza – ami mellesleg egy **string**. (Figyelem! Az await kulcsszót még nem használtuk.)

```
public async Task<string> GetFeedStringAsync(string url)
{
    WebClient wc = new WebClient();
    Task<string> feedResp = wc.DownloadStringTaskAsync(new Uri(url, UriKind.Absolute));
    return feedResp.Result;
}
```

A kód fordul, a program továbbra is futtatható, annak ellenére, hogy a fenti metódus látszólag nem azt a típust adja vissza, amit a szignatúrája megkövetelne. Fontos azonban kiemelni, hogy hiába a **Task** használata, az **await** nélkül a kód szinkron módon fog futni, vagyis az utolsó utasítás bevárja, amíg a **Task** visszaadja az eredményét.

Utolsó lépésként tehát írjuk ki az **await** operátort a **DownloadStringTaskAsync** metódus elé! Megint érdekes hibával szembesülhetünk: a metódus ezek után azt állítja, hogy ő **string**et ad vissza, nem pedig **Task<string>-**et! Ennek megfelelően módosítanunk kell az értékadás bal oldalát, valamint az utolsó utasításban is magát a **feedResp** objektumot kell visszaadnunk, nem pedig annak **Result** tulajdonságát:

```
public async Task<string> GetFeedStringAsync(string url)
{
    WebClient wc = new WebClient();
    string feedResp = await wc.DownloadStringTaskAsync(new Uri(url, UriKind.Absolute));
    return feedResp;
}
```

Az **await**tel jelezzük, hogy bár látszólag szinkron módon hívjuk és használjuk a metódust, szeretnénk, ha az a háttérben automatikusan aszinkron módon futna. Az **await** után következő metódushívás tehát elindul, de a metódus ez után következő része már folytatásként fog lefutni: az külön **Task** lesz, amelyet a fordító elől elrejtve hoz létre.

Mivel ez egy **async** metódus (csak az **async**-kel megjelölt metódusokban használhatjuk az **await**et), ezért ő maga is „**await**elhető". Feltételezve, hogy a **ParseFeedString** metódust is átírtuk a fentieknek megfelelően, a **DoWork** metódus az alábbiak szerint írható át:

```
//az eredeti, szinkron metódus
private void DoWork(string url)
{
    string host = new Uri(url).Host;
    tbkStatus.Text = host + " letöltése...";
    var str = mgr.GetFeedString(url);
    tbkStatus.Text = host + " feldolgozása...";
    var feeditems = mgr.ParseFeedString(str);
    tbkStatus.Text = host + " listázása...";
    AddItems(feeditems);
    tbkStatus.Text = host + " kész.";
}

//az előbbi aszinkron változata
private async void DoWorkAsync(string url)
{
    string host = new Uri(url).Host;
    tbkStatus.Text = host + " letöltése...";
    var str = await mgr.GetFeedStringAsync(url);
    tbkStatus.Text = host + " feldolgozása...";
    var feeditems = await mgr.ParseFeedStringAsync(str);
    tbkStatus.Text = host + " listázása...";
    AddItems(feeditems);
    tbkStatus.Text = host + " kész.";
}
```

Úgy érezzük, hogy a kód szinte semmit sem változott, első ránézésre akár teljesen szinkronnak is tűnhet; a logika átlátható maradt.

A háttérben viszont a metódus egyetlen utasítása sem fog megegyezni azzal, amit a fejlesztő írt. Ehelyett a C# fordító egy beágyazott állapotgép-típust generál, amely a **DoWorkAsync** metódusban leírt folyamat lépésenkénti (tulajdonképpen „**await**enkénti") végrehajtásáért felelős. A **DoWorkAsync** metódus mindössze annyit tesz, hogy példányosítja ezt az állapotgépet, és belépteti az első állapotba. Ezután már az állapotgép feladata lesz az eredeti utasítások végrehajtása és az eredetileg lokális változók megfelelő kezelése. (Ezek a változók az állapotgép példányának adattagjai lesznek.)

Mindez a fejlesztő elől teljesen elrejtve történik, az egész kódújraírásért a C# fordítóprogram a felelős. Az egész folyamat valamelyest hasonlít a C# 2.0 óta jelenlévő képességére, mellyel automatikusan hozathatunk létre iterátor típusokat.

A következő feladatban – ez prímszámok keresésével foglalkozik – kipróbálhatod, hogy milyen könnyű az aszinkron képességek használata. Ebben egy Windows 8 stílusú alkalmazást kell fejlesztened, de mivel azok a fejezetek, melyek konkrétan ezen alkalmazások fejlesztését írják le, hátravannak, itt leginkább csak a Windows 8-tól független részekre fogsz koncentrálni.

> *Ebben a gyakorlatban az egyszerűség kedvéért néhány, amúgy minden alkalmazásban ajánlott infrastrukturális kódot is elhagyunk, így például nem tiltjuk le vagy engedélyezzük a gombokat, illetve csak a példa szempontjából releváns esetben használunk kivételkezelést.*

***Gyakorlat**: Aszinkron metódusok használata*

A C# 5.0 új aszinkron programozási modelljének kipróbálásához kövesd az alábbi lépéseket:

1. Nyisd meg a fejezethez tartozó kódok közül az **`AsyncBegin`** nevű könyvtárban található projektet! Vizsgáld át a **`MainPage.xaml`** fájlban leírt felületet a későbbi módosítások gyorsabb elvégzése végett!
2. Módosítsd a **`MainPage.xaml`** fájlban a **`Page`** elemet: adj hozzá egy eseménykezelőt a **`Loaded`** eseményéhez! A **`Page`** elemnek most így kell kinéznie (az egyes attribútumok megadásának sorrendje lényegtelen):

```
<Page
    x:Class="TPLDemo.MainPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:TPLDemo"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d" Loaded="Page_Loaded">
```

3. Módosítsd a XAML fájlban az „Indítás” és az alatta található „Aszinkron indítás” feliratú gombot: írj fel egy-egy metódust a **`Click`** nevű eseményeikre! A **`Button`** elemeknek most így kell kinézniük:

```
<Button Click="btnStart_Click" Content="Indítás" />
<Button Click="btnAsyncStart_Click" Content="Aszinkron indítás" />
```

4. Nyisd meg a **`MainPage.xaml.cs`** fájlt, amelyben az előbbi oldal mögöttes kódja található! Illeszd be az alábbi kódot a konstruktor után:

```
private void Page_Loaded(object sender, RoutedEventArgs e)
{
    tbFirstNum.Text = (1024).ToString();
    tbLastNum.Text = (1024 * 1024).ToString();
    tbTimeout.Text = 5000.ToString();
}
```

5. A fájlban több **`#region`** és **`#endregion`** címke közé zárt üres rész is található. Illeszd be az alábbi kódot az „Első fázis” régióba:

```
private bool IsPrime(int number)
{
    for (int i = 2; i <= Math.Sqrt(number); i++)
        if (number % i == 0) return false;
    return true;
}

private void FindPrimes(int min, int max)
{
    for (int i = min; i < max; i++)
    {
        if (IsPrime(i)) gvResult.Items.Add(i);
    }
}

private void btnStart_Click(object sender, RoutedEventArgs e)
{
    tbkResult.Text = "Elindítva";
    gvResult.Items.Clear();
```

```
    int min = int.Parse(tbFirstNum.Text);
    int max = int.Parse(tbLastNum.Text);

    FindPrimes(min, max);
    tbkResult.Text = "Kész.";
}
```

6. Illeszd be az alábbi kódot a „Második fázis” régióba:

```
private async Task FindPrimesAsync(int min, int max)
{
    await Task.Factory.StartNew(async () =>
    {
        for (int i = min; i < max; i++)
        {
            if (IsPrime(i))
                await Dispatcher.RunIdleAsync(_ => gvResult.Items.Add(i));
        }
    });
}

private async void btnAsyncStart_Click(object sender, RoutedEventArgs e)
{
    tbkResult.Text = "Elindítva";
    gvResult.Items.Clear();
    int min = int.Parse(tbFirstNum.Text);
    int max = int.Parse(tbLastNum.Text);

    await FindPrimesAsync(min, max);
    tbkResult.Text = "Kész.";
}
```

7. Fordítsd le és futtasd az alkalmazást az F5 billentyű lenyomásával! Ha a fordító hibát jelez, próbáld meg kijavítani az Errors ablakban található javaslatok segítségével! Ha nem sikerül, nyisd meg az ehhez a fejezethez tartozó kódok közül a **`BeginTPL`** projektet, és az abban lévő kóddal összehasonlítva keresd meg a megoldást – vagy egyszerűen írd felül a hibás kódot. Ha már fut a program, kattints az „Indítás” gombra, majd ha a futás véget ért, az „Aszinkron indítás” gombra! Míg az első gombra kattintás után a felület érezhetően megfagy, nem reagál a felhasználói interakcióra, és az eredmények csak a teljes művelet lefutása után jelennek meg, a második, aszinkron esetben a felület válaszképes marad, és az eredmények folyamatosan jelennek meg a felületen.
8. Mentsd el az alkalmazást – ezt fogod továbbfejleszteni a következő feladatban!

Hogyan működik?

A prímeket egy legkisebb és egy legnagyobb szám között keressük, ezért először is az ablak megjelenésekor feltöltjük a **`tbFirstNum`** és **`tbLastNum`** szövegdobozokat két „kerek” számmal, 1024-gyel és 1024*1024-gyel (2., 4. lépések). Ezek futás közben persze átírhatók. A harmadik értékre és szövegdobozra csak később lesz szükség.

A 3. lépésben feliratkozunk két gomb **`Click`** eseményére. Az „Indítás” gomb megnyomásakor az 5. lépésben hozzáadott metódusok jutnak szerephez. A **`btnStart_Click`** beolvassa a két számot a szövegdobozokból, majd ezeket átadva elindítja a **`FindPrimes`** metódust, mely az **`IsPrime`** metódussal megállapítja a minimum és maximum közötti számokról, hogy azok prímek-e, és ha igen, hozzáadja azokat a **`gvResult`** nevű **`GridView`** vezérlőhöz. Látható (és futtatáskor érezhető is), hogy itt a számítás a UI-szálon történik.

A 6. lépésben hozzáadott metódusok már aszinkron végrehajtásúak, ahogy a szignatúrájukban lévő **`async`** módosító, illetve nevük is mutatja.

A **btnAsyncStart_Click** metódus tulajdonképpen csupán annyiban különbözik a **btnStart_Click**től, hogy utolsó utasításaként nem a **FindPrimes**t hívjuk, hanem annak aszinkron változatát, és ezt megjelöljük az **await** módosítóval. Ezzel jelezzük, hogy szeretnénk, hogy a metódus másik szálon fusson.

A **FindPrimes** metóduson az **async** módosító hozzáadásán kívül annyit változtattunk, hogy **void** helyett már **Task** a visszatérési típusa, ennek segítségével lehet bevárni. A **min** és **max** értékek közötti ciklust pedig becsomagoltuk egy **Task**ba, amelyet az **await** módosítóval rögtön el is indítunk. A **Task.Factory StartNew** metódusáról a későbbiekben még lesz szó. Mivel ez a metódus már nem a UI-szálon fut, ezért a UI módosítását (vagyis a prímek hozzáadását a **GridView**-hoz) csak úgy tudjuk elvégezni, hogy a **Dispatcher** nevű osztály segítségével átadjuk a futtatásra váró utasítást vagy utasításokat a UI-szálnak; erre szolgál tehát a **Dispatcher.RunIdleAsync()** hívás. Ez egy aszinkron metódus, ezért hívását az **await** módosítóval megjelölhetjük, ilyenkor viszont a tartalmazó metódusnak is aszinkronnak kell lennie – ezért jelöltük meg a **StartNew** metódusnak átadott lambda kifejezést is az **async** módosítóval:

```
await Task.Factory.StartNew(async () => { ... });
```

Az, hogy egy folyamatnak melyik részét tesszük aszinkronná, tőlünk függ! Dönthettünk volna úgy is, hogy nem a **FindPrimes** metódusból készítünk aszinkron változatot, hanem az **IsPrime**-ból, például az alábbi módon.

```
private async Task<bool> IsPrimeAsync(int number)
{
    return await Task.Factory.StartNew<bool>(() =>
    {
        for (int i = 2; i < Math.Sqrt(number); i++)
            if (number % i == 0) return false;
        return true;
    });
}
```

Ahogy látható, itt a **bool** helyett **Task<bool>**-t adunk vissza, és a **Task.Factory StartNew** metódusának generikus változatát használjuk fel az aszinkron indításra, jelezve, hogy az indítandó művelet egy **bool** értéket ad majd vissza. Ennek a megoldásnak a hátránya az lenne, hogy így minden egyes számhoz külön **Task**ot indítanánk, és ez jelentősen megnövelné az adminisztrációs költségeket. Viszont lenne előnye is – erre a fejezet későbbi részében, a párhuzamosítás megvalósításakor térünk rá.

Tudnivalók az aszinkron metódusokról

Visszahívás futtatása azonos szálon

Az aszinkron metódus „visszahívása” (callbackje) mindig azon a szálon fut le, amelyről meghívták az aszinkron metódust. Azért így oldották ezt meg a tervezők, mert legtöbbször az aszinkron módon futtatott metódus eredményét azon a szálon akarták feldolgozni, amelyről elindították az aszinkron metódust. Elég csak a korábbi példára gondolni, amelyben a feldolgozás eredményét, illetve az állapotjelzést a köztes lépésekről a felhasználói felületen kellett megjeleníteni.

Előfordulnak azonban bizonyos helyzetek, amikor ez a viselkedés – amely egyébként eltér a korábban megszokottól – nem kívánt, és a fejlesztő azt szeretné elérni, hogy a visszahívott metódus, vagyis az aszinkron metódus ugyanazon a szálon fusson továbbra is. Ilyen esetekben a fejlesztőnek kell gondoskodnia arról, hogy a futtatókörnyezet a visszahívást az azonos szálon futtassa.

Mivel ez viszonylag gyakori probléma, így a C# fordító és a futtatókörnyezetek fejlesztői egyszerűvé tették a megoldást. Amennyiben egy aszinkron metódushívásnál szeretnénk, ha a visszahívás azonos szálon történne, a metódus meghívása után a visszaadott objektumon a **ConfigureAwait** metódust kell meghívnunk, paraméterként átadva egy **false** értéket, és ezt a metódushívást megjelölni az **await** módosítóval.

A gyakorlatban ez még egyszerűbb, mint amilyennek hangzik. Az alábbi példakód egy metódus aszinkron hívását mutatja be, a hívó szálra visszatérő folytatással, majd ugyanennek a metódusnak az aszinkron hívását, azonos szálon történő továbbfuttatással.

```
await FindPrimesAsync(min, max);
tbkResult.Text = "Kész.";

await FindPrimesAsync(min, max).ConfigureAwait(false);
tbkResult.Text = "Kész.";
```

Amennyiben a fenti módosítást végrehajtjuk a korábbi feladatban szereplő kódon, a **`tbkResult.Text`** beállítása kivételt fog dobni, mivel a **`FindPrimesAsync`** folytatása már nem a UI-szálra tér vissza.

Mit érdemes aszinkronná tenni?

Egy másik fontos tudnivaló az aszinkronitás használata kapcsán, hogy bár tény, hogy tehermentesíthetjük a hívó szálat, de a háttérben lévő állapotgép használata ugyanakkor némi lassulást is eredményez. Az állapotgépet létre kell hozni, ez memóriafoglalással jár, az inicializálás további időt vesz el, illetve számítani kell rá, hogy esetleg beindul az automatikus szemétgyűjtés is, ami megintcsak lassítja az üzleti logikát. Általában elhanyagolhatóak ezek a hátrányok, azonban érdemes az aszinkronná tétel előtt meggondolni, hogy az adott feladat biztosan elég hosszú lefutású-e ahhoz, hogy megérje aszinkronná tenni!

E probléma kapcsán pedig felmerül a kérdés, hogy mennyire érdemes részekre bontani egy folyamatot, illetve érdemes-e az összes részfolyamatot, lépést aszinkron módon hívhatóvá tenni. Ha a részfeladatok önmagukban is jelentős ideig futhatnak, akkor érdemes lehet az egyes lépéseket aszinkronná tenni. Más esetekben viszont nem feltétlenül jár előnnyel az aszinkronitás következetes (túl)használata.

Nincs ökölszabály arra, honnantól érdemes aszinkronná tenni egy metódust, és hol van az a pont, amikor még nem. Jó megközelítési mód lehet az alábbi: nézzük át a metódusokat, és amelyek előre meg nem sejthető hosszúságú I/O-műveletet tartalmaznak (pl. egy webszolgáltatás hívása, vagy egy fájl beolvasása a háttértárról), illetve feltehetőleg hosszú lefutású számítást tartalmaznak, azokat tegyük aszinkronná. Egy folyamat lépésekre bontásánál pedig a kívülről, a felhasználók által is látható, hosszabb lefutású lépéseknél érdemes az **`async`** módosítót a metódusra tenni.

Az aszinkronitás nem párhuzamosítás...

...legalábbis a két technika céljainak tekintetében.

Technikailag természetesen itt is párhuzamosítunk: a metódus utasításait nem a hívó szálnak kell végrehajtania, hanem egy azzal párhuzamosan futó másiknak. Az aszinkronitás célja azonban pusztán annyi, hogy a hívó szálat tehermentesítse, míg a kanonikusan értett párhuzamosítását a feladat részében vagy egészében rejlő párhuzamosítás leképezése az architektúra által adott lehetőségekre. Vagyis amíg az aszinkronitás feladata, hogy egy szálat tehermentesítsen, addig a párhuzamosítását az, hogy lehetőség szerint minden szálat bevonjon egy feladat megoldásába.

Az alábbi nagyon egyszerű kódrészlet arra világít rá, hogy a korábbiakban ismertetett módszerrel aszinkronná tett kód még nem feltétlenül a lehető legjobb megoldás a problémára:

```
// Eredeti állapot
for (int i = 0; i < list.Count; i++)
{
    GetData(list[i]);
}

// Módosított állapot
for (int i = 0; i < list.Count; i++)
{
    await GetDataAsync(list[i]);
}
```

A **GetDataAsync** metódus hívását először szinkron, majd aszinkron módon kezeljük. Ennek következtében ha a **GetDataAsync** metódusok futása hosszabb időt vesz igénybe, azt már nem a hívó szálnak kell kivárnia. Ha például a UI-ról hívtuk a metódust, sikerült válaszképessé tenni a felületet. Ha korábban például 10 másodpercre lefagyott a UI, most válaszképes lesz, és 10 másodpercig folyamatosan jelennek meg az eredmények.

Viszont az is észrevehető, hogy a lista egyes elemeinek feldolgozása nem függ egymástól. Vagyis megtehetnénk, hogy „egyszerre” hívjuk az összes **GetDataAsync** metódust, hiszen nem lehet probléma abból, hogy a lista második elemének feldolgozásakor még nem áll rendelkezésre az első elem feldolgozásából származó eredmény. Tehát elviekben akár arra is lehetőségünk lenne, hogy az előbb említett 10 mp-es időt lecsökkentsük.

Ezt a problémát aszinkronitással nem tudjuk megoldani. Ennek a megoldására szolgál a többszálúságra épülő párhuzamosítás, amelyről a fejezet fennmaradó része szól.

Bevezetés a Task Parallel Library-be

Miért van szükség a párhuzamosításra?

Az elmúlt évtizedben megváltozott a processzorok fejlődésének iránya.

Korábban évente-kétévente megduplázódott a nyers erő, vagyis a CPU-k másodpercenkénti órajelciklusának mennyisége. Ez a fejlesztő számára azt jelentette, hogy a korábbi algoritmusa kétszer olyan gyorsan futott, anélkül, hogy bármit is tenni kellett volna érte. De ahogy a processzorgyártók elértek egy gyakorlati plafont ezen a téren, a MHz-ek növelése eltűnt – gyakorlatilag ma is 4 GHz alatti processzorokkal kell dolgoznunk, ugyanúgy, ahogy 3-4 x86/x64-es processzor-generációval ezelőtt. Ugyan az architektúra folyamatosan gyorsul, ez azonban korántsem okoz olyan mértékű sebességnövekedést, mint korábban egy ugrás 2 GHz-ről 3 GHz-re.

A 2000-es évek közepén a processzorgyártók új módokat kerestek arra, hogy a fejlődést fenntartsák akkor is, amikor a nyers sebességet nem növelhették tovább jelentősen. Mivel a logikai elemek miniatürizálása tovább folytatódott, megnyílt a lehetőség arra, hogy a lapkán, mely korábban egy magot tartalmazott, többet helyezhessenek el. Napjainkban nem ritka, hogy négy vagy több magot tartalmazó gépeken dolgozhatunk, de előfordulnak több mint 30 maggal rendelkezők is. Az operációs rendszer természetesen képes arra, hogy ezeket ki is használja: a futó alkalmazásokat egy-egy maghoz hozzá tudja rendelni, így ténylegesen több alkalmazás futhat egyszerre. (Végre lehet DivX-kódolás közben is akadásoktól mentesen Crysis-elni.)

Más viszont a helyzet az egyes alkalmazásokon belüli párhuzamosítással! A korábban **T** idő alatt lefutó algoritmus, amely **n** elemet dolgozott fel, továbbra is **T**-közeli idő alatt fut le, ha az elemeket sorban dolgozzuk fel. Vegyük a korábbi példát:

```
for (int i = 0; i < list.Count; i++)
{
    GetData(list[i]);
}
```

Sem a keretrendszer (.NET, vagy Windows Runtime), sem pedig az operációs rendszer nem fog több magon végrehajtani egy ciklust. A **GetData** metódusok sorban egymás után hajtódnak végre; mindegyik megvárja, amíg az előtte lévő lefutott. Vagyis ha a fejlesztő nem teszi manuálisan párhuzamossá az algoritmust, programja nem fogja tudni kihasználni az újabb processzorok nyújtotta számítási kapacitástöbbletet. Ha a fenti ciklust egy négymagos processzoron futtatjuk, lehet, hogy az egyik magot csúcsra járatja, de közben a másik három „alszik”. A potenciális teljesítmény háromnegyede elveszett az alkalmazás számára. Mivel pedig az alkalmazások általában egyre bonyolultabbak lesznek, egyre sürgetőbb, hogy ez a probléma megoldást nyerjen.

A processzor minden további nélkül képes arra, hogy az egyazon alkalmazáshoz tartozó utasításfolyamokat több magon dolgozza fel. A probléma tehát nem hardveres, hanem magasabb absztrakciós szinten keresendő!

Aki már írt olyan többszálú programot .NET-ben, amely jelentősebben leterhelte a CPU-t, az tapasztalhatta, hogy bizonyos esetekben a keretrendszer, illetve az operációs rendszer úgy döntött, hogy egy-egy szálat ténylegesen másik magon kezd futtatni. A probléma az, hogy az eredetileg elsősorban egymagos gépekre tervezett .NET keretrendszer hajlott afelé, hogy csak „legvégső esetben" használja ki a többmagos processzorokat. Nem is lehetett beállítani, hogy egy-egy szál – vagyis párhuzamosan futtatott metódus – melyik magon fusson.

A szálak inkább arra szolgáltak, hogy egy alkalmazásban több olyan feladatot lehessen párhuzamosan futtatni, amelyek egyenként nem foglalták le teljesen a processzort. Például egy háttérszálon beolvasva egy fájlt a program többi része válaszképes maradt. De amikor egy nagy processzorigényű számításhalmazt kellett végrehajtani, a közvetlen szálhasználat jelentős mértékben bonyolította a fejlesztést, esetenként akár annyira, hogy a lehetséges haszon nem érte meg a fejlesztésbe beleölt emberórákat.

A párhuzamosíthatóság problémája tehát már a Windows Runtime előtt megjelent. A Microsoft megoldása a problémára a Task Parallel Library (továbbiakban TPL), mely mind a .NET 4, mind a Windows Runtime részét képezi. Utóbbi keretrendszerben nincs is lehetőségünk közvetlenül szálakat létrehozni.

Párhuzamosan futó műveletek indítása

A TPL feladata egyszerűvé tenni a párhuzamosítást. A korábbi – a **`GetData`** metódust ciklusban használó – példánál maradva megtehetjük, hogy minden **`GetData`** metódust egy-egy külön **`Task`**ként hozunk létre és indítunk el, ahogy az alábbi kód is szemlélteti.

```
for (int i = 0; i < list.Count; i++)
{
    Task.Factory.StartNew(() => GetData(list[i]));
}
```

Az elindított **`Task`**ok egymással párhuzamosan fognak futni, mégpedig annyi szálon, amennyi a programot futtató gépen a legoptimálisabb. Ez megegyezik a magok számával. Nincs lehetőségünk arra, hogy kézzel hozzárendeljünk egy-egy **`Task`** példányt egy-egy processzormaghoz, de erre szükség sincs. Az elkészített **`Task`**ok egy globális sorba kerülnek, melyről az egyes szálak megkapják őket. Így a fenti esetben vélhetően nem egyszerre fog a rengeteg **`Task`** elindulni, hanem egyszerre csak annyi fut, ahány processzormag található az adott gépben, mert ez a legoptimálisabb. Ha az egyik szálnak nincs feladata, kap egy futtatásra váró **`Task`**ot. A háttérben lévő ütemező helyettünk gondoskodik arról, hogy a szálaknak folyamatosan legyen munkája, és ne fordulhasson elő, hogy míg az egyik szál teljes erővel dolgozik (és teljesen lefoglalja az egyik processzormagot), addig a többieknek nincs elég munkája, és kárba vész a teljesítmény.

A legegyszerűbben és leggyorsabban a **`Task`** osztály **`Factory`** tulajdonságának **`StartNew`** metódusával indíthatunk el egy párhuzamosan futó műveletet. Ezek a műveletek nem adnak vissza semmilyen visszatérési értéket. A **StartNew** metódus egy **Task** objektumot ad vissza – ezen tudjuk tesztelni, hogy az adott feladat elindult-e már, lefutott-e stb. A metódus első paramétere egy **`Action`** vagy **`Action<object>`** delegált, vagyis visszatérési típus nélküli és paramétermentes, vagy visszatérési típus nélküli és egyetlen **`object`** paramétert fogadó metódus.

Emellett viszont lehetőségünk van a **`new`** operátor segítségével létrehozni **`Task`** példányokat. Ilyenkor a **`Start`** metódussal indíthatjuk el a létrehozott példányt.

> *Valójában egyik megoldás sem jelent garantáltan azonnali indítást. A **`Task`**ok a korábban vázoltak szerint egy globális sorba, illetve onnan majd egy szál lokális sorába kerülnek, és csak a sorra kerülésükkor indul el a végrehajtásuk.*

```
for (int i = 0; i < list.Count; i++)
{
    Task t = new Task(() => GetData(list[i]));
    t.Start();
}
```

A fenti ciklust futtatva azt tapasztalhatjuk, hogy maga a ciklus nagyon gyorsan befejeződik. A feladatok aszinkron módon, párhuzamosan futnak, tehát hacsak nem várjuk be őket, illetve nem vagyunk kíváncsiak végeredményükre, maga a ciklus jóval azelőtt véget ér, hogy az elindított **Task**ok lefuthattak volna!

A fenti példákban olyan Task példányokat indítottunk, melyek nem adtak vissza semmit – pontosabban a bennük lévő metódus **void** volt. Azonban lehetőségünk van arra is, hogy olyan feladatokat futtassunk párhuzamosan, melyek konkrét eredményt is adnak.

A **StartNew** metódus generikus változatával (ez egy konkrét típust vár típusparaméterként) olyan feladatokat indíthatunk, melyeknek visszatérési értéke is van. Ilyenkor **Func<TResult>** vagy **Func<object,TResult>** delegáltaknak megfelelő szignatúrájú metódusokat kell átadnunk a **StartNew** metódusnak, ahol a **TResult** a metódus visszatérési típusa. Ugyanezt megtehetjük a **Task<TResult>** osztállyal is.

Ilyen esetekben természetesen el kell, hogy érjük valahogy a **Task**ban futtatott metódus által visszaadott értéket. Erre a **Task** osztály **Result** tulajdonságán keresztül van lehetőségünk.

A következő példa azt szemlélteti, hogy egy **bool** típusú értéket visszaadó **GetData** metódust hogyan futtathatunk **Task**ként, illetve hogyan férhetünk hozzá az eredményéhez:

```
for (int i = 0; i < list.Count; i++)
{
    Task<bool> t = new Task<bool>(() =>
    {
        bool b = GetData(list[i]);
        return b;
    });
    t.Start();
    lbxResults.Items.Add(t.Result);
}
```

Az eredményhez szinkron módon férünk hozzá, vagyis ha még nem állt elő a **Task** eredménye, a metódus (a ciklus) annak megszületéséig blokkolni fog.

A **Task** rendelkezik egy **Status** tulajdonsággal, mely az adott **Task** példány állapotát mutatja. Ez egy **TaskStatus** nevű felsorolt típusból kaphat értéket. A 4-1 táblázat ennek lehetséges értékeit foglalja össze.

4-1 táblázat: A TaskStatus felsorolt típus értékei

Érték	Jelentés
Created	A **Task** már futásra kész, de még nem ütemezték be.
WaitingForAcivation	A **Task** inicializálásra és ütemezésre vár.
WaitingToRun	A **Task**ot már beütemezték, de még nem fut.
Running	A **Task** éppen fut.
WaitingForChildrenToComplete	A **Task** éppen fut, és egy vagy több gyermek **Task** lefutására vár.
RanToCompletion	A **Task** rendben lefutott.
Canceled	A **Task** elfogadta a megszakítását, vagy el sem indult, mivel indulása előtt megszakították.
Faulted	A **Task** befejeződött, de egy kezeletlen kivétel miatt.

Párhuzamosan futó műveletek bevárása

Előfordul, hogy egy vagy több párhuzamosan futó műveletet szeretnénk bevárni, vagyis csak akkor folytatni a program logikájának végrehajtását, amikor a párhuzamosított szakasz már véget ért.

Egy **Task** lefutását a **Wait** metódussal tudjuk bevárni. A hívó szál (az a metódus, ahol meghívtuk a **Wait** metódust) addig nem fut tovább, míg a **Task** futása be nem fejeződött.

```
Task<bool> t = new Task<bool>(() =>
{
    bool b = GetData(list[i]);
    return b;
});
t.Start();
t.Wait();
//további utasítások
```

Ha nemcsak egy, hanem több feladatot is szeretnénk bevárni, akkor a **Task** osztály statikus **WaitAll** metódusát kell meghívnunk, melynek egy **Task** tömböt kell átadni, vagy egyenként a bevárandó **Task** példányokat. A hívó metódus addig blokkol, amíg az összes bevárandó **Task** be nem fejeződik.

```
List<Task<bool>> tasks = new List<Task<bool>>();

for (int i = 0; i < list.Count; i++)
{
    Task<bool> t = new Task<bool>(() =>
    {
        bool b = GetData(list[i]);
        return b;
    });
    t.Start();
    tasks.Add(t);
}
Task.WaitAll(tasks.ToArray());
//további utasítások
```

Előfordul, hogy sok elindított **Task**ból csak egynek a befejezésére várunk, vagy arra vagyunk kíváncsiak, melyik művelet fejeződött be legelőször. Ilyenkor a **Task** osztály statikus **WaitAny** metódusát kell használnunk. Ennek paramétere szintén egy **Task** tömb (vagy vesszővel elválasztva a **Task**ok), ezeket várja be. A **WaitAny**-t hívó metódus csak addig blokkol, míg legalább egy **Task** be nem fejeződött. Ha az első lefutott, visszaadja a **Task** indexét, vagyis azt, hogy hányadik paraméter volt vagy hányadik elem a tömbben.

```
List<Task<bool>> tasks = new List<Task<bool>>();

for (int i = 0; i < list.Count; i++)
{
    Task<bool> t = new Task<bool>(() =>
    {
        bool b = GetData(list[i]);
        return b;
    });
    t.Start();
    tasks.Add(t);
}
int index = Task.WaitAny(tasks.ToArray());
bool res = tasks[index].Result;
```

A következő gyakorlatban a korábbi aszinkron prímszámkereső alkalmazást párhuzamosítjuk. Először a korábbi megvalósítást módosítjuk úgy, hogy a **FindPrimesAsync** metódus egy listát adjon vissza, és a

metódus lefutása után írjuk csak ki az elemeket a képernyőre, majd erre építjük rá a Task Parallel Library nyújtotta párhuzamosítást, hogy az egyes számok ellenőrzése ne csak egy magon folyhasson.

***Gyakorlat**: Párhuzamos **Task**ok indítása és bevárása*

A prímszámkereső algoritmus párhuzamosításához kövesd az itt leírt lépéseket!

1. Nyisd meg az előző feladat végén elmentett projektet vagy a fejezethez tartozó kód **`BeginTPL`** könyvtárában lévőt!
2. Nyisd meg a **`MainPage.xaml`** fájlt, és adj hozzá egy-egy eseménykezelőt az „Aszinkron indítás listával” és „Párhuzamosított indítás” feliratú gombok **`Click`** eseményeihez! A két **`Button`** leírása most így kell, hogy kinézzen:

```
<Button Click="btnAsyncStartWithList_Click" Content="Aszinkron indítás listával" />
<Button Click="btnAsyncParallelStart_Click" Content="Párhuzamosított indítás" />
```

3. Nyisd meg a **`MainPage.xaml.cs`** fájlt! Helyezd el a „Harmadik fázis” nevű szekcióban a **`btnAsyncStartWithList_Click`** metódust, és az általa hívott **`FindPrimesAsyncWithList`** metódust! A következő kódot kell beillesztened:

```
private async Task<List<int>> FindPrimesWithListAsync(int min, int max)
{
    return await Task<List<int>>.Factory.StartNew(() =>
    {
        List<int> results = new List<int>();
        for (int i = min; i < max; i++)
        {
            if (IsPrime(i)) results.Add(i);
        }
        return results;
    });
}

async void btnAsyncStartWithList_Click(object sender, RoutedEventArgs e)
{
    tbkResult.Text = "Elindítva";
    gvResult.Items.Clear();
    int min = int.Parse(tbFirstNum.Text);
    int max = int.Parse(tbLastNum.Text);

    var result = await FindPrimesWithListAsync(min, max);
    foreach (var i in result.Take(500))
        gvResult.Items.Add(i);
}
```

4. Helyezd el az előbbi metódusoknak a párhuzamosított változatát (**`btnAsyncParallelStart_Click`** és **`FindPrimesAsyncParallel`**) a „Negyedik fázis” szekcióban! A következő kódot kell beillesztened:

```
async Task<List<int>> FindPrimesParallelAsync(int min, int max)
{
    return await Task<List<int>>.Factory.StartNew(() =>
    {
        List<int> results = new List<int>();
        List<Task> tasks = new List<Task>();

        for (int i = min; i < max; i++)
        {
            tasks.Add(Task.Factory.StartNew(() =>
            {
```

```
                if (IsPrime(i)) results.Add(i);
            }));
        }
        Task.WaitAll(tasks.ToArray());

        return results;
    });
}

async void btnAsyncParallelStart_Click(object sender, RoutedEventArgs e)
{
    tbkResult.Text = "Elindítva";
    int min = int.Parse(tbFirstNum.Text);
    int max = int.Parse(tbLastNum.Text);

    var result = await FindPrimesParallelAsync(min, max);

    gvResult.Items.Clear();
    foreach (var i in result.Take(500))
        gvResult.Items.Add(i);
    tbkResult.Text = "Kész.";
}
```

5. Fordítsd le, és futtasd az alkalmazást! Ha a fordító hibát jelez, próbáld meg kijavítani az Errors ablakban található javaslatok segítségével! Ha nem sikerül, nyisd meg az ehhez a fejezethez tartozó kódok közül a **`BeginTPLCancel`** projektet, és az abban lévő kóddal összehasonlítva keresd meg a megoldást – vagy egyszerűen írd felül a hibás kódot! Ha már fut, kattints az „Aszinkron indítás listával” gombra, majd ha a futás véget ért, a „Párhuzamosított indítás” gombra!

A futás mindkét esetben aszinkron, a különbség a sebességben jelenik meg. Többmagos processzoron a második gomb által indított művelet rövidebb idő alatt fut le – amennyiben a járulékos adminisztrációs költségek nem múlják felül a több mag kihasználásából adódó sebességtöbbletet. Érdemes lehet az alkalmazást **`Release`** fordítással, hibakeresés nélkül indítani, hogy a valóshoz közelebbi viselkedést lássunk. Ezenkívül pedig érdemes lehet a két küszöbszámot (alapesetben 1024 és 1024^2) úgy módosítani, hogy nagyobb számokat vizsgáljon a program, ezeknek ugyanis arányosan hosszabb lesz a vizsgálata, így jobban kidomborodhatnak a több mag használatából adódó előnyök.

6. Mentsd el az alkalmazást – ezt fogod továbbfejleszteni a következő feladatban!

Hogyan működik?

A 3. lépésben hozzáadott metódusok aszinkron módon működnek. A **`FindPrimesWithListAsync`** metódus a korábban már használt **`Task.Factory.StartNew`** metódussal végrehajt egy ciklust, amelynek minden iterációjában megállapítja egy számról, hogy prím-e, és ha igen, hozzáadja egy listához. Ez a lista lesz aztán a metódus eredménye. A **`btnAsyncStartWithList_Click`** metódus aszinkron módon elindítja az előbbit, majd befejeződése után a lista első 500 elemét kiírja a **`GridView`** vezérlőbe (**`gvResult`**).

A 4. lépésben az előbbi két metódus párhuzamosított változatait hoztuk létre. A **`btnAsyncParallelStart_Click`**ben tulajdonképpen semmi nem változott: elindítjuk a prímkereső metódust, de most a párhuzamosított változatot. Az igazi változás a **`FindPrimesParallelAsync`** metódusban keresendő. A ciklusban minden egyes prímvizsgálatot egy-egy önálló **`Task`**ként hozunk létre. Ezeket amellett, hogy elindítjuk, hozzáadjuk egy **`tasks`** nevű listához. A ciklus lefutása után bevárjuk, amíg a lista összes eleme befejeződött – csak ekkor fejeződik be ténylegesen a **`FindPrimesParallelAsync`** metódus törzsét magába foglaló Task, ami az egész ciklust tartalmazta.

Párhuzamosan futó műveletek futásának megszakítása

A **`Task`**ok nemcsak elindíthatók, le is lehet állítani őket, ha utólag rájövünk, hogy (már) nincs szükség a futásukra. Például ha egyszerre több **`Task`** segítségével hajtunk végre egy keresést, és csak az elsőként

megtalált objektum, erőforrás a fontos számunkra, akkor nincs értelme tovább futtatni a többit, amelyek még keresgélnek.

A TPL együttműködésre alapozó megszakítási rendszerrel működik, vagyis jelezhetjük egy Tasknak, hogy szeretnénk megszakítani. Ha a **Task**ban lévő metódus figyel erre a kérésre, akkor megszakíthatja saját magát. Továbbá egy Task létrehozáskor vagy indításkor (**StartNew** esetén) megadhatunk egy olyan objektumot, mely jelzi, hogy esetleg mire sorra kerülne a **Task**, már nem is kell elindítani, mert a művelet megszakítását kérték.

A keretrendszer egy **CancellationTokenSource**-on, illetve az annak egy tulajdonságából kiolvasható **CancellationToken** típusokon keresztül teszi lehetővé a leállítás központi kezelését. Először is egy **CancellationTokenSource** objektumot kell létrehoznunk. Miután létrehozunk egy **Task**ot, átadhatjuk neki a **CancellationTokenSource** példány **Token** tulajdonságát, mely **CancellationToken** típusú. Amennyiben a **Task** tényleges indításakor ez a **Token** már azt jelzi, hogy megszakítást kértek, a **Task** el sem indul. Amennyiben a **Task** már fut, úgy a **Token**ben (és a hozzá tartozó **CancellationTokenSource**-ban) beállt változás nincs hatással a futására, hacsak a programozó nem vizsgálja meg azt.

Megszakítást a **CancellationTokenSource** példány **Cancel** metódusával kérhetünk. Mind a **CancellationToken**, mind a **CancellationTokenSource** rendelkezik egy **IsCancellationRequested** tulajdonsággal, mely azt adja meg, hogy kértek-e már megszakítást:

```
if (!source.IsCancellationRequested)
{
    source.Cancel();
}
```

Kivételkezelés párhuzamosan futó műveleteknél

Kivételek gyakorlatilag bárhol, bármikor keletkezhetnek. Ezek kezelése aszinkron illetve párhuzamosított (vagy többszálú) esetben nehézségekbe ütközik, ugyanis ha a párhuzamosan futó metódusok nem készültek fel a kivétel kezelésére, az előbbi metódus hívója hol kaphatná azt el? A hívott metódus párhuzamosan fut, vagyis a **Task** indításakor a létrehozó metódus talán már nem is rendelkezik referenciával a kivételt dobó **Task**ra.

Éppen emiatt a Task példányok képesek elnyelni a kivételeket. Alapesetben egy **Task**ban keletkezett kivétel nem jut ki a **Task**ból. Természetesen ha az például egy **OutOfMemoryException**, akkor sokat nem segít, hogy a Task elnyelte. Akkor fogjuk megkapni a Task belsejében keletkezett kivételt, amikor megpróbálunk hozzáférni a **Task** eredményéhez (**Result** tulajdonság), vagy esetleg várakozni kezdünk a **Task**ra (**Wait**, **WaitAll**, **WaitAny**), így tehát ezeket az utasításokat érdemes **try…catch** blokkba csomagolni!

Alternatív megoldásként a **Task** objektum **IsCancelled** és **IsFaulted** tulajdonságait vizsgálhatjuk: előbbi azt mondja meg, hogy leállították-e a **Task**ot, utóbbi pedig, hogy kezeletlen kivétel miatt szakadt-e meg. Egyik esetben sem férhetünk hozzá az eredményhez.

Amennyiben egy **Task**ban kivétel keletkezett, azt egy **AggregateException** formájában kapjuk meg. Ennek **InnerExceptions** tulajdonságában találhatjuk meg a tényleges kivételeket, amelyek a **Task**, illetve az abban indított **Task**ok (ha voltak) megszakadását okozták. Így ha egy **Task** elindít több másikat, és a belső **Task**ok megszakadása miatt nem sikerült a külső **Task** végrehajtása, akkor is értesülünk minden egyes kivételről.

A következő gyakorlatban a korábbi prímkereső alkalmazást megszakíthatóvá tesszük, majd beleépítjük azt a lehetőséget, hogy automatikusan megszakítsa a feldolgozást, amennyiben az túlnyúlik egy adott időintervallumon. Utóbbi esetben a TPL néhány, itt még be nem mutatott képességét is felhasználjuk. Az egyszerűbb kód kedvéért a prímkereső művelet ezúttal nem aszinkron lesz, csak a párhuzamosításra koncentrálunk.

***Gyakorlat**: Párhuzamos Taskok megszakítása*

A prímkereső alkalmazás átalakításához kövesd az alábbi lépéseket:

1. Nyisd meg az előző feladat végén elmentett projektet vagy a fejezethez tartozó kód **`BeginTPLCancel`** könyvtárában lévőt!
2. Nyisd meg a **`MainPage.xaml`** fájlt, és adj hozzá egy-egy eseménykezelőt az „Indítás megszakíthatósággal”, „Megszakítás” és „Indítás időkorláttal” feliratú gombok **`Click`** eseményeihez! A három **`Button`** leírása most nagyjából így kell, hogy kinézzen:

```
<Button Click="btnStartWithCancel_Click" Content="Indítás megszakíthatósággal"
    Margin="0,0,0,0" />
<Button Click="btnCancel_Click" Content="Megszakítás" />
<Button Click="btnStartWithTimeout_Click" Content="Indítás időkorláttal"  />
```

3. Nyisd meg a **`MainPage.xaml.cs`** fájlt! Adj hozzá az osztályhoz egy **`CancellationTokenSource`** mezőt **`cts`** néven! Az alábbi kódot kell beillesztened (bárhova az osztályon belül, de ne metódus belsejébe):

```
CancellationTokenSource cts;
```

4. Helyezd el az „Ötödik fázis” szekcióban a **`btnStartWithCancel_Click`** metódust, és az általa hívott **`FindPrimesParallel`** metódust! A következő kódot kell beillesztened:

```
List<int> FindPrimesParallel(int min, int max, CancellationToken token)
{
    List<int> results = new List<int>();
    List<Task> tasks = new List<Task>();
    int i = min;
    while (!token.IsCancellationRequested && i < max)
    {
        tasks.Add(Task.Factory.StartNew(() =>
        {
            if (IsPrime(i)) results.Add(i);
            i++;
        }, token));
    }
    try
    {
        Task.WaitAll(tasks.ToArray());
    }
    catch (AggregateException agx)
    {
        Dispatcher.RunIdleAsync(_ => tbkResult.Text =
        agx.InnerExceptions.Count + "db task megszakítva.");
    }
    return results;
}

async void btnStartWithCancel_Click(object sender, RoutedEventArgs e)
{
    tbkResult.Text = "Elindítva";
    cts = new CancellationTokenSource();

    int min = int.Parse(tbFirstNum.Text);
    int max = int.Parse(tbLastNum.Text);

    var t = new Task<List<int>>(() => FindPrimesParallel(min, max, cts.Token));
    t.Start();
    await t;
```

```
    if (t.Status == TaskStatus.RanToCompletion)
    {
        gvResult.Items.Clear();
        foreach (var i in t.Result.Take(500)) gvResult.Items.Add(i);
    }
    else tbkResult.Text = "Leállítva";
}
```

5. Helyezd el ugyanebben a szekcióban a **`btnCancel_Click`** eseménykezelő metódust is:

```
private void btnCancel_Click(object sender, RoutedEventArgs e)
{
    if (!cts.IsCancellationRequested)
    {
        cts.Cancel();
    }
}
```

6. Helyezd el az előbbi metódusok módosított változatát (**`btnStartWithTimeout_Click`** és **`FindPrimesParallelFor`**) a „Hatodik fázis” szekcióban! A következő kódot kell beillesztened:

```
List<int> FindPrimesParallelFor(int min, int max, CancellationToken token)
{
    List<int> results = new List<int>();
    Parallel.For(min, max, new Action<int, ParallelLoopState>((i, pls) =>
    {
        if (token.IsCancellationRequested) pls.Break();
        if (IsPrime(i)) results.Add(i);
    }));
    return results;
}

private void btnStartWithTimeout_Click(object sender, RoutedEventArgs e)
{
    tbkResult.Text = "Elindítva";
    cts = new CancellationTokenSource();

    int min = int.Parse(tbFirstNum.Text);
    int max = int.Parse(tbLastNum.Text);
    int timeout = int.Parse(tbTimeout.Text);

    Task[] ts = new Task[2];
    ts[0] = Task.Factory.StartNew<List<int>>(() =>
        FindPrimesParallelFor(min, max, cts.Token));
    ts[1] = Task.Delay(timeout);
    int succeededIndex = Task.WaitAny(ts);

    if (succeededIndex == 0)
    {
        gvResult.Items.Clear();
        foreach (var i in ((Task<List<int>>)ts[0]).Result.Take(500))
            gvResult.Items.Add(i);
    }
    else
    {
        cts.Cancel();
        tbkResult.Text = "Leállítva";
    }
}
```

7. Fordítsd le, és futtasd az alkalmazást! Ha a fordító hibát jelez, próbáld meg kijavítani az Errors ablakban található javaslatok segítségével! Ha nem sikerül, nyisd meg az ehhez a fejezethez tartozó kódok közül az **`EndTPLCancel`** projektet, és az abban lévő kóddal összehasonlítva keresd meg a megoldást, vagy egyszerűen írd felül a hibás kódot! Ha már fut, kattints az „Indítás megszakíthatósággal" gombra, majd rögtön utána a „Megszakítás" gombra. A bal oldali panel alján lévő [eredmény] felirat helyén megjelenik, hogy hány Taskot kellett leállítani. Ezután kattints az „Indítás időkorláttal" gombra!

Ha a bal oldali panel alján lévő szövegdobozban elég nagy szám található, a feldolgozás lefut, és megjelenik az eredmény a jobb oldalon. Ha csökkented a szövegdobozban lévő számot (pl. 50 ezredmásodpercre), és ismét elindítod a feldolgozást, eredmények nem jelennek meg, mindössze az alsó szövegblokkban (**`tbkResult`**) látható egy szöveg, amely arról tájékoztat, hogy a feldolgozás megszakadt.

Hogyan működik?

A 3. lépésben hozzáadott **`CancellationTokenSource`** referencia (**`cts`**) azért fontos, hogy az indítandó **`Task`**ok leállíthatóak legyenek. A 4. lépésben megadott **`btnStartWithCancel_Click`** metódus fő feladata ezúttal is a prímkereső metódus (**`FindPrimesParallel`**) futtatása. Mivel az most nem aszinkron (pontosabban azért, mert nem **`Task`**ot ad vissza), ezért a híváskor manuálisan csomagoljuk be egy **`Task`**ba, hogy aszinkron módon indíthassuk! Fontos még, hogy a metódus indítása előtt a **`cts`** nevű **`CancellationTokenSource`**-ot példányosítjuk, és ennek **`Token`** tulajdonságát átadjuk a **`FindPrimesParallel`** metódusnak.

A **`FindPrimesParallel`** metódus ezúttal egy **`while`**-ciklussal dolgozik. A ciklus egyik megállási feltétele, hogy a kapott token (ami **`CancellationToken`** típusú) megszakítást jelezzen. Így ha a ciklus futása közben érkezik egy megszakítás, a hátralévő számokra már nem is jönnek létre a vizsgálatukat végző Task példányok. A **`Task`**ok a létrehozáskor (**`Task.Factory.StartNew`**) szintén megkapják a tokent, így ha már létrejött egy **`Task`**, de még nem indult el, mire megszakításkérés érkezett, a keretrendszer már el sem indítja azt. Mivel az el nem indított **`Task`**ok bevárásakor hibát kapunk, így a **`WaitAll`** hívást egy **`try…catch`** blokkba tettük, melynek **`catch`** ága az **`AggregateException`** típusra figyel. Ha kivétel volt, azt a **`Dispatcheren`** keresztül a **`tbkResult`** szövegblokkban jelenítjük meg.

Az 5. lépésben hozzáadott metódus felelős azért, hogy ha a felhasználó a „Megszakítás" gombra kattint, a tokeneken keresztül a még le nem futott Task példányok megkapják a jelzést, hogy már nem érdemes elindulniuk.

A 6. lépésben hozzáadott **`FindPrimesParallelFor`** metódusban a **`Parallel`** osztály **`For`** metódusának segítségével egyszerűsítjük le a kódot. Ha egy lista egyes elemeit szeretnénk egy-egy **`Task`**ként feldolgozni, nincs szükség arra, hogy manuálisan hozzunk létre Task példányokat; a **`Parallel`** osztály néhány statikus metódusa éppen erre való. Mivel megszakíthatóvá akarjuk tenni ezt is, ezért a **`For`** metódusnak egy olyan lambda kifejezést adunk át, mely két paramétert fogad: az első a ciklusváltozó egész szám (**`i`**), a második egy **`ParallelLoopState`** típusú objektum (**`pls`**). Utóbbi segítségével megszakíthatjuk a ciklus futását az összes magon egyszerre (**`pls.Break()`**), ha a **`CancellationToken`** jelzi, hogy megszakítást kértek.

A **`btnStartWithTimeout_Click`** metódus egy **`Task`**ba csomagolva elindítja a **`FindPrimesParallelFor`** metódust, majd rögtön egy másik **`Task`**ot is indít, mely azon kívül semmit nem csinál, mint hogy a **`tbTimeout`** szövegdobozban megadott időmennyiség után befejeződik (**`Task.Delay()`**). Ezután a **`WaitAny`** metódussal figyeli, hogy a két párhuzamosan futó feladat közül melyik ér előbb véget. Ha a prímkeresés futott le gyorsabban, az első 500 elemet kiírja a **`GridView`**-ba, ha viszont a „timeout" telt le előbb, megszakítja a háttérben futó számítást, és ezt jelzi a **`tbkResult`** szövegblokkon keresztül.

Ebben a fejezetben csak nagyon kis részét mutattuk be a Task Parallel Library teljes arzenáljának. Ez a bevezetés azonban elég ahhoz, hogy bátran nekivághass a Windows Runtime-ban jelen lévő aszinkron és parallel képességek használatának.

Összefoglalás

A C# 5.0 aszinkron képességei mind kényelmi, mind produktivitási szempontból a fejlesztők előnyére válnak. Az **`async`** és **`await`** kulcsszavak, illetve a **`void`**, **`Task`** vagy **`Task<TResult>`** értéket visszaadó metódusok segítségével átlátható, ugyanakkor a hívó szálat nem blokkoló metódusokat tudunk írni. Azokban az esetekben, amikor pedig nem(csak) a hívó szál tehermentesítése a cél, hanem egy vagy több feladat párhuzamos végrehajtása, szintén a **`Task`** és **`Task<TResult>`** osztályok állnak rendelkezésünkre.

5. Windows 8 XAML alapismeretek

Ebben a fejezetben az alábbi témákat ismerheted meg:

- A XAML nyelv szintaktikája: hogyan lehet vezérlőket (és akár egyéb objektumokat) deklarálni és a tulajdonságait beállítani
- Gyakran használt vezérlők felhasználása (tulajdonságok, események)
- Vezérlők dinamikus elrendezése panelek segítségével

Történelmi áttekintés

Mikor arra kerül a sor, hogy felhasználói felületet kell készítenünk egy alkalmazáshoz, akkor elengedhetetlen a XAML technológia ismerete. A XAML az Extensible Application Markup Language kifejezés rövidítése. Ezt a Microsoft 2007-ben vezette be a .NET 3.0 WPF (Windows Presentation Foundation) megjelenésével. A XAML egy XML alapú leírónyelv felhasználói felületek struktúrájának leírására.

Az elődnek számító WinForms-ban a felhasználói felület kialakítása az esetek többségében a vizuális szerkesztőfelületen vezérlők „összeklikkelgetéséből" vagy kódból történő dinamikus generálásból állt. Ez amellett, hogy nagyon hosszú, nehezen kezelhető, és átláthatatlan kódot produkált, még sokszor az üzleti logika és a grafikus felület működésének egybefolyását is jelentette, aminek következtében az alkalmazás tesztelése és esetleges javítása, módosítása vagy bővítése is nagymértékben megnehezedett.

Ennek orvoslására vezette be a Microsoft a XAML-t, amely kényelmesen, deklaratív formában teszi lehetővé a felületek leírását. A deklaratív leírás szemléltetésére a legjobb hasonlat, hogy a XAML segítségével (a C# nyelvvel ellentétben) azt írjuk le, hogy mit szeretnénk megjeleníteni, és nem azt, hogyan.

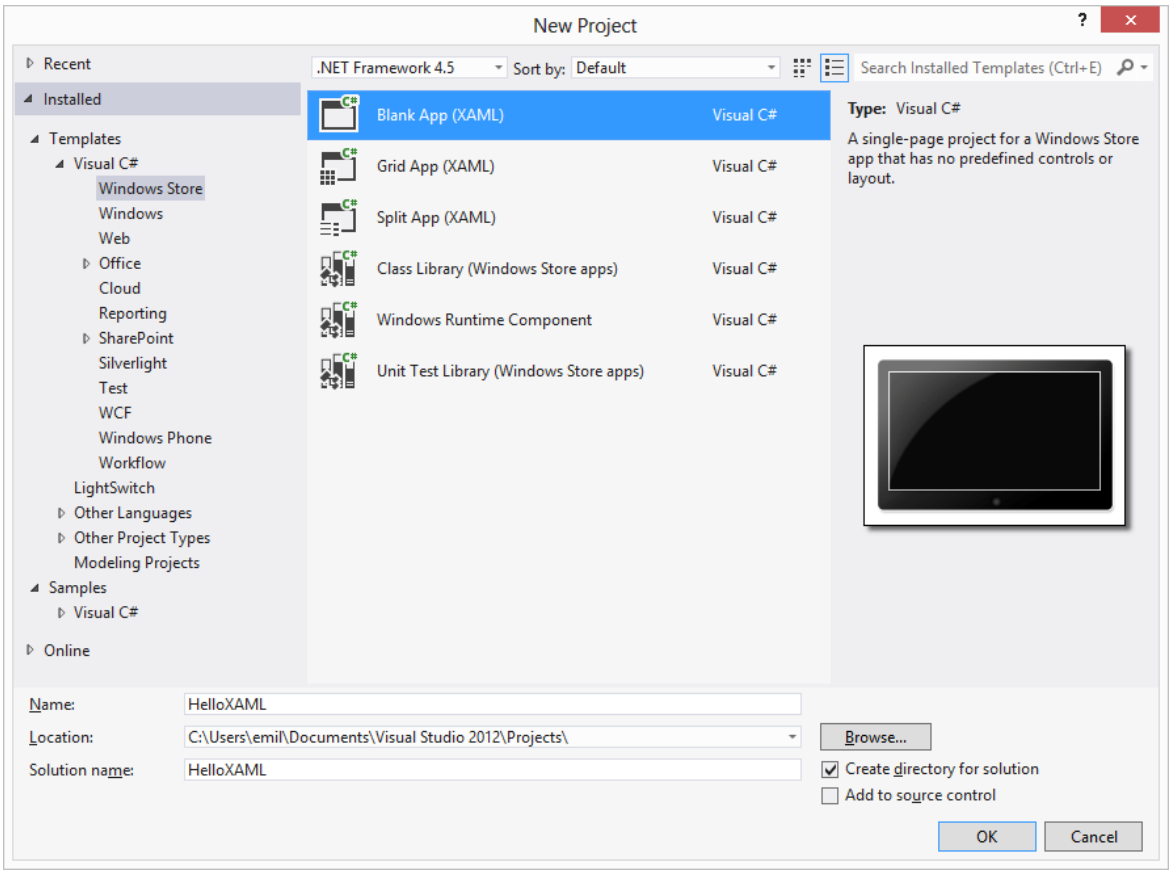
XAML szintaktika

Néhány példán keresztül ismerkedj meg a XAML nyelv szintaxisának alapjaival! Először is, indítsd el a Visual Studio 2012-t, és hozz létre egy új üres Windows 8 stílusú projektet a File | New |Project paranccsal! A New Project dialógusban az 5-1 ábrán látható módon válaszd ki a Visual C# alatt a Windows Store kategóriából a Blank App (XAML) sablont!

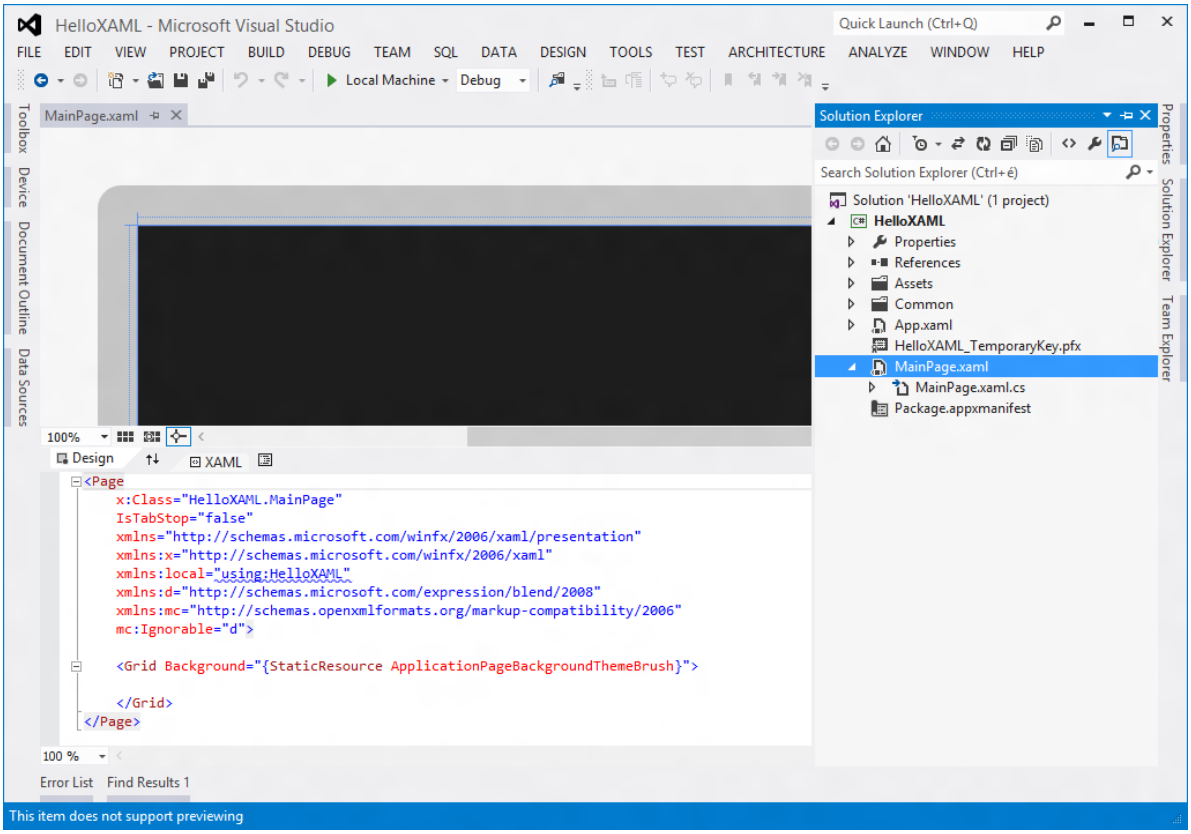
Amikor megnyílik a projekt, először az **`App.xaml.cs`** fájllal fogod szembe találni magad, de ezzel most nem kell foglalkoznod, zárd be azt, és helyette nyisd meg a Solution Explorerből kiválasztva a **`MainPage.xaml`** fájlt, amint azt az 5-2 ábrán látod!

Mint látható, a „XAML fa" gyökéreleme a **`Page`**, amely sok attribútumot és egy **`Grid`** elemet tartalmaz. Ezek közül az attribútumok közül az alábbiakat érdemes kiemelni:

```
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:local="using:HelloXAML"
```



5-1 ábra: Új üres projekt létrehozása



5-2 ábra: A MainPage.xaml fájl a szerkesztőben

Az **`xmlns`** kezdetű sorok névtereket jelölnek (XML NameSpace). Ezek hasonlóak, mint C# nyelv névterei. A felső **`xmlns`** után nincs semmilyen prefix megadva, ez a névtér lesz az alapértelmezett: ha egy objektum

deklarálásakor nem használunk prefixet, akkor az az alapértelmezett névtérben van. Az **xmlns** után szereplő URL a látszattal ellentétben nem egy weboldal címe (ki lehet próbálni, mi történik, ha bemásolod a böngészőbe), csupán egy ilyen érdekes módon megadott névtér.

> *Ez egy úgynevezett álcázott névtér, ami arra jó, hogy egy szerelvény készítője elfedje a szerelvényben található valódi névtereket, és ilyen URL-nek látszó címekkel könnyebbé tegye egy esetleges verzióváltás esetén az átállást abban az esetben, ha az álca mögött a valódi névtér-hierarchia megváltozott.*

A kódrészlet második sorában a **local** névtér definiálása látható. Minden, utólag az oldalhoz adott névtérnek adnunk kell egy prefixet, és ezek nem ütközhetnek. Az attribútum értékében egy általános, „álcázatlan" névtérre való hivatkozás látható. Itt ez a C#-hoz hasonló módon **using** kulcsszóval kezdődik, melyet egy kettőspont, majd a használni kívánt névtér teljes útvonala követ.

Az alapértelmezett névtérben lévő vezérlők nem igényelnek prefixet:

```
<BuiltInControl />
```

Azonban egy saját vezérlő (saját névtérben) csak prefixszel érhető el a XAML kódban:

```
<local:CustomControl />
```

Egyszerűen létrehozhatsz egy vezérlőt – jelen esetben egy nyomógombot. Ehhez mindössze annyit kell tenned, hogy a **Grid** elem nyitó és záró tagjai közé beszúrsz **Button** elemet:

```
<Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
    <Button>Hello XAML</Button>
</Grid>
```

> *Ha a C# nyelvben kellene létrehoznod ezt a gombot, beállítani a tulajdonságait, majd a kívánt panel gyermekelemeihez hozzáadni, az több kódsor leírását jelentené. Ugye mennyivel barátságosabb egy kifejezetten erre a célra készített nyelvvel megtenni mindezt?*

A művelet eredményeként a nyomógomb megjelenik a tervezőfelületen, amint azt az 5-3 ábrán láthatod.

A gomb tartalmát (mely jelen esetben egy egyszerű szöveg) a nyitó és záró tagjai közé írva adtuk meg, azonban ennél adódhatnak bonyolultabb szituációk is.

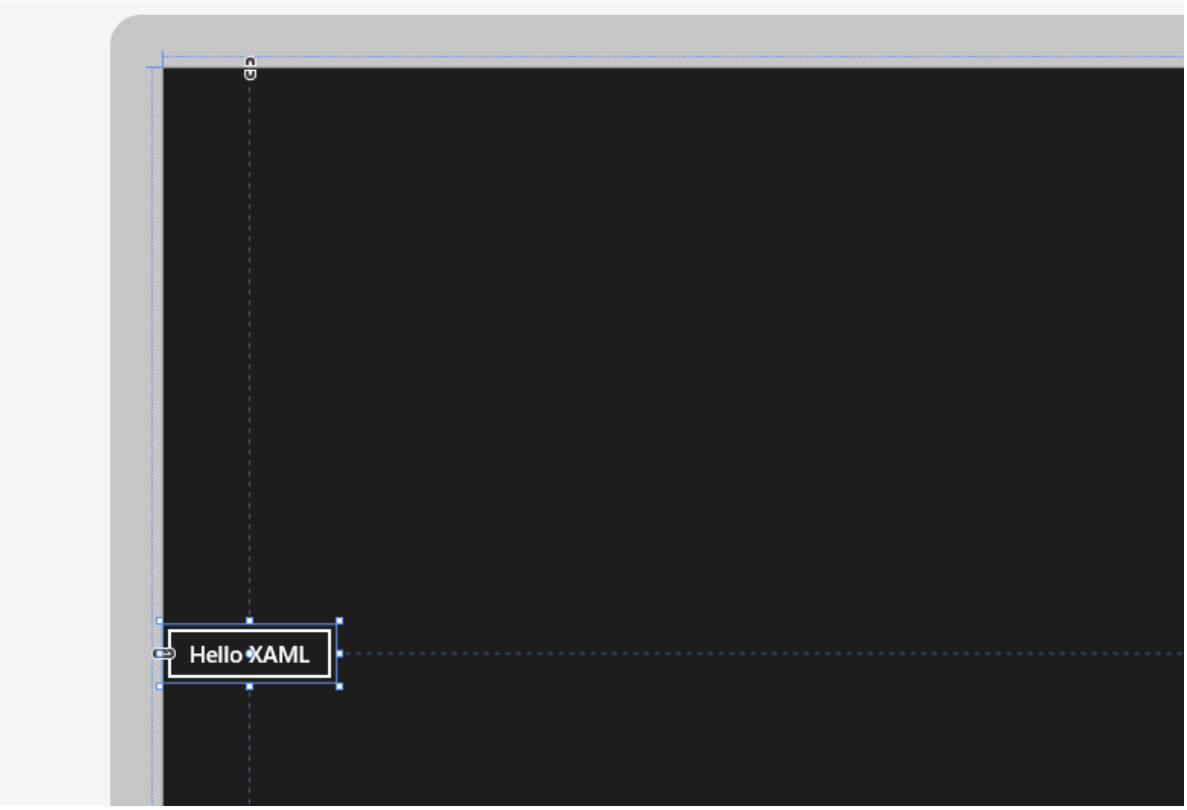
A gomb tartalmát a **Content** tulajdonságán keresztül tudjuk állítani, a fenti kódpéldában ezt adtuk meg implicit módon a „Hello XAML" szöveg beírásával. Emellett azonban lehetőségünk van azt attribútumként is megadni, vagy akár explicit módon kifejteni. Ez utóbbira akkor van szükség, ha egy tulajdonságnak nem tudunk egyszerű szöveges értéket adni, hanem azt csak bonyolultabb módon tudjuk leírni.

Az attribútumos értékadás a következőképpen néz ki:

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

A gomb tartalmát itt a nyitó tagon belül, XML attribútumként adjuk meg. Ez nemcsak a **Content**re érvényes, hanem a gomb (és egyéb vezérlők) összes többi tulajdonságára is. Ha például a nyomógomb méretének megadására volna szükségünk, azt a **Width** és a **Height** attribútumokkal tehetjük meg:

```
 <Button Content="Hello XAML" Width="200" Height="50"></Button>
```



5-3 ábra: Button elhelyezése a felületen

Explicit módon is kifejthetjük a **`Content`** tulajdonságot:

```
<Button>
    <Button.Content>
        Hello XAML
    </Button.Content>
</Button>
```

Itt a nyitó és záró Button tagok között egy újabb XML elemet definiálunk **`Button.Content`** néven, amely a *[VezérlőTípusa].[TulajdonságNeve]* sablont követi, majd azon belül adjuk meg a tulajdonság értékét.

A gomb **`Content`** tulajdonsága **`System.Object`** típusú, azaz bármi lehet. Ez azért van, mert XAML-ben az elemek a végtelenségig egymásba ágyazhatóak, és például a gomb nemcsak szöveget tartalmazhat, hanem gyakorlatilag bármit, akár egy 3D-ben forgó videót is. Ezt a következő – kevésbé elrugaszkodott – példa szemlélteti is, melyben a gomb tartalma egy formázott szöveg, vagy egy kicsit általánosabban tekintve, egy másik vezérlő.

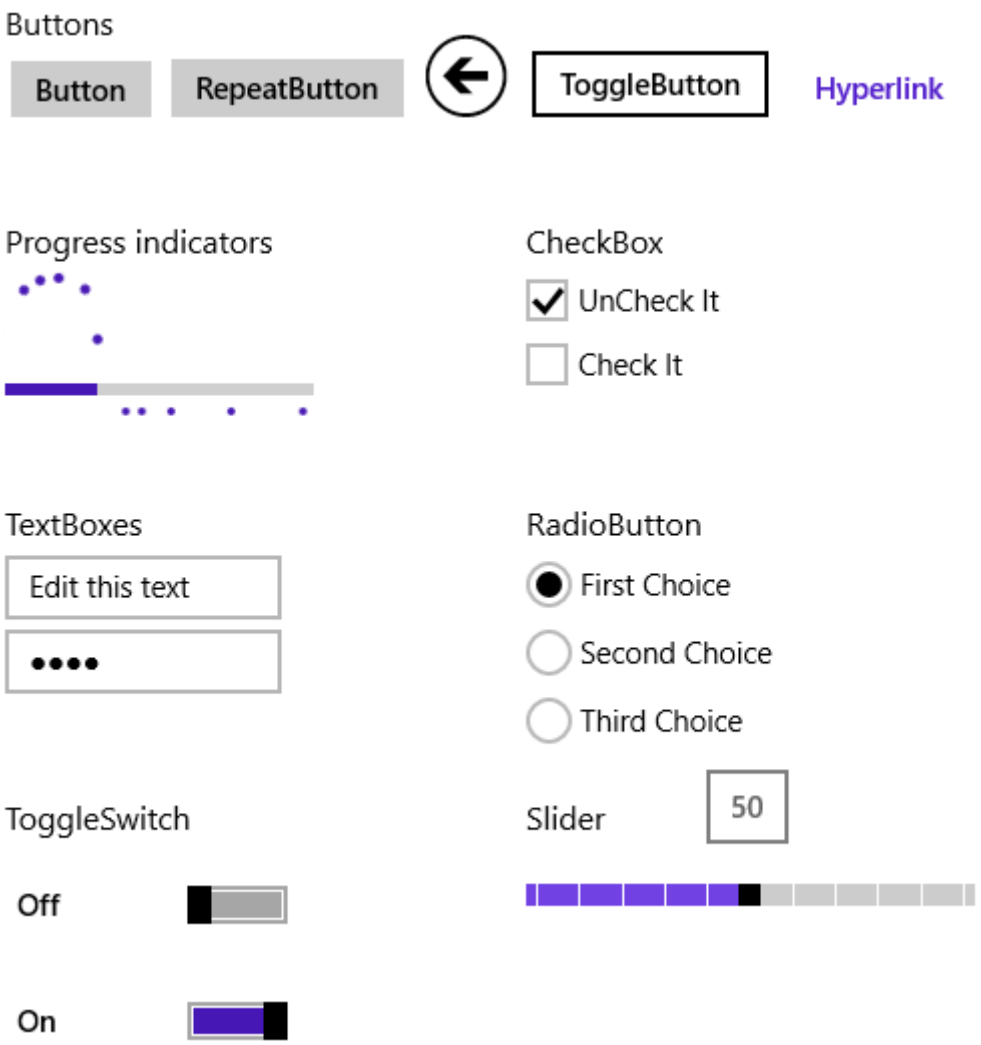
```
<Button>
    <TextBlock FontSize="32" Foreground="Green" Text="Hello XAML" />
</Button>
```

A **`Button`** tag belsejébe írt XAML tartalom a nyomógomb **`Content`** tulajdonságába lesz irányítva.

Beépített vezérlők

Miután megismerhettük a XAML alapvető szintaktikai elemeit, érdemes pár szót ejteni a beépített vezérlőkről, hiszen túlnyomó részben ezek felhasználásával fogjuk alkalmazásunk felhasználói felületét reprezentálni.

A beépített vezérlők egy közel teljes „felsorolását” az 5-4 ábra szemlélteti.



5-4 ábra: Beépített vezérlők

Button

A nyomógomb az elvárásoknak megfelelően egy „olyan valami”, amit meg tudunk nyomni. Azért fogalmazok így, mert a XAML-nek köszönhetően ennél konkrétabb specifikációt nem lehet adni egy gombra, ugyanis a kinézet 100%-ban testre szabható, amit mi sem mutat jobban, mint például hogy az 5-4 ábrán látható első és harmadik vezérlő egészen pontosan ugyanaz a **`Button`** típusú vezérlő, csak más kinézettel.

RepeatButton

A **`RepeatButton`** különlegessége abban rejlik, hogy ha megnyomjuk és nyomva tartjuk, akkor úgy viselkedik, mintha folyamatosan nyomkodnánk. A **`Button`** vezérlővel szemben, ami csak egyetlen **`Click`** eseményt küld, ha lenyomjuk, a **`RepeatButton`** folyamatosan küldi a **`Click`** eseményeket, amíg csak nyomva tartjuk. Ennek a vezérlőnek két fontos tulajdonsága van, a **`Delay`** és az **`Interval`**. Előbbivel ezredmásodpercekben megadhatjuk, hogy a legelső **`Click`** esemény után mennyi idő teljen el, mielőtt elkezdené „szórni” magából a további **`Click`** eseményeket. Ez hasonló, mint a szövegszerkesztőknél a törlés, ahol a backspace megnyomásakor először csak egy karaktert töröl vissza, majd ha kellő ideig nyomva tartjuk azt, akkor elkezd nagyobb tempóban törölni. Az **`Interval`** tulajdonság a tempót fogja meghatározni, azt hogy milyen gyakorisággal küldjön nyomva tartás esetén **`Click`** eseményeket a gomb. Ezt az értéket szintén ezredmásodpercekben kell megadnunk.

HyperlinkButton

Ennek a gombnak az a különlegessége, hogy a **NavigateUri** tulajdonságán keresztül megadhatunk neki egy címet, és amikor rákattintunk, az adott címre navigál. Ha ez egy weboldal címe, akkor megnyitja a böngészőt. Ha egy „mailto" hivatkozás, akkor megnyitja a levelezőalkalmazást az adott címzettel.

> *Lényegében ugyanazt érjük el ennek a gombnak a használatával, mintha a **Launcher** API **LaunchUriAsync** metódusát hívnánk meg egy sima **Button** eseménykezelőjéből. Ezzel az API-val a könyv későbbi részében találkozhatsz.*

ToggleButton

Ez a gomb két vagy három állapotot tud felvenni az **IsThreeState** tulajdonságának értékétől függően. Az aktuális állapotát vagy az **IsChecked** tulajdonságán keresztül tudjuk lekérdezni, vagy a **Checked**, **Unchecked** és **Indeterminate** eseményekre való feliratkozással.

ToggleSwitch

Ez a vezérlő a **ToggleButton** vezérlőhöz hasonlónak tűnik, de a megjelenésén túl a funkcionalitásában is eltér. Annyiban kevesebbet tud, mint a **ToggleButton**, hogy ez szigorúan csak két állapotot vehet fel, (**On** és **Off**). Az aktuális állapotát vagy az **IsOn** tulajdonságon keresztül lehet lekérdezni, vagy a **Toggled** eseményre való feliratkozással. Annyiban többet tud, mint a **ToggleButton**, hogy külön megjelenítendő tartalmat rendelhetünk a ki- és bekapcsolt állapotokhoz az **OnContent** és **OffContent** tulajdonságokon keresztül.

CheckBox

A **CheckBox** funkcionalitása és működése valószínűleg senkinek nem fog nagy meglepetést okozni, lényegében ugyanaz, mint a **ToggleButton**, csak más kinézettel. Ugyanúgy három állapotot vehet fel, a harmadik köztes állapotot az **IsThreeState** tulajdonsággal lehet engedélyezni, és ugyanúgy használhatók az **IsChecked** tulajdonság, illetve a **Checked**, **Unchecked** és **Indeterminate** események.

RadioButton

A **RadioButton** hasonlóan működik, mint a **CheckBox** és a **ToggleButton**. Abban különbözik azoktól, hogy több **RadioButton** vezérlőt elhelyezve a felületen és azokat azonos csoportba sorolva – a **GroupName** tulajdonságukat azonos értékre állítva – a csoporton belül egyszerre mindig csak egy **RadioButton** lehet kijelölve.

TextBlock

Ezt a vezérlőt formázott szövegek megjelenítésére tudjuk használni. Lehetőséget ad a betűtípus tulajdonságainak beállítására (betűtípus: **FontFamily**, méret: **FontSize**, betűköz: **FontStretch**, stílus: **FontStyle**, vastagság: **FontWeight**).

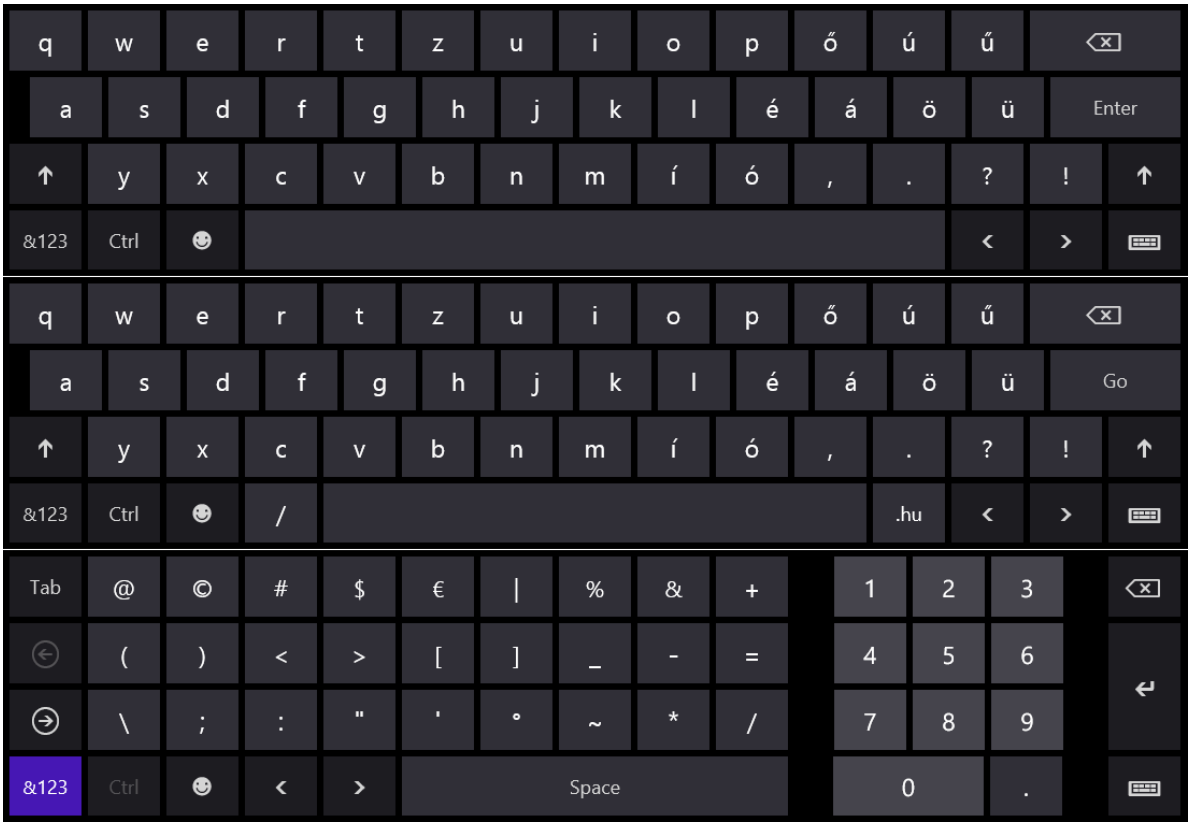
A szöveget balra, középre és jobbra lehet rendezni vagy sorkizárással igazítani (**TextAlign**). A **TextWrap** tulajdonsággal megadható, hogy a vezérlő méretéhez igazodva több sorba tördelje a szöveget, ha az nem férne el egy sorban. A **TextTrimming** tulajdonsággal lehet megadni, hogy a vezérlőbe már ki nem férő tartalmat inkább három ponttal helyettesítse. A szöveg sorközét a **LineHeight** tulajdonsággal lehet módosítani, a színét pedig a **Foreground** tulajdonságon keresztül adhatjuk meg.

> *A **TextBlock** mindezek mellett arra is lehetőséget nyújt, hogy egyes részeit külön-külön formázzuk, és paragrafusokba rendezzük a tartalmát. Ezt az **Inlines** tulajdonságon keresztül lehet megtenni, azonban ennek tárgyalása már kívül esik ennek a fejezetnek a témáján.*

TextBox

Egyszerű szöveg bevitelére a **`TextBox`** vezérlő használható, mely ugyanazokkal a tulajdonságokkal rendelkezik, mint bármelyik másik keretrendszer **`TextBox`** vezérlője, és ezt még néhány Windows 8 specifikus tulajdonság egészíti ki.

Az egyik ilyen az **`InputScope`**, amelyen keresztül megadhatjuk, hogy a szövegdobozba milyen jellegű információt várunk. Ennek köszönhetően amikor a felhasználó egy táblagépen belekattint egy **`TextBox`** vezérlőbe, a várt információ jellegének megfelelő virtuális billentyűzetet fog megjeleníteni a rendszer. Ezek egy részét (Default, Url, Number) az 5-5 ábra szemlélteti.



5-5 ábra: InputScope hatása a virtuális billentyűzetre

A vezérlő helyesírás-ellenőrzést is biztosít, amiről azonban eldönthetjük, hogy szeretnénk-e használni vagy sem. Ez a szolgáltatás két részből áll,egyfelől jelzi, ha valamit helytelenül írtunk, másfelől azt automatikusan ki is tudja javítani. Ezeket rendre az **`IsSpellCheckEnabled`** és az **`IsTextPredictionEnabled`** tulajdonságokkal tudjuk engedélyezni.

PasswordBox

Ez egy speciális **`TextBox`**, mely elrejti a beleírt szöveget. A **`PasswordChar`** tulajdonságon keresztül lehet megadni azt, hogy milyen karakterrel rejtse el a beírt karaktereket. A vezérlőbe írt szöveget (a ténylegest, nem a képernyőn megjelenőt) a **`Password`** tulajdonságon keresztül lehet kiolvasni. A Microsoft gondolt az érintőkijelzős felhasználókra – akik nagyobb eséllyel nyúlnak mellé észrevétlenül a gépeléskor–, így lehetőség van egy kis gombbal ideiglenesen láthatóvá tenni a beírt szöveget. Azt pedig, hogy ez a gomb elérhető legyen-e, az **`IsPasswordRevealButtonEnabled`** tulajdonsággal tudjuk beállítani.

Slider

Ha egy számszerű adat módosítását kell lehetővé tenni a felületen, valószínűleg kényelmesebb megoldást jelenthet egy szövegdoboznál a csúszka használata. A **`Slider`** vezérlő ezt valósítja meg. Lehetőség van a minimális (**`Minimum`**) és maximális (**`Maximum`**) értékek megadására, kis és nagy lépésközök (**`SmallChange`**, **`LargeChange`**) beállítására és vizuális skálaegységek (**`TickFrequency`**) definiálására is. A **`StepFrequency`**

tulajdonsággal pedig azt lehet megadni, hogy a rendelkezésre álló skálán milyen részletességgel vehessen fel értékeket a vezérlő (például csak ötösével vagy tizesével vehessen fel értékeket).

ProgressBar és ProgressRing

Windows 8-ban nagyon nagy hangsúlyt fektetett a Microsoft arra, hogy a felület mindig válaszképes legyen, egy gombnyomás hatására az ne fagyjon meg addig, amíg például egy webszerviz-hívás eredménye visszatér a szerver oldalról.

A **`ProgressBar`** segítségével egy folyamat előrehaladását lehet jelezni. Ha nem tudjuk előre, hogy mikor lesz vége a futó folyamatnak, akkor az **`IsIndeterminate`** tulajdonság **`true`**-ra állításával a vezérlő folyamatosan animálódik. Így a felhasználó azt érzékeli, hogy az adott háttértevékenység még mindig folyamatban van.

A **`ProgressRing`** már csak ilyen határozatlan állapotot tud felvenni, tehát konkrét értéket nem tud mutatni. Ezt a vezérlőt az **`IsActive`** tulajdonságán keresztül lehet aktiválni.

> *Az itt felsorolt vezérlők csupán a legalapvetőbb „önálló és interaktív" vezérlők, ezeken túl még jó pár vezérlő található az SDK-ban különböző médiatartalmak megjelenítésére, listák kezelésére, felületek elrendezésére stb, de ezek szinte mindegyikéről szó lesz a könyv későbbi fejezeteiben.*

Felületek elrendezése

Ebben a fejezetrészben arról lesz szó, hogy milyen módon tudunk több elemet elrendezni a felületen. A Windows 8 esetében nagyon fontos a felületek dinamikus kialakítása, mert a felbontás 1024x768-tól kezdődően 2560x1440-ig, sőt akár még tovább is terjedhet. Sem az nem előnyös, ha egy alkalmazás nem fér bele a rendelkezésre álló felbontásba, sem az, ha egy nagyobb felbontású kijelző fele szabadon marad a felület egy jelentős részén.

A korábbi mintapélda kapcsán érdemes megfigyelni, mi történik, ha egy másik gombot is lerakunk a felületre:

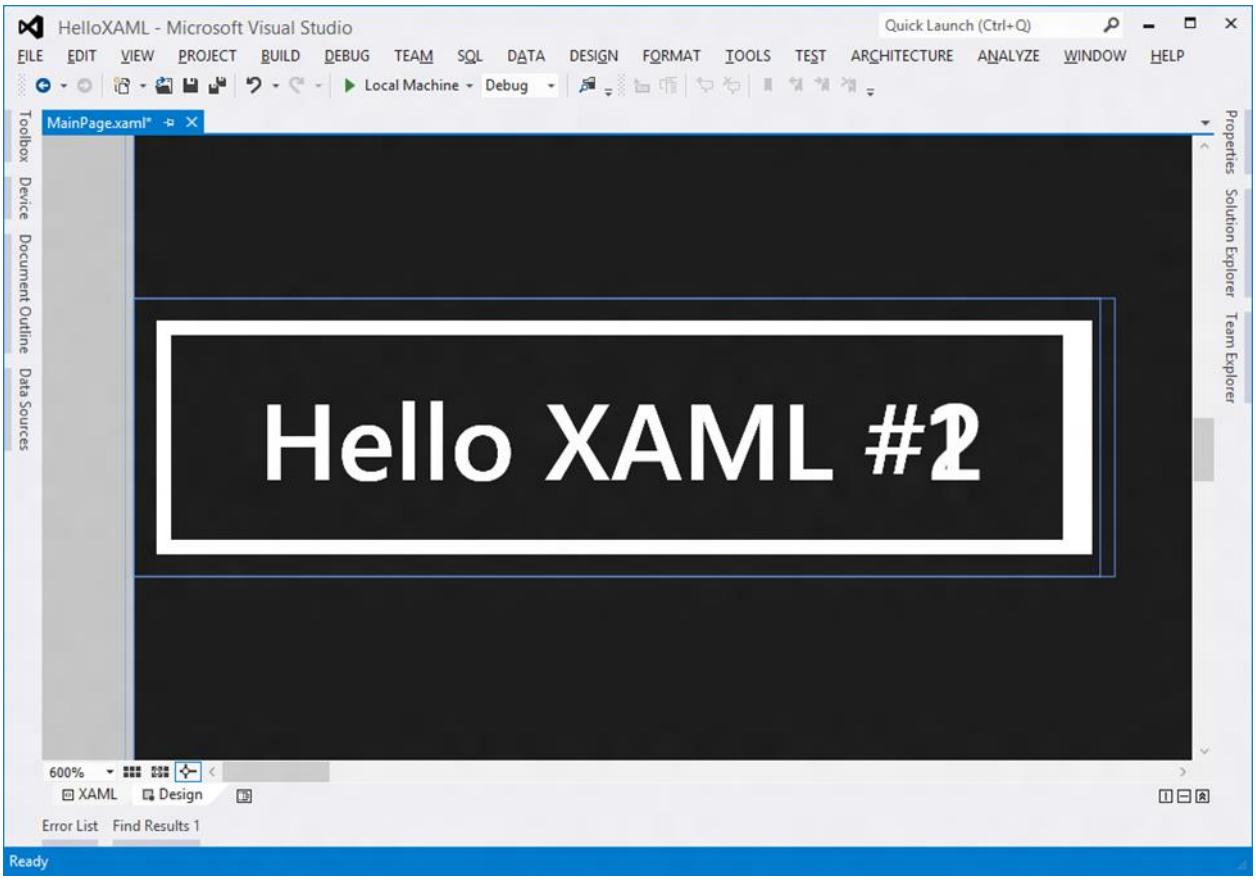
```
<Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
    <Button Content="Hello XAML #1" />
    <Button Content="Hello XAML #2" />
</Grid>
```

A tervezőfelületen (5-6 ábra) az látható, hogy a két gomb egymás felett helyezkedik el.

A XAML-ben az elemeknek nem szokás abszolút pozíciót adni, hanem különböző panelek segítségével ajánlott elrendezni őket. A panelek olyan vezérlők, melyek több másik vezérlőt tartalmazhatnak, és azokat különböző logika alapján dinamikusan rendezik el.

> *Érdemes megjegyezni, hogy lényegében kétféle elem van: az egyik pontosan egy elemet tud tartalmazni (mint például a nyomógomb), a másik pedig valamilyen panel, amely több további elemet tartalmaz, és azokat adott logika alapján rendezi el. Így ha például arra van szükségünk, hogy egy gombon több másik elemet helyezzünk el, akkor a gomb egy panelt tartalmaz és a többi elemet pedig ebben a panelben helyezzük el.*

Négy beépített paneltípussal találkozhatsz, ismerd meg ezeket!



5-6 ábra: Két gomb egymás felett helyezkedik el

Grid

A **Grid** a legjobban testre szabható panel. A **Grid** táblázat, melyben sorokat és oszlopokat definiálhatunk, ezeknek pedig fix, relatív vagy automata szélességet illetve magasságot adhatunk.

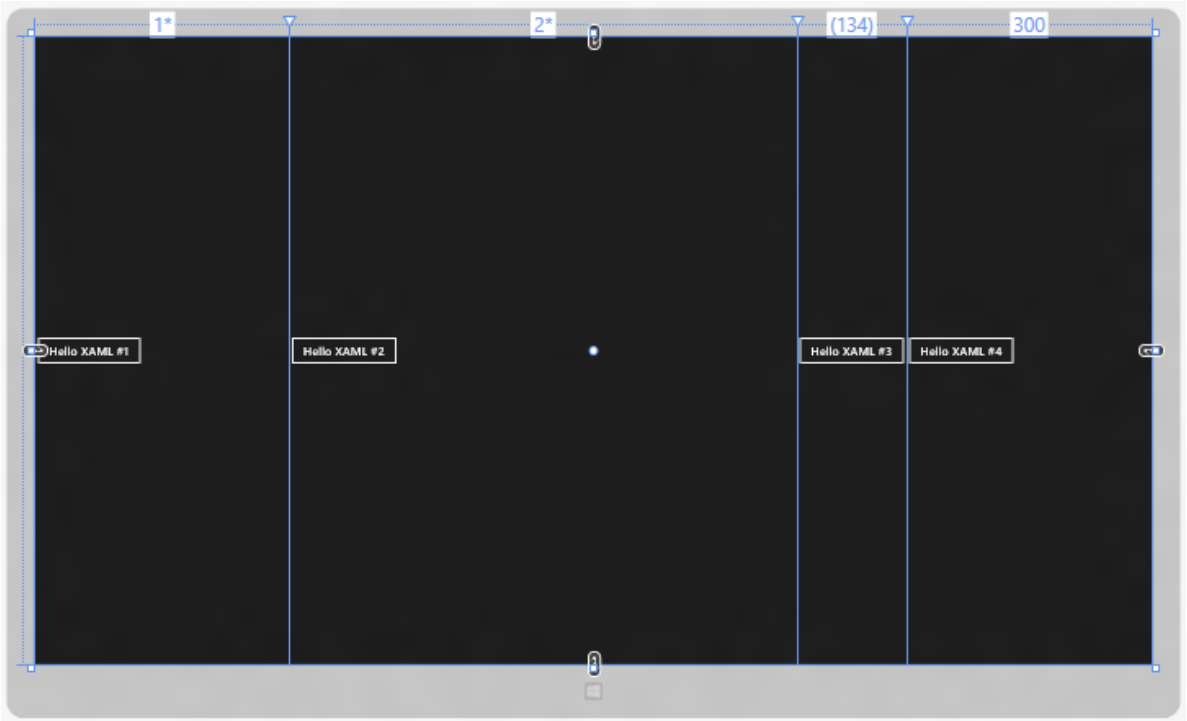
A Visual Studióban a Blank Application (XAML) sablon alapértelmezetten egy **Grid** vezérlőt helyezett az oldalba, így azt már nem kell külön definiálnunk. A sorok és oszlopok által meghatározott cellákba további vezérlőket helyezhetünk el, és definiálhatjuk az egyes cellák közötti átnyúlásokat is. Az elemek elhelyezése előtt definiálnunk kell a **Grid** sorait és oszlopait.

A következő példa bemutatja a **Grid** használatának alapjait:

```
<Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">

    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="*" />
        <ColumnDefinition Width="2*" />
        <ColumnDefinition Width="Auto" />
        <ColumnDefinition Width="300" />
    </Grid.ColumnDefinitions>

    <Button Grid.Column="0" Content="Hello XAML #1" />
    <Button Grid.Column="1" Content="Hello XAML #2" />
    <Button Grid.Column="2" Content="Hello XAML #3" />
    <Button Grid.Column="3" Content="Hello XAML #4" />

</Grid>
```

A **Grid** oszlopait a **ColumnDefinitions** tulajdonságán keresztül definiáljuk. Ez olyan gyűjtemény, ami **ColumnDefinition** típusú objektumokat tartalmaz. A példában a méretezés mindegyik formája jól látható. Az első két „csillagos szélességű” oszlop 1:2 arányban fogja megosztani a rendelkezésükre álló területet, a

harmadik oszlop pontosan olyan széles lesz, mint amilyen szélességet a benne lévő elem (vagy elemek) megkövetelnek, míg a negyedik oszlop fix 300 pixel szélességű lesz. A méretezés eredményét az 5-7 ábra mutatja.



5-7 ábra: Grid oszlopok méretezése

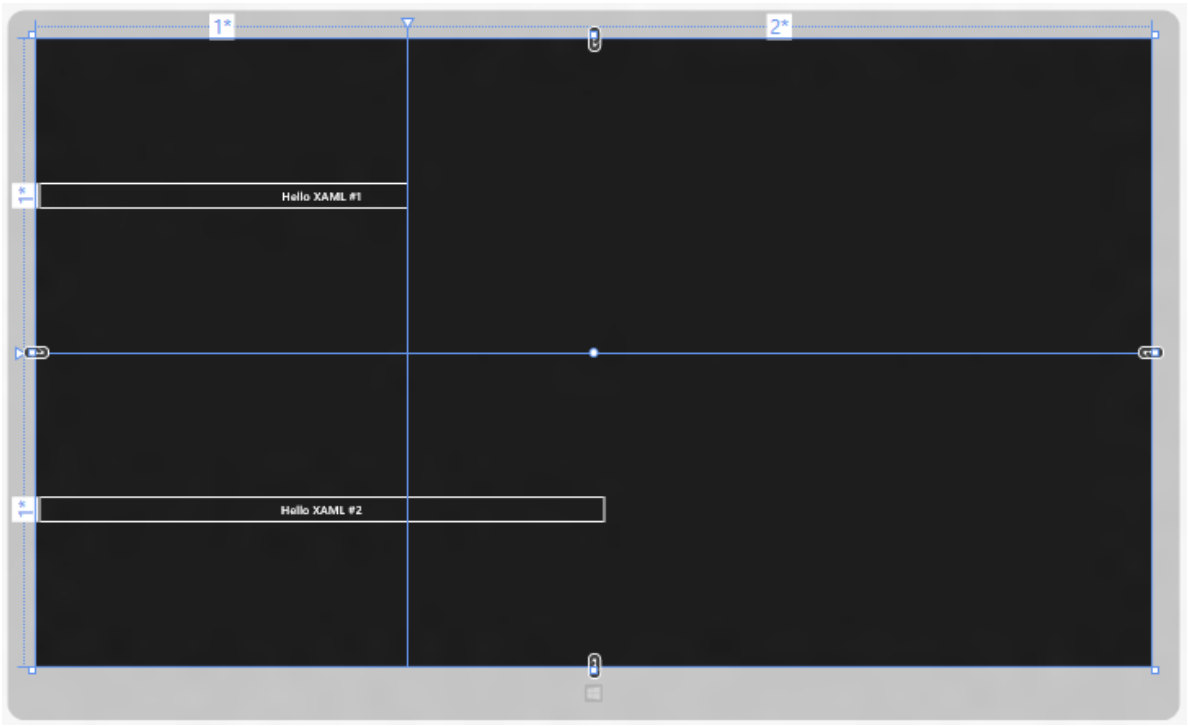
A **Grid** sorait hasonló módon tudjuk megadni, de ekkor a **RowDefinitions** tulajdonságba kell írnunk **RowDefinition** elemeket, melyeknek nem a szélessége, hanem a magassága lesz beállítható.

A **Grid** lehetővé teszi, hogy egy elem túlnyúljon a celláján. Ennek szemléltetéséhez alakítsuk át egy kicsit az előbbi példát úgy, hogy csak az első két oszlopot tartsuk meg, és hozzunk létre két egyenlő magasságú sort, melyekben egy-egy nagyon széles – az első oszlopon túlnyúló – gombot helyezünk el:

```
<Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">

    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="*" />
        <ColumnDefinition Width="2*" />
    </Grid.ColumnDefinitions>

    <Grid.RowDefinitions>
        <RowDefinition />
        <RowDefinition />
    </Grid.RowDefinitions>

    <Button Grid.Row="0" Width="700" Content="Hello XAML #1" />
    <Button Grid.Row="1" Grid.ColumnSpan="2" Width="700"  Content="Hello XAML #2" />

</Grid>
```

Az oszlop- és sordefiníciók alapértelmezett szélessége és magassága egy csillag, így jelen példában a két üres sordefiníció valójában két „egy csillag” magasságú sor, amelyek egyenlő mértékben osztják meg a rendelkezésre álló magasságot.

A gomboknak az előző példával ellentétben nem a **Grid.Column**, hanem a **Grid.Row** tulajdonságát kell beállítanunk, és érdemes figyelmet szentelni a második gomb **Grid.ColumnSpan** tulajdonságának, ugyanis annak a beállításával tudjuk az átnyúlást szabályozni. Az 5-8 ábra mutatja a végeredményt.



5-8 ábra: Grid elemeinek túlnyúlása másik cellába

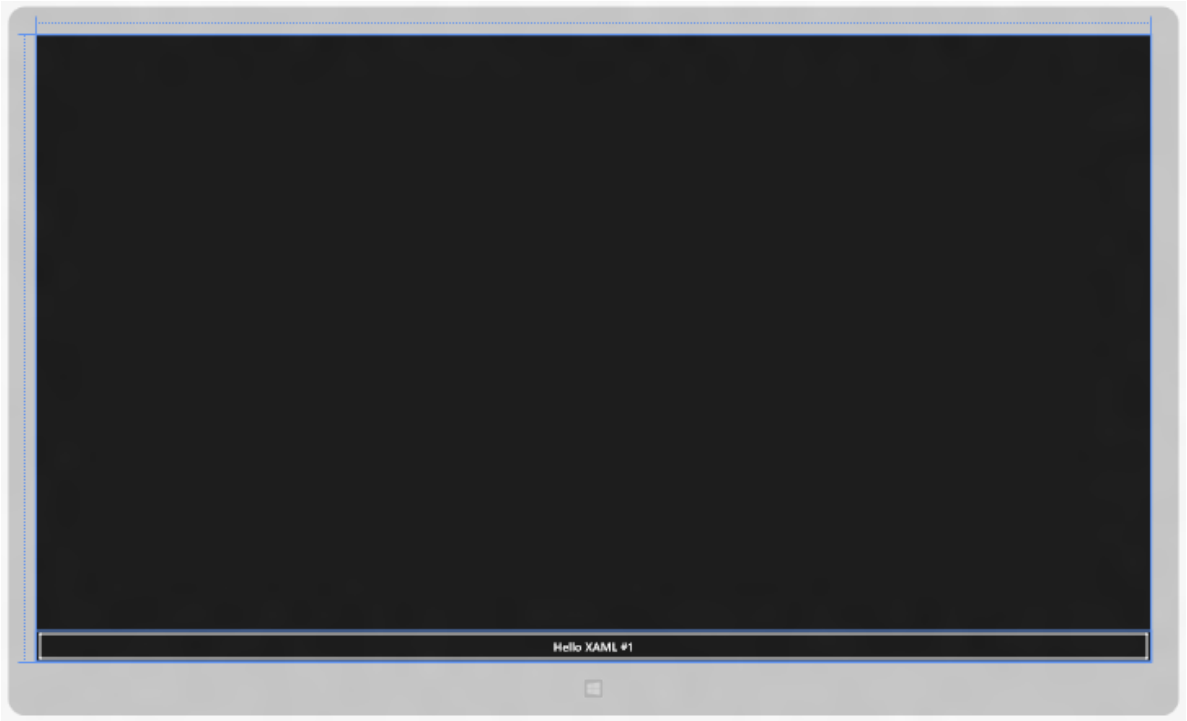
Jól látható, hogy az első gomb (aminek nem állítottunk be semmi extra tulajdonságot) le van vágva a cellahatáron, a második gomb azonban átnyúlik a második oszlopba.

Alignment

A vezérlők a rendelkezésükre álló területen belül több módon is elrendezhetők. Felmerülhet az igény, hogy egy elem ne balra, hanem középre vagy jobbra legyen igazítva; ne a lap közepére, hanem inkább a tetejére vagy az aljára helyezzük. Az az elvárás is megfogalmazódhat, hogy a vezérlő töltse ki a rendelkezésre álló területet. Mindezen elvárásoknak az elemek **`HorizontalAlignment`** és **`VerticalAlignment`** tulajdonságainak állításával tudunk megfelelni. A következő kis kódrészlet ezeknek a tulajdonságoknak a használatát mutatja be, az eredményt pedig az 5-9 ábrán láthatjuk:

```
<Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
    <Button
        HorizontalAlignment="Stretch"
        VerticalAlignment="Bottom"
        Content="Hello XAML #1" />
</Grid>
```

A gomb az elvárásoknak megfelelően az oldal alján helyezkedik el úgy, hogy kitölti az oldal teljes szélességét.

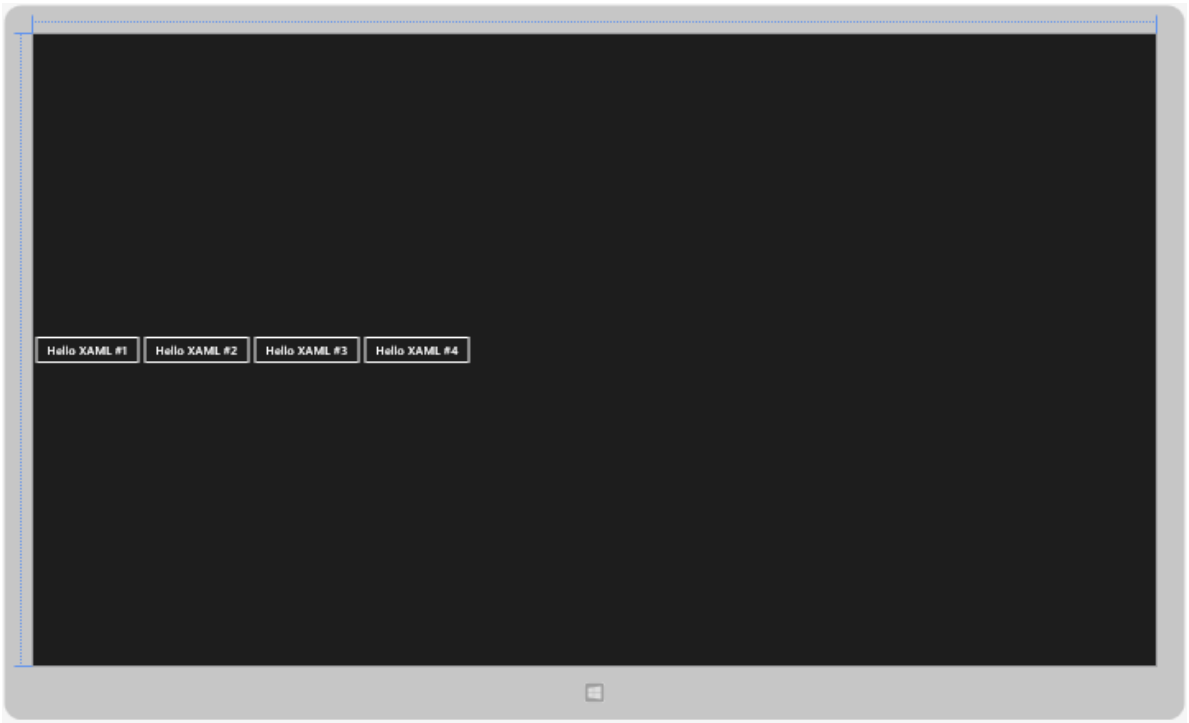


5-9 ábra: Elemek vízszintes és függőleges elrendezése

StackPanel

A következő fontos panel a **`StackPanel`**. Ez a vezérlő függőlegesen (ez az alapértelmezett) vagy vízszintesen egymás után helyezi a beágyazott elemeket. Az elrendezés irányát az **`Orientation`** tulajdonságon keresztül lehet beállítani. A következő kódrészlet és az 5-10 ábra ennek a panelnek a használatát szemléltetik.

```
<Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
    <StackPanel Orientation="Horizontal">
        <Button Content="Hello XAML #1" />
        <Button Content="Hello XAML #2" />
        <Button Content="Hello XAML #3" />
        <Button Content="Hello XAML #4" />
    </StackPanel>
</Grid>
```

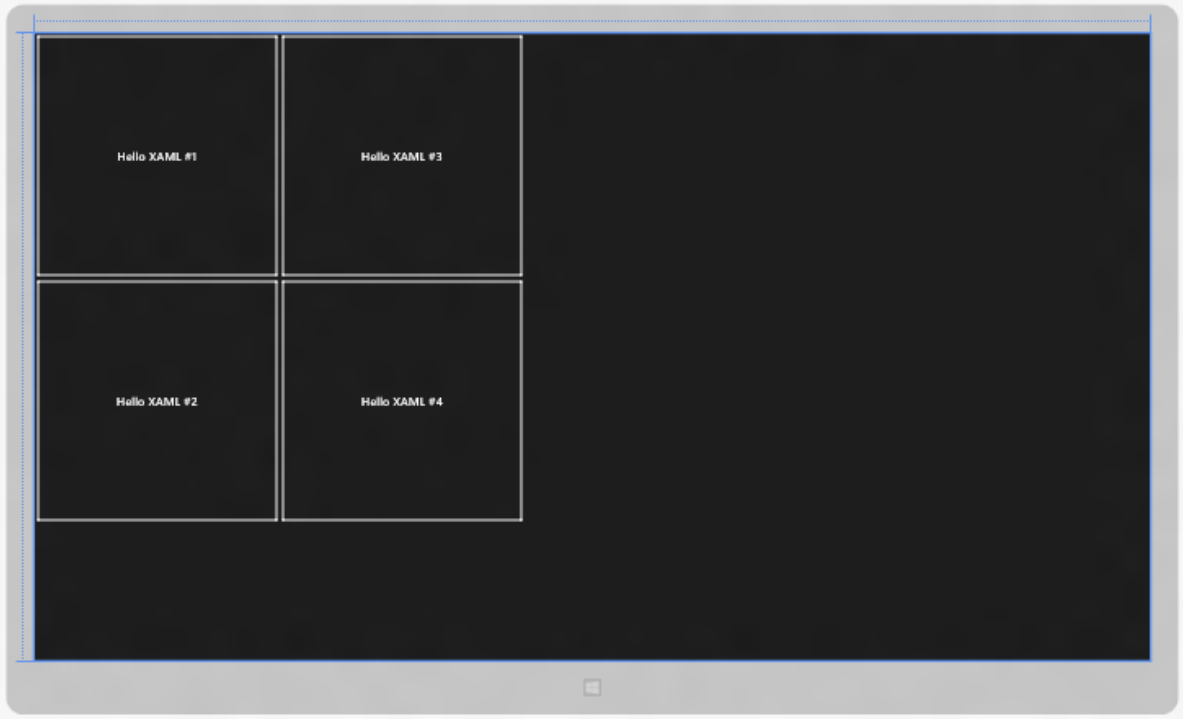


5-10 ábra: StackPanel működése

VariableSizedWrapGrid

Jogosan merülhet fel a kérdés, hogy mi van, ha annyi elem kerül a **`StackPanel`**-be, hogy kilóg az oldalról. És a válasz nem más, mint hogy ki fog lógni az oldalról. Azon probléma megoldásában, hogy a rendelkezésre álló szélesség vagy magasság elérése után egy új sorba vagy oszlopba kezdje el rakosgatni az elemeket a rendszer, a **`VariableSizedWrapGrid`** fog segítséget nyújtani. A következő kódrészlet és az 5-11 ábra ennek a panelnek a használatát mutatja be:

```
<Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
    <VariableSizedWrapGrid>
        <Button Width="300" Height="300" Content="Hello XAML #1" />
        <Button Width="300" Height="300" Content="Hello XAML #2" />
        <Button Width="300" Height="300" Content="Hello XAML #3" />
        <Button Width="300" Height="300" Content="Hello XAML #4" />
    </VariableSizedWrapGrid>
</Grid>
```



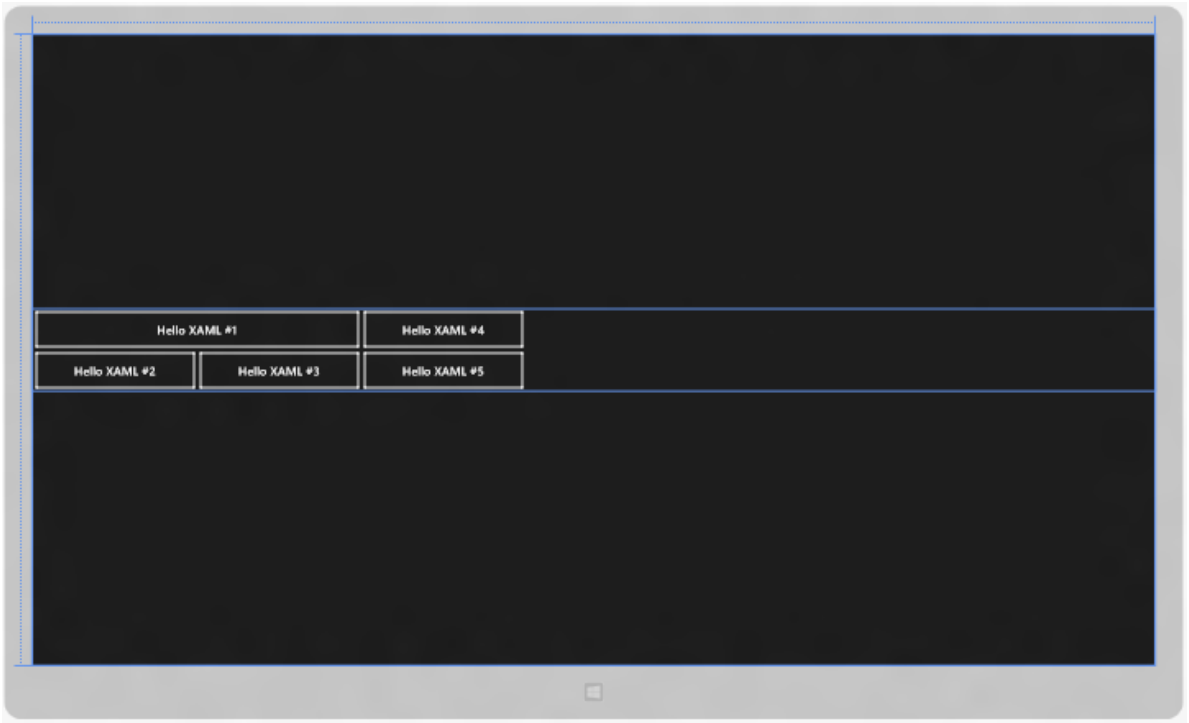
5-11 ábra: VariableSizedWrapGrid működés közben

A beágyazott vezérlőket alapértelmezett beállításai mellett fentről lefelé, majd balról jobbra fogja elhelyezni, de ezt könnyedén módosíthatjuk az **`Orientation`** tulajdonságának állításával. Ez a vezérlő azért van, hogy egyenlő méretű elemeket ilyen rácsos elhelyezésben, de mégis dinamikusan rendezzen el a felületen (sorok és oszlopok számának tekintetében). Amennyiben egy elemnek több sorra vagy oszlopra van szüksége, akkor azt az elem **`VariableSizedWrapGrid.ColumnSpan`** vagy **`VariableSizedWrapGrid.RowSpan`** tulajdonságának állításával tudjuk jelezni. Érdemes figyelembe venni, hogy a vezérlő az oszlopok és sorok méretét az első elem méretéből határozza meg. Így ha rögtön az első elem szélesebb vagy magasabb, mint a „normál” elemek, akkor a megjelenítés során nagyobbak lesznek a rácspontok a kelleténél.

A panel extra beállításait a következő kódrészlet és az 5-12 ábra szemléltetik.

```
<Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
    <VariableSizedWrapGrid ItemWidth="200" ItemHeight="50" Height="100">

        <Button Content="Hello XAML #1"
                HorizontalAlignment="Stretch"
                VerticalAlignment="Stretch"
                VariableSizedWrapGrid.ColumnSpan="2" />

        <Button Content="Hello XAML #2"
                HorizontalAlignment="Stretch"
                VerticalAlignment="Stretch" />

        <Button Content="Hello XAML #3"
                HorizontalAlignment="Stretch"
                VerticalAlignment="Stretch" />

        <Button Content="Hello XAML #4"
                HorizontalAlignment="Stretch"
                VerticalAlignment="Stretch" />
```

```
        <Button Content="Hello XAML #5"
                HorizontalAlignment="Stretch"
                VerticalAlignment="Stretch" />

    </VariableSizedWrapGrid>
</Grid>
```



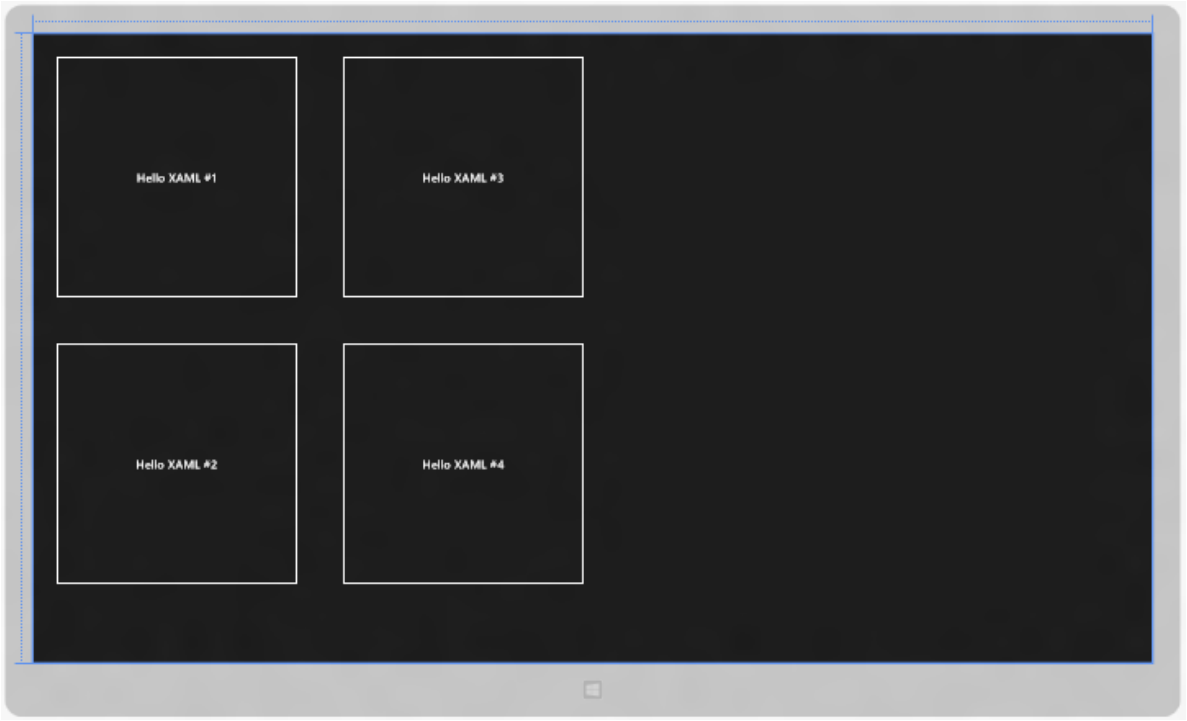
5-12 ábra: VariableSizedWrapGrid testreszabottan

Margin

Az utóbbi két panel – és az előző vezérlő – kapcsán felvetődhet a kérdés, hogy miképpen lehet megoldani, hogy az elemek egymáshoz képest kicsit szellősebben helyezkedjenek el. Erre a vezérlők **Margin** tulajdonsága használható. A **Margin**nal azt lehet szabályozni, hogy az elemek mekkora távolságot tartsanak a körülöttük lévő többi elemtől.

A **Margin** használatát és működését a következő kódrészlet és az 5-13 ábra fogja szemléltetni.

```
<Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
    <VariableSizedWrapGrid>
        <Button Width="300" Height="300" Margin="25" Content="Hello XAML #1" />
        <Button Width="300" Height="300" Margin="25" Content="Hello XAML #2" />
        <Button Width="300" Height="300" Margin="25" Content="Hello XAML #3" />
        <Button Width="300" Height="300" Margin="25" Content="Hello XAML #4" />
    </VariableSizedWrapGrid>
</Grid>
```



5-13 ábra: A Margin tulajdonság használata

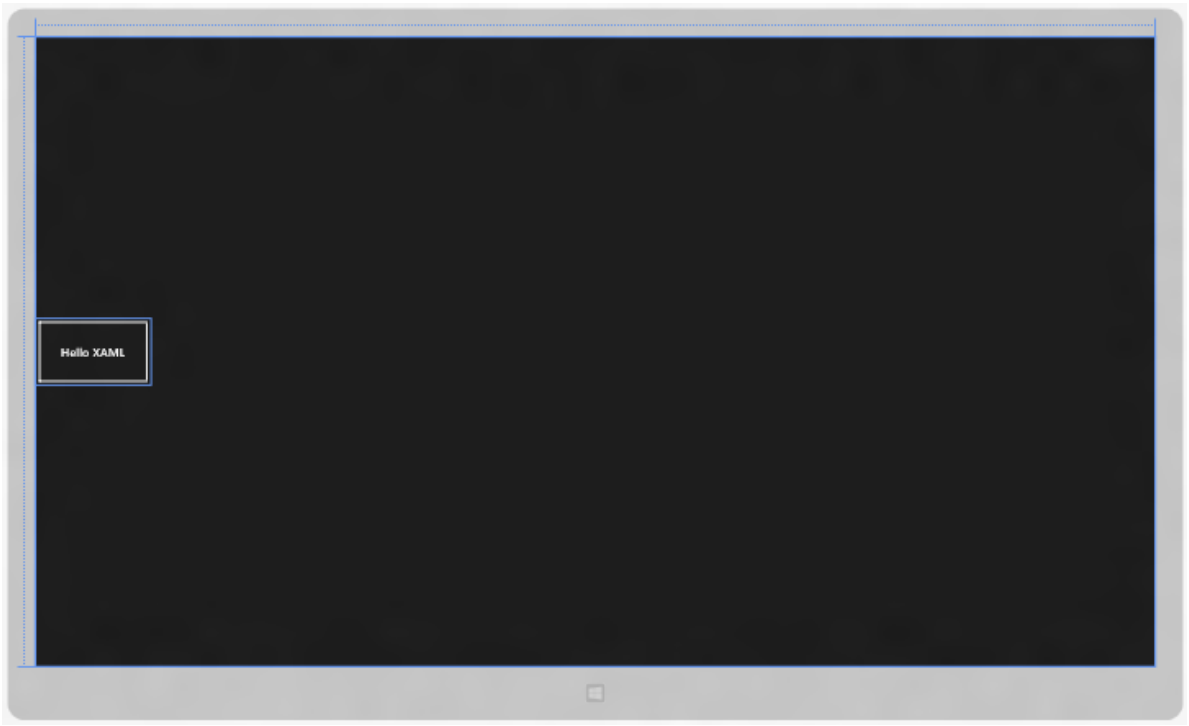
Az előbbi ábrán jól látható a **Margin** hatása. A gombok a képernyő szélétől 25, viszont egymástól összesen 50 egység távolságra helyezkednek el, mivel a **Margin** virtuálisan megnöveli a vezérlők méretét. Virtuálisan, mivel a vezérlő nem, csupán az igényelt hely mérete nő meg. Ezen az egy paraméteres változaton túl kettő és négy paraméterrel is meg lehet adni a margók méretét. Két paraméter megadása esetén az első szám az oldalsó, a második pedig az alsó és felső margó méretét fogja megadni. Négy paraméter megadása esetén pedig értelemszerűen egyenként állíthatjuk az összes oldal margóját bal,felső, jobb, alsó sorrendben. A következő kódsor a többparaméteres margóbeállítást mutatja:

```
<Button Content="Hello XAML #1" Margin="10, 20, 30, 40" />
```

Padding

A **Margin** hatásához hasonló a **Padding** tulajdonság, amit talán a „belső margó” kifejezéssel lehetne a legjobban körülírni. A **Padding** azt határozza meg, hogy egy vezérlő és tartalmának falai között mekkora távolság legyen. Ugyanezt a hatást el lehetne érni azzal is, ha a szóban forgó tartalomnak adnánk **Margin** értéket. Az esetek többségében ízlés kérdése, hogy ki melyiket használja inkább, vagy hogy melyikhez van hozzáférése (amennyiben a tartalom például dinamikusan generált, és annak **Margin** tulajdonságához ily módon nem férünk hozzá). A **Padding** tulajdonság használatát a következő kódrészlet és az 5-14 ábra mutatja be:

```
<Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
    <Button Padding="25" Content="Hello XAML" />
</Grid>
```



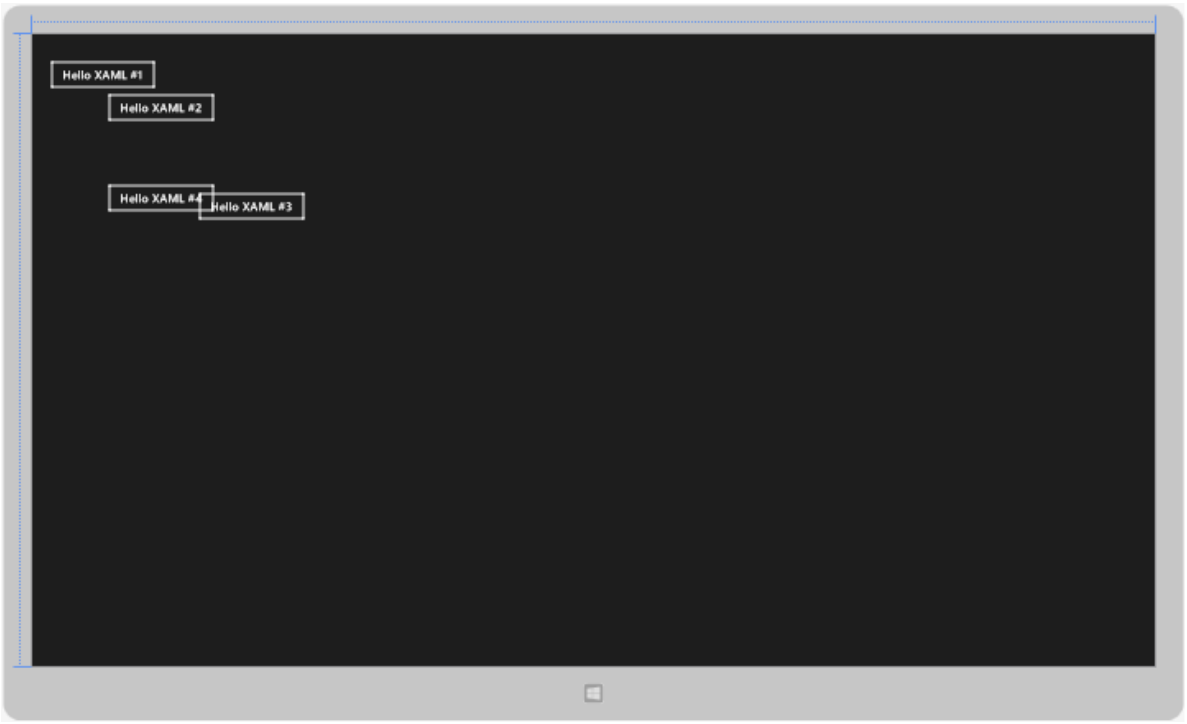
5-14 ábra: Padding használata

Canvas

Ejtsünk szót arról a panelről is, amelyik a legjobban hasonlít a WinForms-os ősre: ez a **Canvas**. Ebben a panelben a vezérlők fix koordináták alapján kerülnek elrendezésre, melyeket a vezérlőkön a **Canvas.Left** és **Canvas.Top** tulajdonságokkal lehet beállítani. A következő kis kódrészlet és az 5-15 ábra a **Canvas** használatát mutatja be:

```
<Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
    <Canvas>
        <Button Content="Hello XAML #1" Canvas.Left="20" Canvas.Top="30" />
        <Button Content="Hello XAML #2" Canvas.Left="90" Canvas.Top="70" />
        <Button Content="Hello XAML #3" Canvas.Left="200" Canvas.Top="190" />
        <Button Content="Hello XAML #4" Canvas.Left="90" Canvas.Top="180" />
    </Canvas>
</Grid>
```

Mivel a **Canvas** a beágyazott elemeket abszolút pozíciókra helyezi el, ezért a képernyő méretének változását ez az elhelyezkedés nem képes követni. Így, mivel erősen rombolja a felület flexibilitását, a **Canvas**-t csak akkor érdemes használni, ha más panel típusú vezérlő nem alkalmas a feladatra.



5-15 ábra: Canvas használata

Transzformációk

Az eddig bemutatott elrendezési módszerek azt szabályozzák, hogy a vezérlők a felülethez és egymáshoz viszonyítva hogyan helyezkedjenek el, és mekkora területet foglaljanak el. A transzformációk ezzel szemben úgy képesek befolyásolni a vezérlők vizuális megjelenését, hogy az nincs kihatással az elrendezési és helyfoglalási logikára. Például ha egy gomb megnyomásakor azt akarjuk, hogy a megnyomást egy vizuális visszajelzés is jelezze, és kicsit zsugorodjon össze – mintha tényleg beljebb lenne nyomva –, akkor azt nem jó ötlet a **`Width`** és **`Height`** tulajdonságok csökkentésével megoldani. Egyfelől, mert nem lehet százalékosan megadni az új kívánt méretet, másfelől pedig azért, mert ezeknek a tulajdonságoknak a változtatásával az elem összezsugorodna, és az esetlegesen körülötte lévő többi tartalom azonnal reagálna, és kitöltené a felszabadult helyet. Ezt az effektust a következő kódrészlet és az 5-16 ábra szemlélteti:

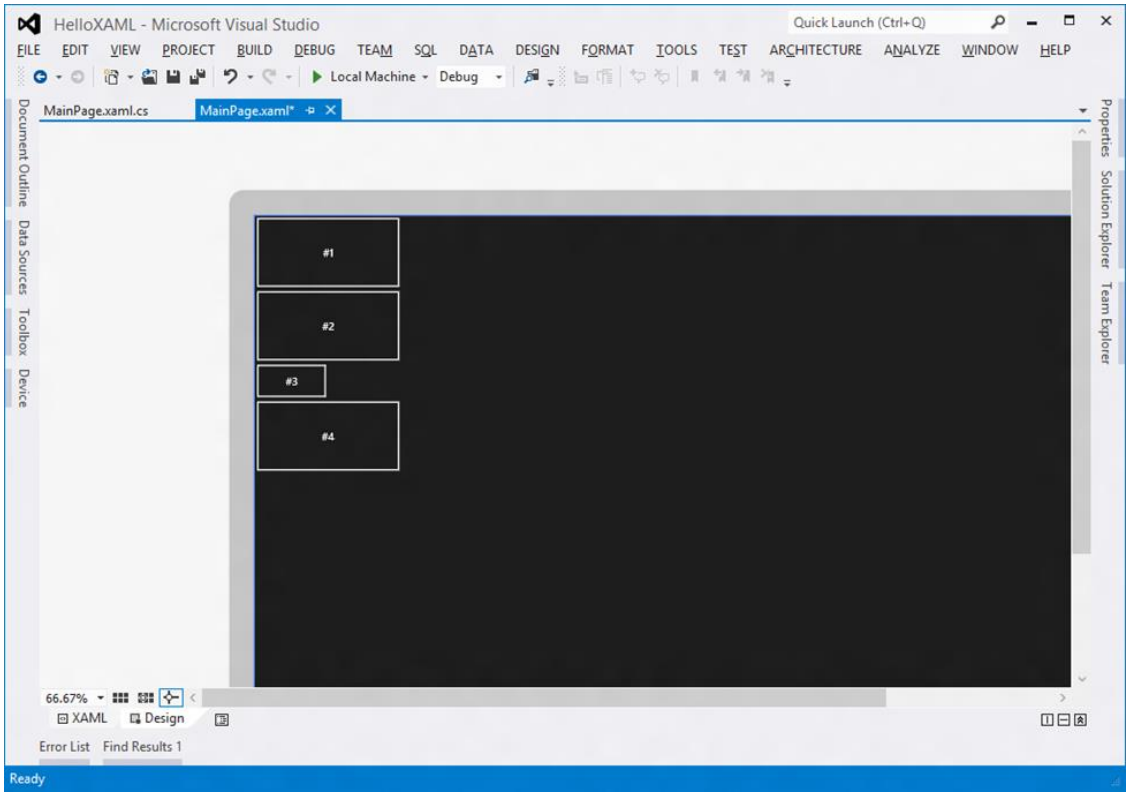
```
<Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
    <StackPanel>
        <Button Content="#1" Width="200" Height="100" />
        <Button Content="#2" Width="200" Height="100" />
        <Button Content="#3" Width="100" Height="50" />
        <Button Content="#4" Width="200" Height="100" />
    </StackPanel>
</Grid>
```

Erre a problémára a megoldást a transzformációk jelentik.

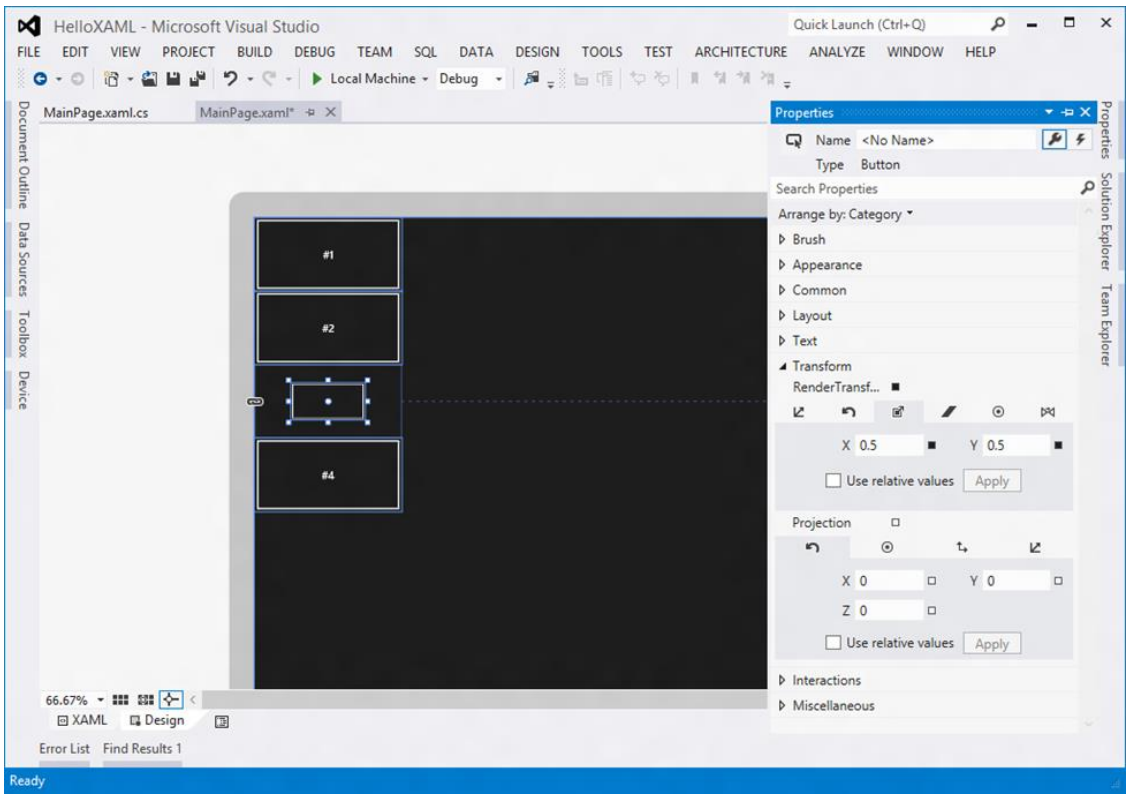
A következő kódrészlet és az 5-17 ábra szemlélteti a transzformáció használatát:

```
<Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
    <StackPanel>
        <Button Content="#1" Width="200" Height="100" />
        <Button Content="#2" Width="200" Height="100" />
        <Button Content="#3" Width="200" Height="100" RenderTransformOrigin="0.5,0.5" >
            <Button.RenderTransform>
                <ScaleTransform ScaleX="0.5" ScaleY="0.5"/>
            </Button.RenderTransform>
        </Button>
```

```
        <Button Content="#4" Width="200" Height="100" />
    </StackPanel>
</Grid>
```



5-16 ábra: Helytelen méretváltoztatás

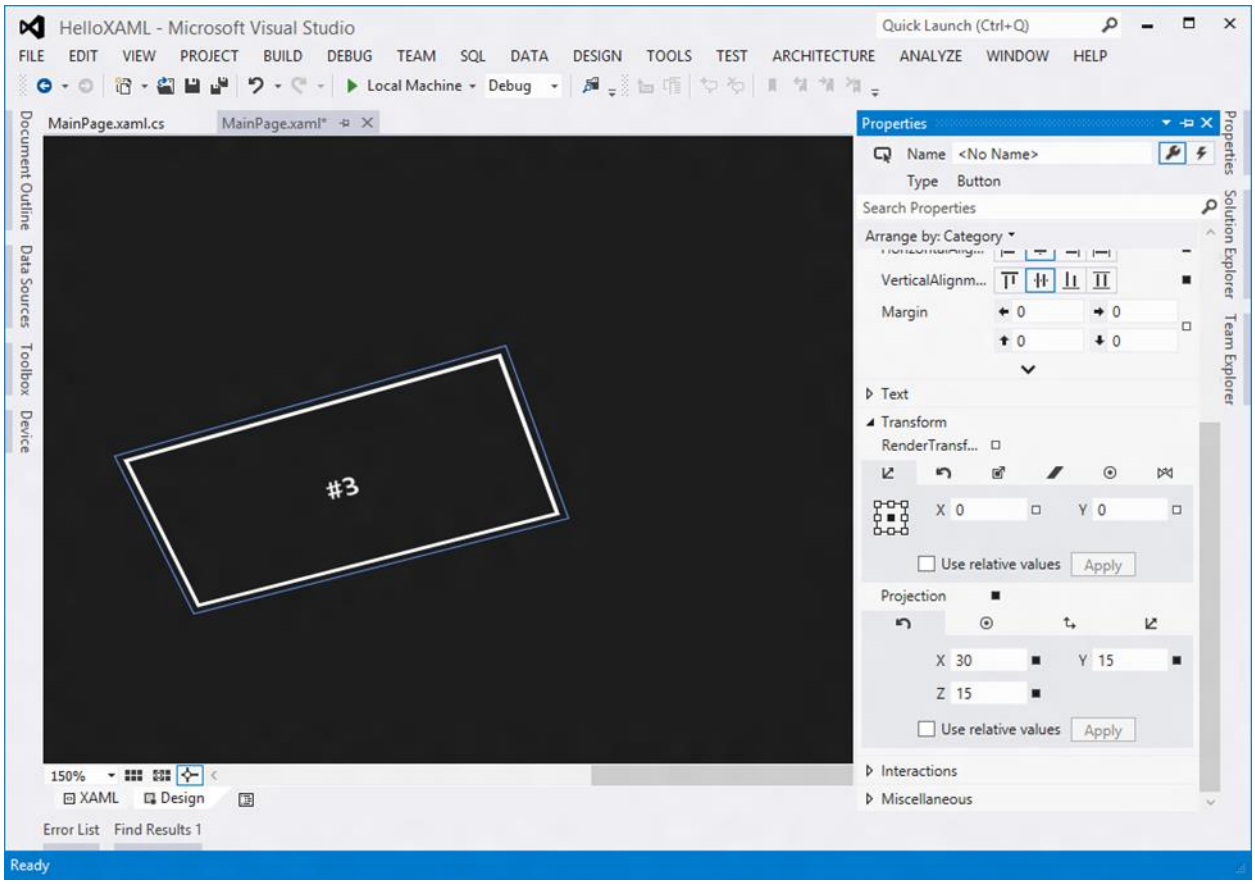


5-17 ábra: Méretváltoztatás transzformációval

A transzformációkat a vezérlők **RenderTransform** tulajdonságán keresztül lehet megadni, és az egyszerű átméretezésnél sokkal izgalmasabb dolgokat is lehet velük művelni. Lehetőség van eltolásra, forgatásra, nyírásra, skálázásra, és mindezeknek a transzformációknak meg lehet adni a középpontját is, például egy forgatás esetében azt a pontot, amely körül a vezérlőt el akarjuk forgatni. Ezeken túl még projekciókat – 3D hatású transzformációkat – is alkalmazhatunk, azaz a vezérlőket X, Y, és Z tengelyen is el lehet mozdítani és a tengelyek körül forgatni. Mindezt – fontos megjegyezni – úgy, hogy a vezérlők elrendezésére nem lesz hatással. Ez azt jelenti, hogy az elrendezés logikájáért felelős kódok úgy érzékelik, a vezérlő az eredeti helyén, az eredeti méretében van, és ennek megfelelően nem rendezik át a transzformált vezérlő körül lévő elemeket, azok például kicsinyítésnél nem csúsznak össze.

A projekció használatát és működését a következő kódrészlet, illetve az 5-18 ábra szemlélteti:

```
<Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
    <Button Content="#3" Width="200" Height="100"
            HorizontalAlignment="Center" VerticalAlignment="Center" >
        <Button.Projection>
            <PlaneProjection RotationX="30" RotationY="15"
                             RotationZ="15" GlobalOffsetZ="100"/>
        </Button.Projection>
    </Button>
</Grid>
```



5-18 ábra: Projekció

XAML + C#

Természetesen nem lehet mindent pusztán a XAML leírással megoldani, és bizonyos esetekben (minél jobban ismerjük a XAML-t, annál kevesebb esetben) szükség van arra, hogy a XAML felületünket C# kóddal támogassuk. Ilyen például az a helyzet, amikor egy eseményt kell kezelnünk vagy az, ha egy elem tulajdonságait kódból szeretnénk kiolvasni vagy állítani.

Egy vezérlő eseményére való feliratkozás ugyanúgy néz ki, mint bármelyik tulajdonság értékének beállítása:

```
<Button Content="Hello XAML #1" Click="ButtonClick" />
```

Ennek a kódnak az eredménye az lesz, hogy megjelenik egy eseménykezelő a felület mögöttes kódjában, és ez a következőképpen néz ki:

```
private void ButtonClick(object sender, RoutedEventArgs e)
{
    // ...
}
```

Az eseménykezelő **`sender`** paraméterét **`Button`** típusúvá konvertálva tudunk használható referenciát szerezni az eredeti (eseményt kiváltó) gombra, és ennek megfelelően az eseménykezelőből be tudjuk állítani a nyomógomb különböző tulajdonságait (például **`IsEnabled`**, **`Visibility`** stb.).

Ha arra van szükségünk, hogy a mögöttes kódfájl tetszőleges részéről elérjük a nyomógombot a neve alapján, akkor nevet kell adnunk a gombnak, amit az **`x:Name`** tulajdonságának állításával tudunk megtenni:

```
<Button x:Name="btn1" Content="Hello XAML #1" />
```

Ezek után a kód tetszőleges részéről el tudjuk érni a gombot a **`btn1`** nevű változón keresztül.

> *Mindent, amit XAML-ben meg tudunk tenni, azt kódban (akár C#, akár Visual Basic vagy C++ nyelv használatával) is meg tudjuk csinálni, ám ez fordítva már nem igaz.*

Összegzés

Ebben a fejezetben megtanulhattad a XAML nyelv alapvető szintaktikáját, és a vezérlők dinamikus elhelyezését a felületen – fix koordináták vagy méretek használata nélkül.

A 6. fejezetben egy kicsit mélyebben megismerheted a XAML-t és azt, hogy milyen lehetőségek adódnak a felületen elhelyezett vezérlők tulajdonságainak (így akár egy címke szövegének) manipulálására.

6. Windows 8 XAML ismeretek — mélyebben

Ebben a fejezetben az alábbi témákat ismerheted meg:

- Erőforrások, amelyek segítségével lehetőség nyílik központi helyen definiálni szövegeket, számokat, színeket és egyéb tetszőleges objektumokat, hogy azokat az egyes vezérlőkhöz rendelhessük
- Stílusok, amelyekre az egyes vezérlők hivatkozhatnak megjelenésük és viselkedésük egyszerű definiálásához
- Sablonok, melyekkel lehetőség nyílik a vezérlők kinézetének teljes testreszabására
- Animációk, melyekkel a vezérlők tulajdonságait lehet látványosan megváltoztatni
- Adatkötések, melyek lehetővé teszik, hogy a felületet és annak tulajdonságait programozás nélkül adatokhoz kapcsoljuk

Erőforrások

Mikor felhasználói felületeket készítünk, igen nagy valószínűséggel találkozunk olyan helyzetekkel, amikor bizonyos tulajdonságokat (nyomógomb vagy címke szövegek, szín, betűméret stb.) másolgatunk egyik elemről a másikra. Ha később valamilyen módosítás kapcsán csak egy színt szeretnénk megváltoztatni, akkor kénytelenek vagyunk előkeresni az adott szín összes előfordulását – a korábbi másolásnak köszönhetően –, és azokat egyenként átírni.

Nagyon hasonló problémák kezelésére C#-ban lehetőségünk van osztály szintű statikus változókat és konstansokat definiálni, amelyekre több helyről is hivatkozhatunk. Később, ha változtatnunk kell, elegendő csak egy helyen – az adott statikus változó vagy konstans értékadásánál –módosítani.

Hasonló lehetőséget nyújtanak a XAML-ben az *erőforrás*ok, amelyekkel aztán az oldal (vagy akár az egész alkalmazás) különböző részeiből elérhető adatokat definiálhatunk. Egy erőforrás bármi lehet! Egy szöveg, egy szín, egy szám, egy Boolean érték, akár egy komplex egyedi objektum is, melynek a különböző tulajdonságait is el tudjuk érni. Az erőforrások a teljes alkalmazás vizuális fájának bármely szintjén megadhatóak, így akár definiálhatjuk például egy kiemelés színét alkalmazás szinten, oldal szinten, vagy ha csak az oldal egy adott részében akarjuk felhasználni, akkor például egy **`Grid`** vagy **`StackPanel`** erőforrásai között is.

Az erőforrások definiálása az elemek **`Resources`** tulajdonságán keresztül történik. Ez egy **`ResourceDictionary`** típusú szótár, amiben a kulcs egy **`string`**, az érték pedig **`System.Object`** példány, vagyis az erőforrás gyakorlatilag bármilyen típusú lehet. Egy erőforrás definíciója a XAML-ben a következőképpen néz ki:

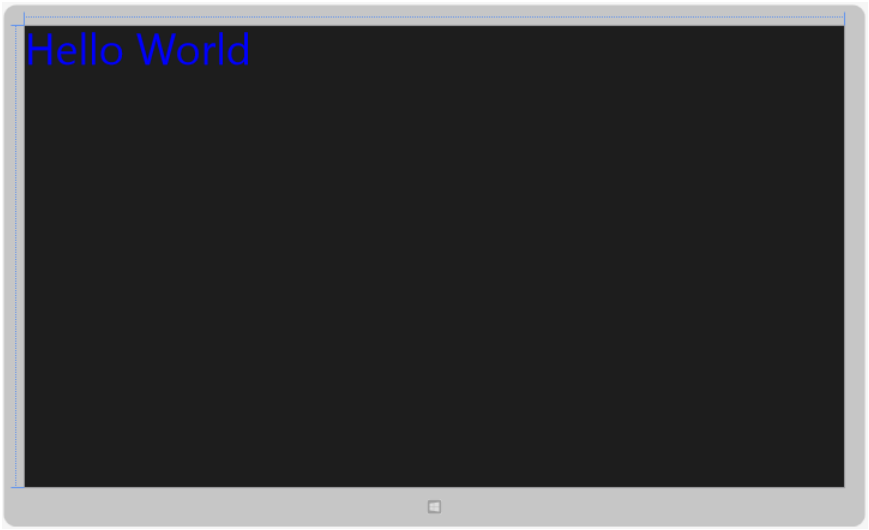
```
<Page.Resources>
    <SolidColorBrush Color="LightBlue" x:Key="BlueBrush" />
</Page.Resources>
```

Látható, hogy ez egy oldal szintű erőforrás, hiszen a **`Page`** objektumhoz kapcsolódik. Az erőforrás **`x:Key`** tulajdonsága az a kulcs, amelyre hivatkozva elérjük az adott objektumot az erőforrások között. Ennek értelemszerűen egyedinek kell lennie.

Egy erőforrásra az alábbi jelöléssel hivatkozhatunk:

```
<TextBlock Text="Hello World" FontSize="72" Foreground="{StaticResource BlueBrush}" />
```

A művelet eredményét a 6-1 ábra szemlélteti.



6-1 ábra: Színezés erőforrással

Az erőforrásra hivatkozó tulajdonság értékéül ezt a speciális szintaxist kell alkalmazni: kapcsos zárójelek között a **StaticResource** azonosítót adjuk meg, amit az erőforrás kulcsa követ.

Az erőforrás hivatkozások feloldása úgy történik, hogy a megjelenítő motor a hivatkozó elemtől a gyökér felé indul a vizuális fán, és átfésüli a szülőelemeket, hogy tartalmaznak-e olyan erőforrást, melynek a kulcsa a hivatkozásban szerepel. Ha a hivatkozás a szülőelemen nem található, akkor annak szülőjén folytatódik a keresés, és így tovább, egészen a vizuális fa gyökerének eléréséig. Ha különböző szinteken ugyanolyan kulccsal van definiálva két különböző erőforrás, akkor ennek alapján azt az erőforrást fogja megtalálni a rendszer, amelyik közelebb van a hivatkozáshoz. Így például ha definiálunk egy kiemelés színt alkalmazás szinten, és azt az alkalmazás különböző oldalain használjuk, amikor egy oldalon valami okból kifolyólag mégis más színre van szükség, akkor egyszerűen ott, az oldal szintjén felüldefiniáljuk a színt meghatározó erőforrást.

Sőt, mi több, lehetőség van arra is, hogy ezeket az erőforrásokat egy külön XAML fájlban írjuk össze, és aztán úgy, egy csomagként illesszük bele valamelyik erőforrás-gyűjteménybe! Már ebből sejthető, hogy mennyire egyszerű is lenne a különböző stíluselemeket kiemelni egy fájlba, annak több különböző verzióját elkészíteni, majd tetszés szerint azt a témát használni – egyetlen kódsor átírásával –, amelyikhez éppen kedvünk van. Vagy ha éppen ez az erőforrás-gyűjtemény különböző címke és nyomógomb szövegeket tartalmaz, akkor ilyen úton is kényelmesen hozzá lehet kezdeni egy többnyelvű alkalmazás elkészítéséhez.

Egy ilyen fájl készítéséhez a következő lépéseket kell megtennünk:

- Hozzá kell adni a projekthez egy Resource Dictionary típusú elemet a Project | Add New Item paranccsal.
- Ebbe a XAML fájlba helyezzük azokat az erőforrásokat, amelyeket egyébként közvetlenül az oldal erőforrásainál definiálnánk. Természetesen ügyelünk arra, hogy az x:Key tulajdonságok ne ütközzenek.
- Az alkalmazás megfelelő szintjén betöltjük ezt a külső fájlt. Később, amikor a felület megjelenését tesztelgetjük különböző beállításokkal, akkor csupán a betöltő részben kell egyetlen sort átírnunk ahhoz, hogy egy másik – a tesztelés céljára használt – fájl kerüljön betöltésre.

A betöltést végző XAML definíció a következőképpen néz ki:

```
<Page.Resources>
    <ResourceDictionary>
        <ResourceDictionary.MergedDictionaries>
            <ResourceDictionary Source="MyResources.xaml"/>
        </ResourceDictionary.MergedDictionaries>
        <!-- Ide lehet írni a helyben definiált erőforrásokat -->
    </ResourceDictionary>
</Page.Resources>
```

Amennyiben az erőforrások teljes egészében külső fájlban helyezkednek el, vagy legalábbis azon a szinten, ahol az erőforrásfájlra hivatkozunk, és nincs más helyben használt erőforrás, akkor ez a hosszas kifejtett szintaktika egyetlen sorra is lerövidíthető:

```
<Page.Resources>
    <ResourceDictionary Source="MyResources.xaml"/>
</Page.Resources>
```

Beépített erőforrások

Nincs szükség minden esetben arra, hogy saját erőforrásokat definiáljunk! A Windows rengeteg beépített erőforrást és témát tartalmaz (színek, betűméretek, igazítások, helykihagyások és számtalan egyéb formázás), amelyek dizájn szempontjából is átgondoltak, esztétikusak. Ezek használatával alkalmazásunk könnyedén használhatja például a rendszer színeit, de akár bonyolultabb stílusokat és sablonokat is.

A beépített témák között sötét, világos és magas kontrasztúak egyaránt vannak. A részletekért, értékekért érdemes belepillantani a programok telepítési könyvtára (alapértelmezésben **`C:\Program Files (x86)`**) alatt a **`Windows Kits\8.0\Include\winrt\xaml\design`** mappa **`generic.xaml`** fájljába, amely az imént említett hatalmas mennyiségű erőforrást és a vezérlők alapértelmezett megjelenését tartalmazza a három különböző témához.

Ha odafigyelünk arra, hogy az alkalmazáson belül ne "beégetett" stílusjegyeket használjunk (főleg színek esetében), akkor könnyedén illeszkedhetünk a három alaptémához – vagy akár egy teljesen egyedi, valamelyik alapra épülőhöz – anélkül, hogy a felülethez hozzá kellene nyúlni.

> *A beépített vezérlősablonok is ezekre a témákra épülnek. Azt, hogy a sötét és a világos téma közül melyiket használjuk, még ugyan az alkalmazás döntheti el, de a magas kontrasztú megjelenést már a felhasználó fogja kikényszeríteni a számítógép beállításainál! Ha a vezérlők alkalmazkodnak ahhoz – márpedig alkalmazkodnak –, a saját tartalmunk meg nem, az eredmény csúnya lesz.*

Azt, hogy az alkalmazás sötét vagy világos témát használjon, az **`App.xaml`** fájlban lehet beállítani az Application gyökérelem **`RequestedTheme`** attribútumának **`Dark`** vagy **`Light`** értékre állításával:

```
<Application
    x:Class="App19.App"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:App19"
    xmlns:localData="using:App19.Data"
    RequestedTheme="Light">
    <!-- Application.Resources -->
</Application>
```

Stílusok

Nézzünk meg egy olyan példát, amikor minden erőforrást központilag definiálunk, és azokra hivatkozunk egy vezérlő tulajdonságainak megadásánál!

```
<TextBlock Text="Hello World"
           HorizontalAlignment="{StaticResource MyHorizontalAlignment}"
           VerticalAlignment="{StaticResource MyVerticalAlignment}"
           Margin="{StaticResource MyMargin}"
           TextWrapping="{StaticResource MyTextWrapping}"
           Foreground="{StaticResource MyFontColor}"
           FontStyle="{StaticResource MyFontStyle}" />
```

Ha sok ilyen vezérlőnk van, akkor komoly figyelmet kíván, hogy annak minden tulajdonságára a helyes erőforrást használjuk. Ez nemcsak rengeteg gépelési munkát jelenthet, de egyúttal hibázási lehetőséget is. Azért, hogy ezt a feladatot igazán kényelmesen és hatékonyan lehessen elvégezni, *stílus*okat használhatunk. A stílusok nemcsak vizuális tulajdonságokat határozhatnak meg, hanem egy vezérlő tetszőleges tulajdonságát be lehet vele állítani.

A stílusokra úgy lehet tekinteni, mint egy célzott erőforráscsomagra. Célzott, ugyanis meg kell adni, hogy milyen típusú elemekre akarjuk az adott stílust ráhúzni, majd az adott típus tulajdonságainak tudunk értékeket adni. Egy egyszerű kódrészlet többet mond ezer szónál, lássuk, hogyan írhatjuk le az előző XAML részlet beállításaihoz tartozó stílust:

```
<Page.Resources>
    <Style x:Key="MyTextBlockStyle" TargetType="TextBlock">
        <Setter Property="HorizontalAlignment" Value="Center" />
        <Setter Property="VerticalAlignment" Value="Top" />
        <Setter Property="Margin" Value="25" />
        <Setter Property="TextWrapping" Value="Wrap" />
        <Setter Property="Foreground" Value="LightBlue" />
        <Setter Property="FontStyle" Value="Italic" />
    </Style>
</Page.Resources>
```

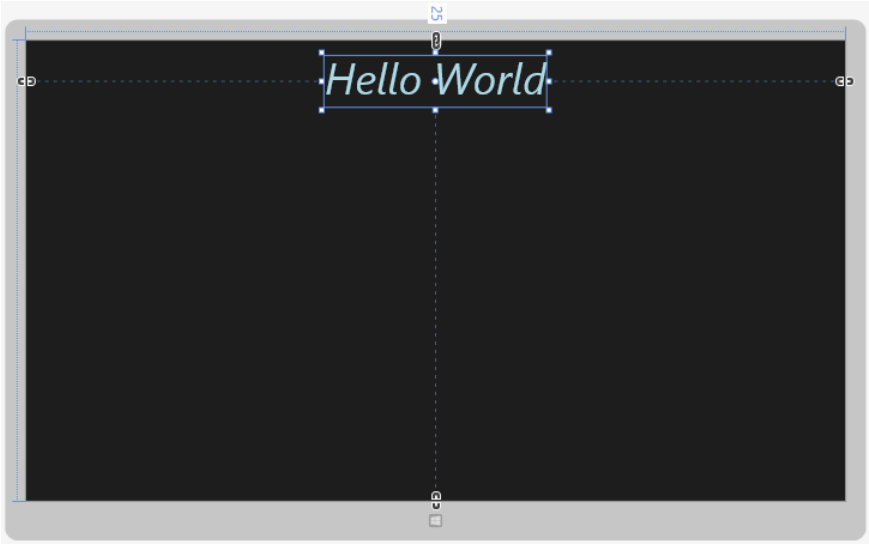
A stílus is egy erőforrás, ennek megfelelően adni kell neki egy kulcsot, és a **TargetType** tulajdonsággal definiálni kell, hogy a stílus milyen típusú elemekre vonatkozik. Ennek a típusnak nem feltétlenül kell konkrét vezérlőnek lennie. A **TargetType** lehet például egy **Panel** is, és így ez a stílus az összes **Panel**ből leszármaztatott vezérlőre (**Grid**, **StackPanel**, **Canvas** stb.) alkalmazható lesz. A **<Style>** tag belsejében tetszőleges számú Setter elemet definiálhatunk, melyek azt írják le, hogy egy adott tulajdonságnak milyen értéket akarunk adni.

A stílus felhasználásával pedig már lényegesen egyszerűbben és tömörebben állíthatjuk be a vezérlő tulajdonságait, mint a korábbi „szószátyár” példában:

```
<TextBlock Text="Hello World" Style="{StaticResource MyTextBlockStyle}" />
```

A művelet eredménye a 6-2 ábrán látható.

A **Style** tulajdonsághoz egyszerűen hozzárendeljük a stílust leíró erőforrást, és ezzel a vezérlő összes tulajdonsága megfelelően lesz beállítva. Amennyiben egy újabb tulajdonságnak akarunk a stíluson keresztül értéket adni, azt elegendő a stílus definíciójában módosítani, a vezérlőhöz nem kell hozzányúlnunk.



6-2 ábra: Stílus használata

Mivel ez is csak egy erőforrás, így ha különböző szinteken azonos kulccsal definiálunk más-más stílust, akkor a „legközelebbi" stílust fogja megkapni a vezérlő. Ez azonban nem feltétlenül fedi elvárásainkat, mert lehet, hogy egy meglévő bonyolult stílust szeretnénk felhasználni, csak épp néhány tulajdonság módosításával vagy hozzáfűzésével! Szerencsére a stílusok között öröklődést lehet definiálni, és annak segítségével egyszerűen tudjuk ezt a feladatot megoldani. Az öröklődést a stílus **BasedOn** tulajdonságának beállításával tudjuk jelölni, amiben egy másik stílusra (erőforrásra) kell hivatkoznunk. Például ha feltételezzük, hogy a korábban bemutatott stílus egy oldalcím kiemelésére létrehozott stílus, melyet csak az egyedi színnel egészítettünk ki, akkor így nézne ki a stílus definíciója:

```
<Page.Resources>
    <Style x:Key="MyHeaderStyle" TargetType="TextBlock">
        <Setter Property="HorizontalAlignment" Value="Center" />
        <Setter Property="VerticalAlignment" Value="Top" />
        <Setter Property="Margin" Value="25" />
        <Setter Property="TextWrapping" Value="Wrap" />
        <Setter Property="FontStyle" Value="Italic" />
    </Style>
    <Style x:Key="MyTextBlockStyle" TargetType="TextBlock"
           BasedOn="{StaticResource MyHeaderStyle}">
        <Setter Property="Foreground" Value="LightBlue" />
    </Style>
</Page.Resources>
```

A vezérlőben nincs szükség a stílus hivatkozásának megváltoztatására, hiszen csupán a stílusaink struktúráját alakítottuk át.

Lehetőség van *implicit stílus*ok definiálására is. Ezek olyan stílusok, amelyeknek nem adunk meg kulcsot – csak **TargetType**-ot. A **TargetType** tulajdonságban meghatározott vezérlők automatikusan, mindenféle explicit jelölés nélkül megkapják az adott stílust. Az implicit stílust a vezérlők alapértelmezett stílusának leírásához érdemes használni. Ezt a stílust központi helyen lehet definiálni, a stílusra nem kell hivatkozni, tisztább maradhat a kód. A háttérben a rendszer az adott típusú elemek **Style** tulajdonságában alapértelmezett módon az implicit stílust hivatkozza meg – ha van ilyen. Mivel az implicit stílusnak nincs kulcsa, így sem hivatkozni nem lehet rá, sem másik stílust leszármaztatni belőle. Ha egy vezérlőnek mégis explicit módon másik stílust adunk meg, akkor az felülírja az implicitet. Az implicit stílusban megadott tulajdonságértékeket nem fogja megkapni a vezérlő még akkor sem, ha esetleg azokat az explicit módon hivatkozott stílus nem írná felül.

Sablonok

A stílusok segítségével lehetőségünk van a vezérlők tulajdonságainak beállítását központosítani, ám nem tudjuk gyökeresen megváltoztatni a vezérlő megjelenését, illetve viselkedését. Nem tudunk egy alapértelmezetten téglalap alakú gombból kör alakú gombot készíteni! Az ilyen szintű módosításokra a sablonokat tudjuk használni. Ugyanúgy, ahogy minden vezérlőnek van **`Style`** tulajdonsága, úgy egy **`Template`** tulajdonsága is van, melyen keresztül teljesen újradefiniálhatjuk a vezérlő vizuális fáját.

> *Figyelem! Ezt nem szabad összekeverni azzal, hogy a vezérlők tetszőleges mélységben egymásba ágyazhatóak!*

Ahhoz, hogy a fent leírt példát megvalósítsuk (kör alakú gomb készítése), csupán néhány sor XAML kódot kell írni, amint azt ez a példa is mutatja:

```
<Button Content="Hello World">
    <Button.Template>
        <ControlTemplate>
            <Grid>
                <Ellipse Fill="DarkBlue"
                         Stroke="White"
                         StrokeThickness="2" />
                <ContentPresenter
                    Margin="25"
                    HorizontalAlignment="Center"
                    VerticalAlignment="Center" />
            </Grid>
        </ControlTemplate>
    </Button.Template>
</Button>
```

A kód eredményét a 6-3 ábra szemlélteti.

6-3 ábra: Egyedi sablon egy nyomógombon

A megoldás a vezérlő **`Template`** tulajdonságának kifejtésében lakozik. Az egy **`ControlTemplate`** típusú objektumot tartalmaz, ez írja le a vezérlőhöz tartozó konkrét vizuális fát. Mivel a **`ControlTemplate`** egyetlen elemet tartalmazhat csak, érdemes valamelyik panelt használni, hogy abba több elemet is beágyazhassunk.

A példában ez a panel egy **`Grid`**, mely egy **`Ellipse`** elemet és egy **`ContentPresenter`** típusú elemet tartalmaz. A **`ContentPresenter`** jelöli a vizuális fában azt a helyet, ahova a nyomógomb **`Content`** tulajdonságában meghatározott tartalom lesz irányítva a megjelenéskor. Jelen esetben ez csupán egy szöveg (**`TextBlock`**), de természetesen itt sokkal komplexebb vizuális fa is megjelenhet.

Ha a Visual Studióba bemásoljuk a kis példakódot, majd futtatjuk azt, észrevehetjük, hogy a gomb elvesztette minden állapotát. A kinézetét a sötétkék ellipszis jelenti, amely – eltérően a gomb alapvető működésétől – nem változik meg, ha fölé visszük az egeret, ha megnyomjuk az egér gombját, ha a gomb letiltott állapotba kerül, megkapja a fókuszt vagy éppen elveszíti azt. Mindig ugyanúgy néz ki! Ezt a problémát a vizuális állapotok kezelésével lehet megoldani – a fejezet későbbi részében erről is szó lesz.

Emellett van egy másik sablontípus is, ami különböző adatstruktúrák megjelenítésére alkalmas, de erről a fejezet későbbi részében lesz szó.

Animációk

Az animációk sokkal fontosabb szerepet játszanak egy intuitív felhasználói felület létrehozásában, mint azt a legtöbben egyáltalán csak sejtenénk. Egy kellemesen animált felület igényességet, a mindig simán, finoman mozgó „folyékony" felület pedig magas minőséget sugall.

Az animációkat gyakran éri az a vád, hogy feleslegesek, pazarolják az erőforrásokat, csak az időt húzzák és a produktív munkát gátolják, stb. Egy villogó és görgetéskor ugráló felület követése viszonylag sok figyelmet igényelhet, mert görgetésnél meg kell keresnünk, hogy hol hagytuk abba az olvasást, és ez igen frusztráló lehet. Ezzel ellentétben egy animált görgetés esetében az animációnak köszönhetően már lényegesen egyszerűbb nyomon követni, hogy mennyit is gördült a tartalom. Hasonló dolgot tapasztalhatunk különböző felületek közti navigációnál, amikor egyszer csak elénk ugrik egy ablak, és nem tudjuk, hogy honnan került oda.

Ablakkezelő felületeknél (mint például a Windows) az ablakok megnyitása és elrejtése (tálcára való lerakása) egy kis animációval történik, hogy lássuk, hova tűnt az ablak, és aztán amikor legközelebb meg akarjuk nyitni, már nem feltétlenül kell sem ikonokat nézegetni, sem feliratokat, hanem a vizuális memóriánkból pillanatok alatt előjön, hogy a keresett program a tálca melyik részére került a lekicsinyítése után. Ezek az apró figyelmességek nagyon hasznosak lehetnek a produktivitás szempontjából is.

Az animációk alapvető felépítésüket tekintve párhuzamba hozhatóak a stílusokkal. Az animációkat is az erőforrások között tudjuk tárolni, és definiálhatjuk, hogy melyik vezérlő melyik tulajdonságát hogyan változtassák meg. A megváltoztatandó tulajdonság – hasonlóan a stílusokhoz – nem feltétlenül kell, hogy közvetlenül vizuális állapotot jelöljön, adott esetben – bár furán hangzik – lehetőség van akár egy gomb szövegét is „animálni", megváltoztatni egy célértékre bizonyos idő elteltével.

Nézzünk meg egy példát egyszerű animáció definiálására!

```
<Page.Resources>
    <Storyboard x:Name="MyAnimation"
                Storyboard.TargetName="button">
        <DoubleAnimation
            Storyboard.TargetProperty="(Button.Width)"
            By="500" Duration="0:0:1"
            EnableDependentAnimation="True" />
    </Storyboard>
</Page.Resources>
```

Az animáció alapeleme a **Storyboard** objektum, ez ágyazza be az animációkat. A **TargetName** tulajdonság definiálja, hogy a **Storyboard** mely vezérlőhöz van rendelve. A beágyazott animációk ezen a vezérlőn fognak működni, de bármelyik konkrét animációt a **Storyboard**on belül saját **TargetName** tulajdonságával átirányíthatjuk más vezérlőre. Érdemes még észrevenni, hogy nem az **x:Key**, hanem az **x:Name** tulajdonsággal adtunk nevet a **Storyboard** objektumnak, vagyis ezzel a névvel azt el fogjuk érni a mögöttes kódfájlból.

A **Storyboard** animációkat tartalmaz, az aktuális példában egy **DoubleAnimation** típusút (lebegőpontos értékű tulajdonság animálására használható). Ezen az objektumon a **Storyboard.TargetProperty** tulajdonság állításával lehet megadni, hogy az animálni kívánt vezérlő mely tulajdonságát akarjuk változtatni: ez ebben az esetben a nyomógomb szélessége. Az animáció számos beállítási lehetőséget engedélyez. Több módon is megadhatjuk a kívánt végeredményt a **From**, **To**, és **By** tulajdonságok segítségével:

- Ha csak a **From** tulajdonságot állítjuk, akkor az animáció kezdetén az animált tulajdonság ebbe az állapotba ugrik, majd animálva visszaváltozik eredeti állapotába.
- Ha a **From** mellett használjuk a **By** tulajdonságot is, akkor a **From** érték **By** értékkel való változtatásáig fog eljutni az animáció. Például ha egy vezérlő szélességét animáljuk, és a **From** értéke 100, a **By** pedig 50, akkor a végeredmény 150 lesz.
- Ha a **From** mellé a **To** tulajdonságot is beállítjuk, akkor az elvárásoknak megfelelően a kiindulási értéktől a végértékig fogja animálni a tulajdonságot az animáció.
- Ha csak a **By** tulajdonságot használjuk, akkor a tulajdonság eredeti értékét fogja a változás mértékének megfelelően animálni.
- Ha a **To** tulajdonságot használjuk egymagában, akkor a tulajdonság eredeti értékétől ezen megadott végértékig fogja animálni a rendszer a tulajdonságot.

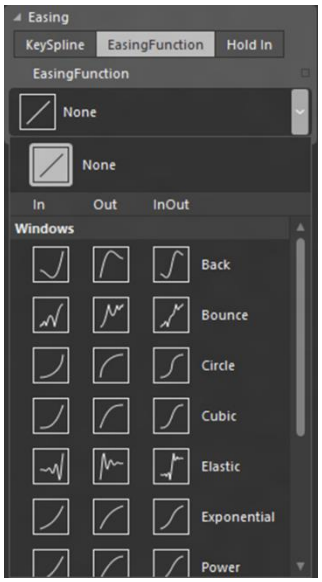
A **Duration** tulajdonságban kell megadni az animáció végrehajtásának időtartamát **óra:perc:másodperc** formátumban.

Természetesen még több animációt is belecsomagolhatunk a **Storyboard**ba. Az animációt a megfelelő esemény hatására (például egy gombnyomás) indíthatjuk a mögöttes kódfájlból a **Storyboard** objektum **Begin()** műveletének hívásával.

A **DoubleAnimation** mellett van még két speciális animáció, a **ColorAnimation** és a **PointAnimation**, melyek neve el is árulja, hogy milyen típusú tulajdonságok animált kezelésére használhatók. Az **ObjectAnimation** igazi „joly joker”, amely tetszőleges típusú tulajdonságot tud egy adott idő elteltével a megfelelő értékre állítani – tehát akár egy Boolean tulajdonságot is tudunk „animálni”.

Minden animációnak vagy egy **UsingKeyFrames** végződésű verziója (pl. **DoubleAnimationUsingKeyFrames**), amellyel – hogy több közbenső állapotot definiáljunk az animációkhoz –, leírhatjuk, hogy az animált tulajdonság menet közben mikor és milyen értéket vegyen fel.

Az eddig elhangzottak alapján azt lehetne gondolni, hogy az animációk idő-elmozdulás görbéje egy egyenes vonal, azaz az animáció közben az egyes állapotok között egyenletes sebességgel vannak az értékek (például szélesség) animálva. Az **ObjectAnimation** típust kivéve az összes többinek van egy **EasingFunction** tulajdonsága („csillapítófüggvény”), melyet sok előre definiált értékből tudunk kiválasztani, amint azt a 6-4 ábra is mutatja.



6-4 ábra: Csillapítófüggvények állapotai Expression Blend-ben

Az animációk futhatnak a CPU-n (függő, *dependent animation*) vagy a GPU-n (független, *independent animation*) is. Minden beépített animáció független, vagyis azok a GPU-n futnak és egyáltalán nem terhelik a CPU-t. Az olyan animációk, amik diszkrét értékeket vesznek fel, és még néhány egyéb, mint például a **ColorAnimation**, már a CPU-n futnak, azaz az animáció köztes értékeinek kiszámítása a UI szálat terheli. Ennek esetleges kellemetlen következménye lehet, hogy túl sok függő animáció párhuzamos futtatása

esetén esetleg azok akadozhatnak, és így kis túlzással talán még annál is rosszabb hatást keltenek, mint ha eleve nem is lettek volna.

Beépített animációk – ThemeAnimation

Az animációkat nem szokás kézzel megírni, erre a kiváló eszköz az Expression Blend, amivel egyszerűen össze lehet kattintgatni egy animációt. Az esetek többségében azonban az animáció két érték között fog változni, az viszont felhasználói élmény szempontjából kritikus, hogy ezt milyen módon teszi. Mennyi idő alatt, milyen csillapítófüggvény használatával és azon belül is milyen paraméterezéssel stb.

Annak érdekében, hogy az egész rendszer és az alkalmazások szintjén konzisztensek legyenek az animációk, a Microsoft rengeteg előre definiált animációt (**`ThemeAnimation`**) ad a kezünkbe, azokat éppen csak hozzá kell kapcsolni egy vezérlőhöz. A 6-1 táblázat ezeket az animációkat foglalja össze.

6-1 táblázat: Beépített animációk

Az animáció neve	Leírás
`DragItemThemeAnimation`	Ezt az animációt akkor látjuk, amikor megfogunk egy elemet a drag-and-drop művelet elején. A kapcsolt vezérlő mérete egy kicsit megnő, és részlegesen átlátszó lesz.
`DragOverThemeAnimation`	Ez az animáció akkor játszódik le, amikor drag-and-drop művelet közben egy potenciális célterület fölé érünk a kurzorral. A csatolt vezérlőt függőlegesen enyhén eltolja.
`DropTargetItemThemeAnimation`	A **`DragItemThemeAnimation`** animáció ellentettje. Elengedéskor visszazsugorodik a megfogott vezérlőelem.
`FadeInThemeAnimation`	Az átlátszóság árnyalásával jeleníti meg a vezérlőt.
`FadeOutThemeAnimation`	Az átlátszóság árnyalásával tünteti el a vezérlőt.
`PopInThemeAnimation`	A vezérlőt az átlátszóság árnyalásával és berepüléssel jeleníti meg.
`PopOutThemeAnimation`	A vezérlőt az átlátszóság árnyalásával és kirepüléssel jeleníti meg.
`PointerDownThemeAnimation`	Ez az animáció egy vezérlő „megnyomását” szemlélteti. A vezérlő kicsit összehúzódik, mintha a virtuális Z tengelyen (mélység) tényleg kicsit hátrébb mozdulna a benyomás hatására.
`PointerUpThemeAnimation`	A vezérlő az elengedésekor visszaáll eredeti méretére.

Visual State Manager

Most már tudod, hogyan kell saját animációkat készíteni és az előre csomagolt animációkat felhasználni. Az animációkat a háttérben különböző események fogják indítani, amelyek általában a felületről vagy az üzleti logika felől érkeznek, és így az animációk a felület állapotainak változásához kapcsolódnak.

Jó példa lehet erre egy regisztrációs űrlap, amelynél egyik állapot az, hogy megnyomható a „regisztráció” gomb. Ehhez például az kell, hogy megfelelő email címet írjunk be, eddig még nem létező felhasználónevet adjunk meg és így tovább. Egy másik állapottal is rendelkezhet a felület, valamiféle hibajelző állapottal, ha még sem sikerült volna érvényes adatot szolgáltatnunk.

De egy másik – sokkal közelibb – példa egy egyszerű gomb. A gombnak is több különböző állapota van. Másképpen jelenik meg, ha éppen felette van a kurzor, ha meg van nyomva, ha alapállapotban van, le van tiltva, esetleg éppen a fókuszban van.

Az ilyen jellegű állapotkezelés megkönnyítésére született a *Visual State Manager* (VSM), mely a nevéből sejthetően a vizuális állapotok közti navigálást könnyíti meg. A VSM állapotcsoportokat tartalmaz, azok pedig konkrét állapotokat. Egy-egy állapot kapcsán definiálni lehet egy **`Storyboard`**ot, amely az adott állapotba „animálja” a rendszert.

A VSM felépítése a következőképpen néz ki (nem teljes kód):

```
<VisualStateManager.VisualStateGroups>
    <VisualStateGroup x:Name="OpenStates">
        <VisualState x:Name="Open">
            <Storyboard>
                <PopInThemeAnimation TargetName="MyContent" />
            </Storyboard>
        </VisualState>
        <VisualState x:Name="Closed">
            <Storyboard>
                <PopOutThemeAnimation TargetName="MyContent" />
            </Storyboard>
        </VisualState>
    </VisualStateGroup>
</VisualStateManager.VisualStateGroups>
```

A VSM-et jellemzően az oldal legmagasabb szintű elemébe (gyökérelem) szokás helyezni, ami általában egy **`Grid`**.

Az állapotcsoportok azért jók, mert a csoporton belüli állapotok kizáróak egymásra nézve, azaz csoportonként mindig csak egy állapot lehet aktív. A fenti példában van egy **OpenStates** csoport, melyen belül két állapot van: **Open** és **Closed**. Ezek közül pedig egy időben csak egy lesz aktív –nincs olyan, hogy valami egyszerre meg is van nyitva meg be is van zárva.

Az állapotok vezérlése pedig kódból a **`VisualStateManager`** osztály **`GoToState()`** metódusával történik a következőképpen:

```
VisualStateManager.GoToState(this, "Open", true);
```

Az első paraméter az a vezérlő (jelen esetben maga az oldal), aminek az állapotai között váltani akarunk. A második paraméter az állapot neve, amire váltani akarunk. A harmadik paraméter pedig azt szabályozza, hogy az állapotváltás pillanatszerű vagy animált legyen-e – ez a kódrészlet az utóbbit használja.

Az alapvető felépítés mellett pedig még lehetőség van arra is, hogy átmeneteket készítsünk két konkrét állapot között egy adott állapotba ugráshoz vagy az állapot elhagyásához. Erre azért lehet szükség, mert elképzelhető olyan helyzet, hogy az értékek alapértelmezett animációja nem tetszik, esetleg nem elegendő, és az állapotok közti átmenet némi finomhangolásra szorul.

Az ilyen vizuális állapotok (és állapotok közti átmenetek) szerkesztésére szintén kiváló eszköz az Expression Blend.

Beépített átmenet-animációk – ThemeTransition

Volt már szó alacsony szintű animációkról, beépített animációkról, vizuális állapotokról, de ezek mindegyikénél a fejlesztőnek kellett elindítani egy animációt vagy kódból átlépni egy másik állapotba. Vannak olyan általánosan használt műveletek, amelyeket csak rá kell csatolni egy adott vezérlőre, és onnantól kezdve az automatikusan figyeli a megfelelő eseményeket, és elvégzi a vizuális állapotok kezelését. Ilyen például az elemek animált megjelenítése, eltüntetése vagy épp átrendezése. Az ilyen előre definiált átmenetek (**`ThemeTransition`**) teljes listáját a 6-2 táblázat foglalja össze.

6-2 táblázat: Beépített átmenet-animációk

Az átmenet-animáció neve	Leírás
`EntranceThemeTransition`	Oldalról becsúszó, árnyalást használó effekt, amelyet egy vezérlő megjelenítésére használhatunk. Bármilyen vezérlőre jól használható.
`ContentThemeTransition`	Egy adott vezérlő tartalmának változásakor kivezeti a régi tartalmat és bevezeti az újat.

Az átmenet-animáció neve	Leírás
RepositionThemeTransition	Egy vezérlő pozícióváltozását „simítja” el.
AddDeleteThemeTransition	Egy vezérlőhöz csatolva annak a felülethez való hozzáadásakor, majd az onnan történő eltávolításakor a vezérlő animáltan fog megjelenni, illetve eltűnni.
ReorderThemeTransition	Kifejezetten listavezérlőknél érdemes használni, az elemek átrendezésekor animálja az elemek átmozgását az új pozíciójukba (nem egyezik meg a **RepositionThemeTransition** nal).

Ezeket az animációkat háromféle módon is használhatjuk.

Rárakhatjuk (akár többet is egyszerre) a vezérlőre annak **Transition** tulajdonságán keresztül.

```
<Button Content="Hello World" >
    <Button.Transitions>
        <TransitionCollection>
            <EntranceThemeTransition />
        </TransitionCollection>
    </Button.Transitions>
</Button>
```

Ilyenkor az animáció az adott elemre vonatkozik, esetünkben a **<Button>**-ra.

Aztán a következő két mód pedig azokra az esetekre vonatkozik, amikor egy vezérlő gyermekelemén vagy gyermekelemein kívánjuk az animációt használni.

Az egyik eset, amikor egy **ContentControl**t használunk, ilyen a legtöbb vezérlő, például a gomb is. Amennyiben azt szeretnénk, hogy a tartalmának változásakor történjen animáció, akkor a **ContentTransitions** tulajdonságán keresztül kell megadnunk azokat:

```
<Button Content="Hello World" >
    <Button.ContentTransitions>
        <TransitionCollection>
            <ContentThemeTransition />
        </TransitionCollection>
    </Button.ContentTransitions>
</Button>
```

A másik eset az, amikor valamilyen **Panel**re (**Grid**, **StackPanel**, **Canvas**, stb.) szeretnénk „ráhúzni” animációt. Ezt az adott **Panel ChildrenTransitions** tulajdonságán keresztül tudjuk megtenni:

```
<StackPanel>
    <StackPanel.ChildrenTransitions>
        <EntranceThemeTransition />
    </StackPanel.ChildrenTransitions>
    <Button Content="#1" />
    <Button Content="#2" />
    <Button Content="#3" />
</StackPanel>
```

Azontúl, hogy elegendő csupán egy helyen megadni az animációt (ahelyett, hogy minden egyes elemhez külön definiálnánk), az is megfigyelhető, hogy a panelben lévő elemek például az **EntranceThemeTransition** esetén enyhe késleltetéssel töltődnek be. Látványos effekt, és ráadásul készen kapjuk!

A **ListView** és **GridView** vezérlők beépítetten tartalmazzák ezeket az animációkat, nincs szükség egyetlen extra kódsor leírására sem.

Adatkötés

A fejezet eddigi részében volt szó arról, hogy miképpen definiálhatunk erőforrásokat, illetve hogyan tudjuk bizonyos események hatására változtatni a felület elemeinek tulajdonságait. Annyi még hiányzik a teljes kép ismeretéből, hogy miképpen lehet a felületet dinamikusan változtatni.

A Microsoft célja a XAML-lel (és a mögötte lévő technológiákkal) az, hogy elkülönülhessen a dizájnerek és a fejlesztők munkája, és így az üzleti logika ne legyen „hozzádrótozva" a felülethez. A kettő összekötése pedig nem kódból fog történni, hanem az ún. *adatkötés* segítségével.

A koncepció alapja, hogy az alkalmazás „motorját" vagy üzleti logikáját teljesen elkülönítve ajánlott fejleszteni, úgy, hogy az semmit nem tud a felületről, majd pedig az alkalmazás állapotát egyszerű osztályokkal, adatstruktúrákkal reprezentálni. Például ha egy oldalon egy személy adatait akarjuk megjeleníteni, akkor lesz az oldal mögött egy **`Person`** típusú objektum, mely egyszerű tulajdonságokat tartalmaz, mint például a **`FirstName`**, **`LastName`**, **`EmailAddress`** és így tovább:

```
public class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string EmailAddress { get; set; }
}
```

Az „oldal mögött" kifejezés az oldal **`DataContext`** tulajdonságát takarja. Minden vezérlőnek – beleértve az oldalt is – van **`DataContext`** tulajdonsága, melynek beállításával egy tetszőleges objektumot adhatunk át kontextusként a vezérlőnek, majd azon a vezérlőn belül hivatkozhatunk ennek az átadott objektumnak a tulajdonságaira. Ez a kontextus hasonló az erőforrásokhoz, csak itt az analógia szemszögéből a gyűjteményt maga az objektum, a gyűjtemény egyes elemeit pedig az objektum tulajdonságai jelentik.

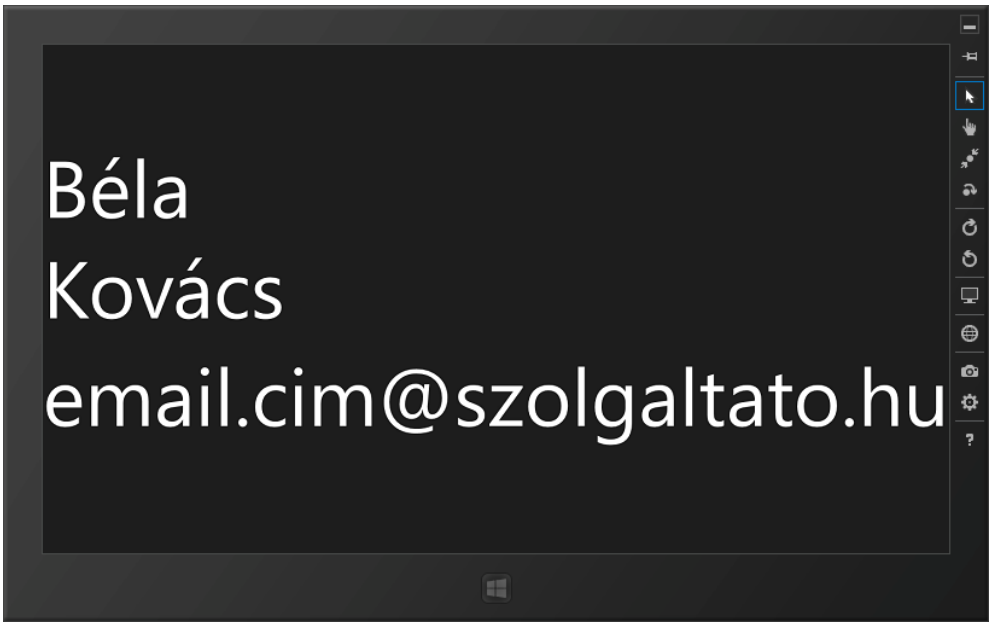
A **`DataContext`** tulajdonság beállítása kódból a következőképpen néz ki:

```
this.DataContext = new Person
{
    FirstName = "Béla",
    LastName = "Kovács",
    EmailAddress = "email.cim@szolgaltato.hu"
};
```

Innentől kezdve vissza lehet térni a XAML kódhoz, mert az egyes tulajdonságokra való hivatkozásokat már ott lehet leírni:

```
<StackPanel>
    <TextBlock Text="{Binding FirstName}" />
    <TextBlock Text="{Binding LastName}" />
    <TextBlock Text="{Binding EmailAddress}" />
</StackPanel>
```

Az eredményt a 6-5 ábra mutatja.



6-5 ábra: Egyszerű adatkötés példa eredménye

A hivatkozás szintaktikája hasonlít az erőforrásra való hivatkozáséra, itt azonban a **`StaticResource`** helyett a **`Binding`** kulcsszót kell használnunk. Az adatkötés hivatkozások feloldása is hasonlít az erőforrásokéra, de nem egyezik meg azokkal. A **`DataContext`**et a gyermekelemek automatikusan öröklik, ha csak ez az öröklődés explicit módon felül nincs bírálva. A hivatkozás pedig csak az adott elem **`DataContext`** objektumának egy tulajdonságára mutathat. Azaz, ha van egy **`Book`** és egy **`Person`** típusunk, és a **`Book`** van az oldal kontextusának beállítva, míg a **`Person`** valahol alacsonyabb szinten, például a fentebb látható **`StackPanel`** kontextusában szerepel, akkor a **`StackPanel`**en belül levő elemek csak a **`Person`** objektum tulajdonságaira hivatkozhatnak.

Az előbbi gondolatmenetnél maradva azt lehet mondani, hogy **`DataContext`** „lefolyik” a vizuális fán. Tegyük ehhez még hozzá, hogy a **`DataContext`** maga is (mint a vezérlők legtöbb tulajdonsága) „adatköthető”. Ez azt eredményezi, hogy ha van egy bonyolult többszintű adatstruktúránk, azt is kényelmesen fogjuk adatkötéssel használni a felületen.

Itt egy példa a bonyolult, többszintű adatstruktúrára (felhasználva az előbbi **`Person`** osztályt):

```
public class Corporation
{
    public string CorpName { get; set; }
    public Person CEO { get; set; }
}
```

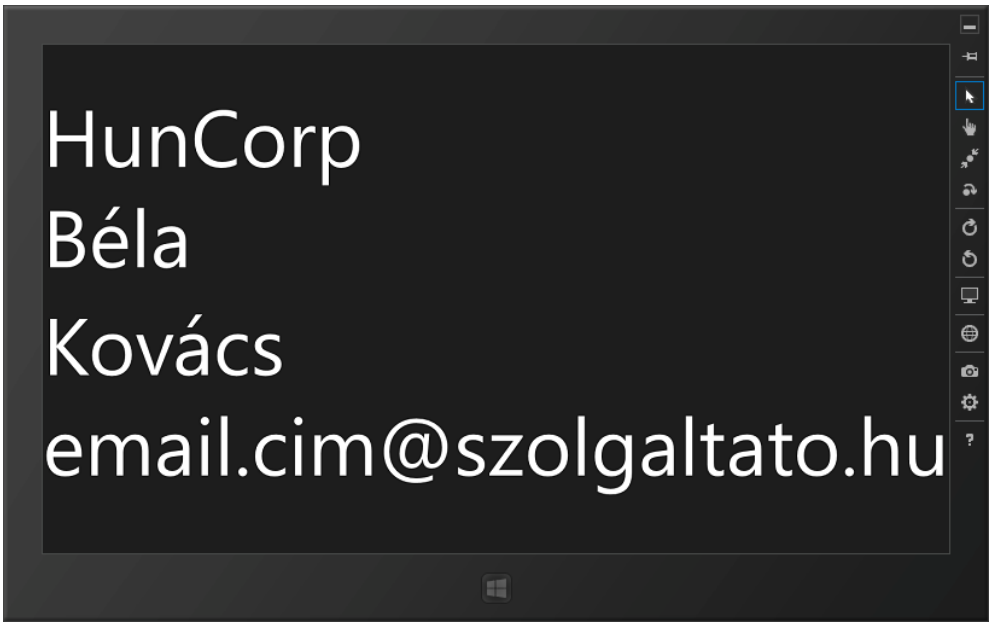
Mikor ebből egy példányt beállítunk az oldal kontextusának, az a következőképpen fog kinézni:

```
this.DataContext = new Corporation
{
    CorpName = "HunCorp",
    CEO = new Person
    {
        FirstName = "Béla",
        LastName = "Kovács",
        EmailAddress = "email.cim@szolgaltato.hu"
    }
};
```

A XAML kód, ami pedig az adatkötést elvégzi, a következőképpen néz ki:

```
<StackPanel>
    <TextBlock Text="{Binding CorpName}" />
    <StackPanel DataContext="{Binding CEO}">
        <TextBlock Text="{Binding FirstName}" />
        <TextBlock Text="{Binding LastName}" />
        <TextBlock Text="{Binding EmailAddress}" />
    </StackPanel>
</StackPanel>
```

Ennek a lépésnek az eredménye a 6-6 ábrán látható.



6-6 ábra: DataContext „lefolyása” a vizuális fán

Természetesen ez lehetne lényegesen bonyolultabb is, és a személyes adatokat megjelenítő rész lehetne egy önálló **`UserControl`**, mely csak annyit feltételez, hogy **`Person`** típusú elemet fog kontextusának kapni, és annak a tulajdonságaira hivatkozik.

Ez a megoldás egyelőre még statikus és nem igazán dinamikus. Fel tudunk használni egy felülettől független adatstruktúrát, melynek hivatkozhatunk a tulajdonságaira, de ennél többet szeretnénk elérni. Tételezzük fel, hogy a vállalat vezérigazgatója az alkalmazás futtatása közben hirtelen megváltozik, az alkalmazás pedig annyira figyelmes, hogy ennek hatására a megjelenített adatokat is azonnal frissíti. Életszerű, ugye?

Ennek eléréséhez egy kicsit módosítani kell a kontextus átadását, és nemcsak egyszerűen „beletolni” az oldal **`DataContext`** tulajdonságába a **`Corporation`** objektumot, hanem eltárolni egy referenciát arra a példányra. Az oldal konstruktora így fog kinézni:

```
// ...
Corporation corp;

//...
public MainPage()
{
    this.InitializeComponent();

    corp = new Corporation
    {
        CorpName = "HunCorp",
        CEO = new Person
        {
            FirstName = "Béla",
```

```
            LastName = "Kovács",
            EmailAddress = "email.cim@szolgaltato.hu"
        }
    };

    this.DataContext = corp;
}
```

Az volna az elvárásunk, hogy a megjelenített tartalom változtatásához a mögötte álló adatstruktúrán kell változtatnunk, és ezt a változást a felület észreveszi és megjeleníti, ahogy és ahol csak akarja.

Helyezzünk egy gombot a felületre!

```
<StackPanel>
    <Button Content="Új Főnököt!" Click="NewBossClick" />
    <TextBlock Text="{Binding CorpName}" />
    <StackPanel DataContext="{Binding CEO}">
        <TextBlock Text="{Binding FirstName}" />
        <TextBlock Text="{Binding LastName}" />
        <TextBlock Text="{Binding EmailAddress}" />
    </StackPanel>
</StackPanel>
```

A gombhoz tartozó **`Click`** eseménykezelő legyen a következő:

```
private void NewBossClick(object sender, RoutedEventArgs e)
{
    corp.CEO = new Person
    {
        FirstName = "Géza",
        LastName = "Kiss",
        EmailAddress = "vallalati.emailcim@vallalat.hu"
    };
}
```

Ám a kód futtatásakor hiába várjuk, hogy a gombra kattintás után a felület Kiss Gézát jelenítse meg, a dolog nem működik! Azért nem, mert a felület csak akkor frissül, ha értesítést kap, hogy a tartalma változott, és ezért frissítenie kell egy adatkötött vezérlő tartalmát! A **`Corporation`** osztály tulajdonságának átírása pedig semmiféle értesítést nem generál. Ezt a problémát az **`INotifyPropertyChanged`** interfész megvalósítása oldja meg:

```
public class Corporation : INotifyPropertyChanged
{
    public string CorpName { get; set; }

    private Person ceo;
    public Person CEO
    {
        get { return ceo; }
        set
        {
            if (value != ceo)
            {
                ceo = value;
                if (PropertyChanged != null)
                    PropertyChanged(this, new PropertyChangedEventArgs("CEO"));
            }
        }
    }

    public event PropertyChangedEventHandler PropertyChanged;
```

```
}
```

Az **INotifyPropertyChanged** interfész mindösszesen egy **PropertyChanged** eseményt tartalmaz. A CEO tulajdonság *setter* metódusa két sablont is alkalmaz. Egyrészt ellenőrzi, hogy a tulajdonság értéke valóban változott-e, és csak ebben az esetben módosítja. Másrészt csak akkor emeli a **PropertyChanged** eseményt, ha van hozzárendelt eseménykezelő művelet. Ha ezt az utóbbi ellenőrzést nem tennénk meg, akkor kivételt dobna a rendszer, mivel nincs senki feliratkozva az eseményre. A **PropertyChanged** eseménynek két paramétere van, az első a küldő – ami a kódok túlnyomó részében a **this** kulcsszót fogja jelenteni (főleg, hogy egy objektum eseményét csak az objektumon belülről lehet dobni) –, míg a második paraméter a megváltozott tulajdonság nevét tartalmazó szokásos esemény argumentum objektum.

Ezután a változtatás után már működik a példaprogram, és a gomb megnyomásakor jelzi a vezérigazgató változását, az új beállításoknak megfelelő név jelenik meg a képernyőn. Mindezt úgy, hogy a kódból hozzá sem kellett nyúlni egyetlen vezérlőhöz sem!

Az adatkötés lehet kétirányú is, vagyis a felhasználói felület változását a rendszer visszavezeti az adatkötés kontextusába. A következő példa a kétirányú adatkötést fogja szemléltetni. Ehhez egy „fizetés” (**Salary**) mezőre van szükség az adatstruktúrában, amely a CEO tulajdonsághoz hasonlóan értesítést küld a saját változásáról, és a felületen egy csúszkára meg még egy címkére.

A felület XAML kódja az alábbi módon változik meg:

```
<StackPanel>
    <Button Content="Új Főnököt!" Click="NewBossClick" />
    <Slider Value="{Binding CEO.Salary, Mode=TwoWay}" />
    <TextBlock Text="{Binding CorpName}" />
    <StackPanel DataContext="{Binding CEO}">
        <TextBlock Text="{Binding FirstName}" />
        <TextBlock Text="{Binding LastName}" />
        <TextBlock Text="{Binding EmailAddress}" />
        <TextBlock Text="{Binding Salary}" />
    </StackPanel>
</StackPanel>
```

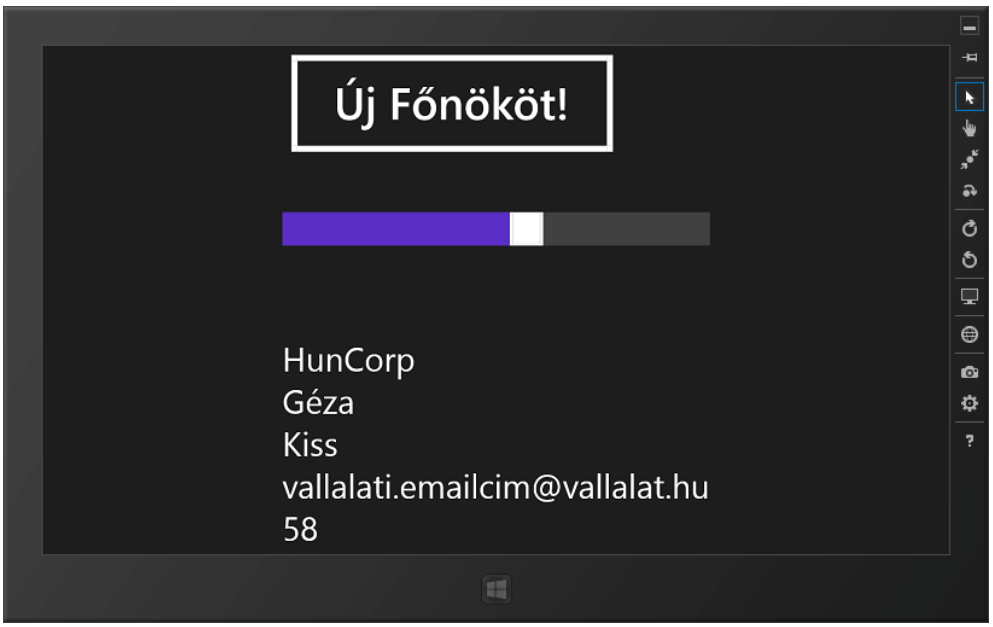
A működéshez a **Person** osztályt az alábbiak szerint kell módosítani:

```
public class Person : INotifyPropertyChanged{    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string EmailAddress { get; set; }

    private int salary;
    public int Salary
    {
        get { return salary; }
        set
        {
            if (value != salary)
            {
                salary = value;
                if (PropertyChanged != null)
                    PropertyChanged(this, new PropertyChangedEventArgs("Salary"));
            }
        }
    }

    public event PropertyChangedEventHandler PropertyChanged;
}
```

És végül a **NewBossClick** metódus módosított változata (az új főnök fizetést is kap):

```
private void NewBossClick(object sender, RoutedEventArgs e)
{
    corp.CEO = new Person
    {
        FirstName = "Géza",
        LastName = "Kiss",
        EmailAddress = "vallalati.emailcim@vallalat.hu",
        Salary = 30
    };
}
```

A példa eredménye a 6-7 ábrán látható.



6-7 ábra: Kétirányú adatkötés példa eredménye

Hogyan is működik mindez? A trükk a **`Slider`** adatkötésénél keresendő. A **`Mode=TwoWay`** beállításával kétirányúvá tettük az adatkötést, aminek az az eredménye, hogy ha a felületen változik a tulajdonság értéke, akkor az visszaíródik az adatkötött tulajdonságba. Azaz, ha mozgatjuk a csúszkát, annak értéke folyamatosan és automatikusan visszakerül az adatkötött tulajdonságba. Ezt a felületen onnan látjuk, hogy a **`Salary`** tulajdonságra rá van kötve egy címke is, így a csúszka mozgatása közben a tulajdonság aktuális értékét a címkén is látni lehet.

Az utolsó példa a listák adatkötését fogja szemléltetni és egyúttal bemutat egy második sablontípust, az adatsablont is.

A felület egy gombból és egy listavezérlőből áll, és a kódja sem bonyolultabb a korábbiaknál. Az előző céges-főnökös példát annyival módosítjuk, hogy egy főnök helyett alkalmazottak listáját tárolja.

A XAML kód tehát a következőképpen néz ki:

```
<StackPanel>
    <TextBlock Text="{Binding CorpName}" />
    <Button Content="Új Beosztottat!" Click="NewEmployeeClick" />
    <ListView ItemsSource="{Binding Employees}">
        <ListView.ItemTemplate>
            <DataTemplate>
                <StackPanel>
                    <TextBlock Text="{Binding FirstName}" />
                    <TextBlock Text="{Binding LastName}" />
                    <TextBlock Text="{Binding EmailAddress}" />
                </StackPanel>
            </DataTemplate>
```

```
        </ListView.ItemTemplate>
    </ListView>
</StackPanel>
```

A példa a **Corporation** osztály egy módosított változatával dolgozik:

```
public class Corporation
{
    public string CorpName { get; set; }
    public ObservableCollection<Person> Employees { get; set; }

    public Corporation()
    {
        Employees = new ObservableCollection<Person>();
    }
}
```

Az új beosztott rögzítéséhez tartozó esemény kódja az alábbi:

```
private void NewEmployeeClick(object sender, RoutedEventArgs e)
{
    corp.Employees.Add(new Person
    {
        FirstName = "Alkal",
        LastName = "Mazott",
        EmailAddress = "alkal.mazott@ceg.hu"
    });
}
```

A példa eredménye a 6-8 ábrán látható.

6-8 ábra: Lista adatkötése és DataTemplate használata

A XAML kódban látható, hogy amikor egy listavezérlő tartalmához akarunk adatot kötni, akkor nem a **DataContext**, hanem az **ItemsSource** tulajdonságát kell állítani. Adatkötés után azonban a lista elemei **Person** típusú objektumok, amelyeket alapértelmezett módon a listavezérlő a **ToString()** metódusmeghívásával jelenít meg. Ám ez nem feltétlenül fedi az elképzeléseinket az adatok megjelenítésével kapcsolatban! Ezen problémát hivatott megoldani az adatsablon (**DataTemplate**), mely egészen pontosan arra alkalmas, hogy egy tetszőleges objektumhoz megjelenést rendeljünk. Ezt a sablont

át kell adni a listának, mégpedig az **`ItemTemplate`** tulajdonsága beállításával. Az ebben leírt vizuális fa fogja jelen esetben a **`Person`** típusú elemek megjelenését meghatározni.

A **`Corporation`** osztály C# kód érdekessége az **`ObservableCollection`** típus. Ez a típus egy olyan lista, ami megvalósítja az **`INotifyCollectionChanged`** interfészt, ami ugyanazt a szerepet játssza listáknál, mint az **`INotifyPropertyChanged`** interfész az objektumoknál: értesítést küld a kapcsolt felhasználói felület elemeknek, ha a konténer tartalma megváltozik. Egyébként hiába pakolgatnánk szorgosan az elemeket egy **`IEnumerable`** típusú listába, senki nem fog arról semmiféle értesítést kapni, ha új elem érkezett, törlődött, vagy ha megváltozott az elemek sorrendje.

Összegzés

Ebben a fejezetben megismerkedhettél a XAML több technikájával, amelyek segítségével egy vezérlő tulajdonságainak deklarációját – vagy épp azok megváltoztatását – elválaszthatod és függetlenítheted a konkrét vezérlőtől, és így növelheted a kód (stílusok, sablonok vagy akár a vezérlők mögött lévő kód) újrafelhasználhatóságát.

A vezérlők tulajdonságait érdemes minél nagyobb mértékben erőforrásokkal és stílusokkal leírni egy központi, jól elérhető és könnyedén szerkeszthető helyen, ahelyett, hogy minden vezérlő helyben „beégetett" tulajdonságokat tartalmazna. Ez egy-egy stíluselem módosításánál is hatalmas segítséget jelent, és a XAML kód is átlátható marad.

Különböző állapotok kezelésére mögöttes kódból való bűvészkedés helyett érdemes animációkat és a Visual State Managert használni, így rengeteg felesleges kódolást spórolhatsz meg amellett, hogy az állapotváltásokat is animálhatod.

A vezérlők tulajdonságainak dinamikus állítása esetén az adatkötés használata az ajánlott, így a XAML kód (felület) és a C# kód (üzleti logika) függetlenek maradhatnak egymástól. Ez növeli a kód átláthatóságát, javíthatóságát, tesztelését, és adott esetben könnyedén le lehet cserélni az alkalmazás teljes felületét az üzleti logika érintése nélkül – például táblagépről telefonra való portolásnál.

7. Modern vezérlők használata Windows 8 stílusú alkalmazásokban

Ebben a fejezetben az alábbi témákat ismerheted meg:

- Adatok hatékony kezelése, szűrése és csoportosítása
- Windows 8 specifikus listás vezérlők – **`ListView`**, **`GridView`** és **`FlipView`** – és használatuk
- Néhány speciális komponens – **`SemanticZoom`** és **`AppBar`** – használata

A Windows 8 operációs rendszer és platform egyik legnagyobb újdonsága a merőben új felhasználói felület. Ez nemcsak az új koncepció, a *Modern UI* Windows operációs rendszerébe való beágyazását jelenti, hanem új, kifejezetten az ilyen stílusú felületekhez tervezett vezérlők megjelenését eredményezte. Ezek a vezérlők azon túl, hogy az új szemlélet és elv szempontjait figyelembe véve kerültek megalkotásra, rendkívül markánsan meghatározzák a Windows 8 alkalmazások megjelenését, valamint az operációs rendszerhez kapcsolódó felhasználói élményt is.

A vezérlők a mai felhasználói felületek minden alapvető igényét teljesítik: teljes mértékű testreszabhatóságot, könnyű kiterjeszthetőséget kínálnak, támogatják a nagyméretű adathalmazokkal végzett hatékony munkát, igényalapú adatbetöltést, kulcsrakész animációkat biztosítanak, kezelik az egér és érintés jellegű interakciókat.

Ebben a fejezetben ezekkel a vezérlőkkel ismerkedünk meg részletesen.

Hatékony adatkezelés a CollectionViewSource segítségével

Az előző fejezetekben az adatkötésről szóló bekezdésekben megismerkedhettünk a listás adatkötéssel, és ezen belül is az **`ObservableCollection<T>`** típussal. Ez segített abban, hogy a listában történő változásokat – elemek hozzáadása, törlése –a felhasználói felület, illetve a **`Binding`** objektum felé delegáljuk az **`INotifyCollectionChanged`** interfészen keresztül. Amint a lista megváltozik, bekövetkezik a **`CollectionChanged`** esemény, és a **`Binding`** objektum kiértesítésre kerül, így a felhasználói felületet a futásidejű környezet frissíti. Ez a mechanizmus tartja szinkronban az adathalmazt és a kapcsolódó vezérlőket. Bár ez rendkívül kényelmes, sokkal kevésbé hatékony, mint gondolnánk, különösen, ha nagyobb változásokról van szó.

Gondoljuk végig, hogy mi történne, ha egy nevekből álló A-Z-ig ábécérendben rendezett listát szeretnénk átrendezni Z-A-ig, azaz éppen ellentétes irányba. Ahogy rendezési algoritmusunk folyamatosan egyesével helyezné át az elemeket az **`ObservableCollection`**-ben, úgy következnének be minden lépésnél a **`CollectionChanged`** események, melyek hatására a rendszer megpróbálná szinkronban tartani a felhasználói felületet. Ez a mechanizmus túlságosan leterhelné a felhasználói felületet, nem is beszélve arról, hogy a rendezést és az eredeti állapot tárolását minden egyes esetben nekünk kellene megvalósítani. Így már nemcsak a felhasználói felületre helyeznénk túlzott terhet, de a fejlesztőkre is, a produktivitást jelentős mértékben csökkentenénk. Pontosan ugyanez a helyzet nemcsak a rendezés, hanem a szűrés és a csoportosítás esetében is.

Az ilyen problémák megoldására az eredeti lista, az **`ObservableCollection<T>`** fölé egy speciális objektumot, a **`CollectionViewSource`**-ot használhatjuk.

A **`CollectionViewSource`** objektum az eredeti listát elrejtve egy nézetet képez az adatokról. Ezen a nézeten hajthatjuk végre a szűrés, a rendezés és a csoportosítás műveleteket. A műveletek hatása csupán a nézetet érinti, az eredeti listát érintetlenül hagyja, így a lista eredeti állapotának tárolásáról nem kell

gondoskodnunk. Ennél is fontosabb, hogy a **CollectionChanged** esemény csak a művelet befejezését követően történik meg, a műveletek végrehajtása közben nem. A **CollectionViewSource** használata az eredeti lista használata helyett semmilyen hátránnyal nem jár, így az eredeti listán bekövetkező változásokat is azonnal közvetíti a **Binding** objektum és a felhasználói felület felé.

Az alábbi kódpélda bemutatja a **CollectionViewSource** használatát:

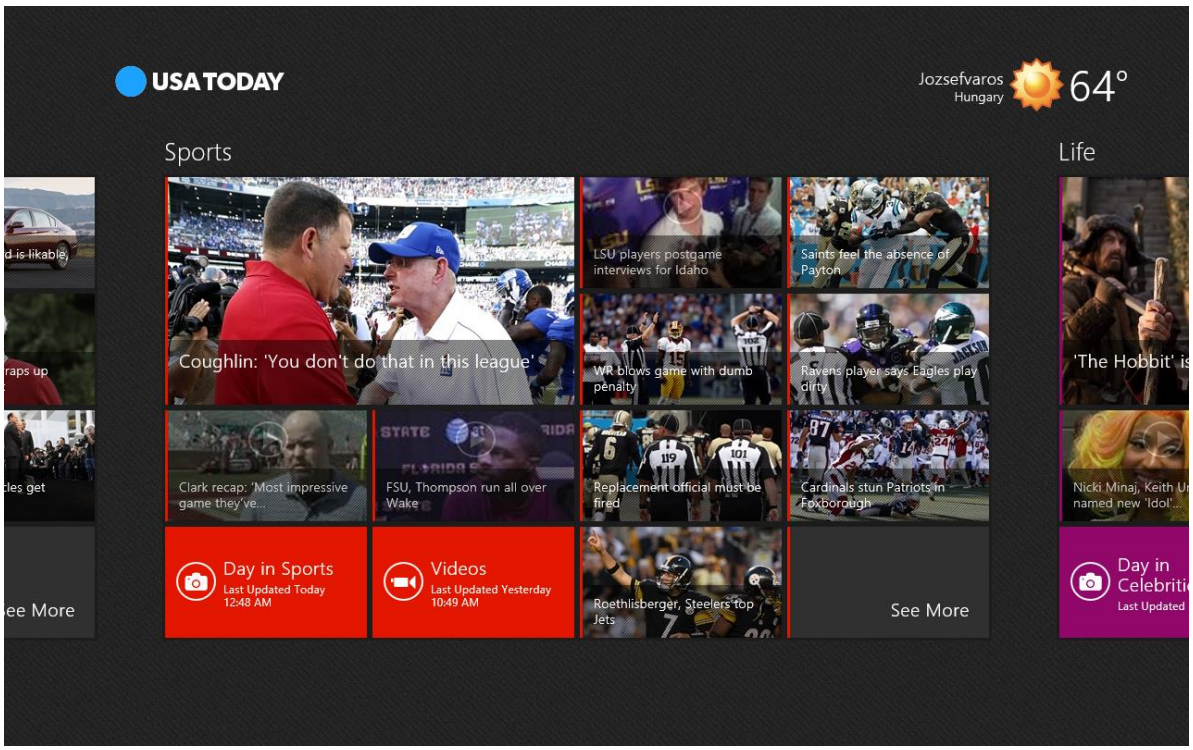
```
var food = new ObservableCollection<string>(new[] { "spaghetti", "ravioli", "lasagna",
    "steak" });

var cvs = new CollectionViewSource { Source = food };
listView.ItemsSource = cvs.View;
```

A fenti példában egy **string** listát adunk át a **CollectionViewSource** objektumnak a **Source** tulajdonságán keresztül. A **CollectionViewSource** egy speciális objektum. A listás vezérlő számára az értékes információ azonban a **View** tulajdonságban van. A **View** az a nézet, amit a **CollectionViewSource** előállít. Ez reprezentálja azt az adathalmazt, amit meg akarunk jeleníteni. A **CollectionViewSource View** tulajdonsága implementálja az **IEnumerable** interfészt, így tetszőleges **ItemsControl ItemsSource** tulajdonságához hozzárendelhető.

A **CollectionViewSource** objektum Windows 8 alkalmazások esetén a korábban, esetleg más platformon megismert és a fent említett képességekhez képest limitált funkcionalitással rendelkezik. A szűrést és a rendezést jelenleg *nem támogatja*! A korlátozások ellenére az objektum jelentősége nem csökkent, a csoportosítás, valamint az adatnavigáció továbbra is fontos szerepet játszik az alkalmazások életében.

Csoportosítás a CollectionViewSource segítségével

A fejezet későbbi bekezdéseiben részletesen foglalkozunk a **GridView** vezérlővel és annak működésével. A vezérlő ismeretének hiányában egyelőre gondolatainkban helyettesítsük azt a 7-1 ábrán láthatóval!



7-1 ábra: A GridView vezérlő egy kész Windows 8 alkalmazásban

Jól látható, hogy az adatok listája, jelen esetben cikkek a kategóriájuk szerint (sport, életmód stb.) csoportosítva vannak. Az ilyen csoportok kezelésére és a **GridView**, **ListView** vezérlőkkel való összekapcsolására a legalkalmasabb objektum a **CollectionViewSource**. Az alábbi példakód demonstrálja az adatforrás szerkezetét:

```
public class News
{
    public string Title { get; set; }
    public string NewsBody { get; set; }
    public DateTime Date { get; set; }
    public string Author { get; set; }
}

public class Category
{
    public int CategoryId { get;set; }
    public string CategoryName { get; set; }
    public ObservableCollection<News> NewsList { get; set; }
}

public class MainPageViewModel
{
    public ObservableCollection<Category> Categories { get; set; }
}
```

A felhasználói felületet a **MainPageViewModel**-ben található **Categories** gyűjteményhez kötve a kategóriák listáját kaphatjuk meg. Minden egyes kategória példányhoz tartozik egy hír lista (**NewsList**). Azonban hiába kötnénk ezt az adatstruktúrát egy **GridView** vezérlőhöz, attól még a vezérlő nem tudná, hogy itt bármi is csoportosítva van, nem értené meg az adatszerkezetet. A **CollectionViewSource** éppen ebben segíthet!

Az alábbi kódban látható, miként kell felkonfigurálni a **CollectionViewSource** objektumot, hogy a fenti adatszerkezet alapján csoportokat képezzen:

```
public class MainPageViewModel
{
    public ObservableCollection<Category> Categories { get; set; }
    public ICollectionView CategoriesView { get; set; }

    public void MainPageViewModel()
    {
        ...
    }

    public void Initialize()
    {
        var cvs = new CollectionViewSource
        {
            Source = Categories,
            IsSourceGrouped = true,
            ItemsPath = new PropertyPath("NewsList")
        };
        CategoriesView = cvs.View;
    }
}
```

A kódban a **CollectionViewSource** objektum **IsSourceGrouped** tulajdonságát beállítjuk **true** értékre, ezzel is jelezve, hogy az adatforrás csoportosított struktúrával rendelkezik. Az **ItemsPath** tulajdonság azt határozza meg, hogy a csoportot reprezentáló lista elemeinek mely tulajdonsága tartalmazza a csoporton belüli elemeket. Jelen esetben ez a **NewsList** tulajdonság.

Adatnavigáció a CollectionViewSource segítségével

A **CollectionViewSource** egy másik előnye az adatnavigáció. A kapcsolódó nézeten az aktuálisan kiválasztott elemet (**CurrentItem**) manipulálhatjuk a listán előre vagy hátra lépdelve. Ahogy lépkedéssel

változik az aktuális adat, az a felhasználói felületre is visszavezetésre kerül, így a kiválasztott elem ott is folyamatosan változik.

Az alábbi példakód az adatnavigációt demonstrálja:

```
var food = new ObservableCollection<string>(new[] { "spaghetti", "ravioli", "lasagna",
    "steak" });
var cvs = new CollectionViewSource { Source = food };
listView.ItemsSource = cvs.View;

if(cvs.View.IsCurrentAfterLast)
{
    cvs.View.MoveCurrentToLast();
}
else if(cvs.View.IsCurrentBeforeFirst)
{
    cvs.View.MoveCurrentToFirst();
}

cvs.View.MoveCurrentToNext();
cvs.View.MoveCurrentToPrevious();
cvs.View.MoveCurrentToPosition(2);
```

A kód **`if…else`** szerkezete biztosítja, hogy se előre, se hátrafelé ne lehessen a listáról lenavigálni. Amint a listáról lelépnénk, automatikusan az első, illetve az utolsó elemre pozicionálunk. Az utolsó három kódsor pedig az előre, a hátra, illetve az adott indexű pozícióba lépést demonstrálja.

Listás adatok megjelenítése és a ListViewBase osztály

Az előző fejezetrészben azt taglaltuk, hogy miként érdemes előkészíteni az adatokat a listás vezérlők számára. Ebben a részben a listás vezérlők alapjaival ismerkedünk meg.

A XAML alapú technológiák a listákat **`ItemsControl`**-ok segítségével reprezentálják. Ez az ősosztály, ami egyben önálló vezérlő is, azonban önmagában nagyon keveset tud. A Windows 8 stílusú alkalmazásokban megjelenő listás vezérlők azonban ennél sokkal többet igényelnek! Ezért ezek a vezérlők egy az **`ItemsControlból`** származó osztályból, a **`ListViewBase`**-ből származnak. Az operációs rendszer jelenlegi verziójában mindössze két ilyen vezérlő van, a **`ListView`** és a **`GridView`**. A két vezérlő különlegessége, hogy igazából saját kóddal, logikával nem rendelkeznek. Minden tudásukat a **`ListViewBase`**-ből nyerik. Különbség csupán abban van, hogy az elemeket milyen alapértelmezett működés és elrendezés szerint prezentálják. Egy kis testreszabással könnyen lehet a **`GridView`**-ból **`ListView`**-t, **`ListView`**-ból pedig **`GridView`**-t csinálni. Így a legfontosabb és legizgalmasabb feladat a **`ListViewBase`** osztály megismerése.

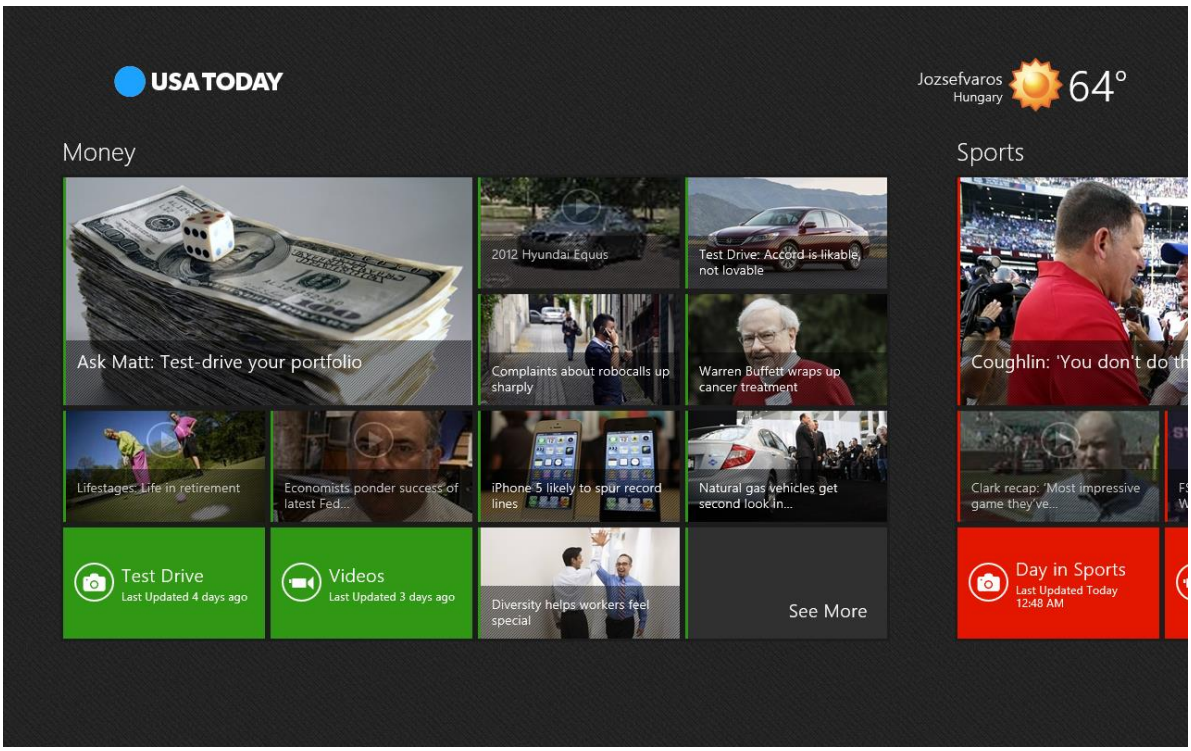
A **`ListViewBase`**-ből származó vezérlők a következő funkciókat támogatják:

- Csoportok kezelése, megjelenítése, testreszabása
- **`SemanticZoom`** vezérlőben felhasználhatók (jelenleg csak a **`GridView`** és a **`ListView`** tudja ezt)
- Aszinkron, igényalapú adatbetöltés
- Virtualizáció (amíg olyan panelt használunk, ami ezt támogatja)
- Egér és érintés események egymással ekvivalens kezelése
- Elemek kiválasztása, átrendezése „fogd és húzd” (*drag-and-drop*) módszerrel

Láthatjuk, hogy a **`ListViewBase`** osztály számos funkcionalitást támogat. Az egyes tulajdonságok és képességek így a **`GridView`** és a **`ListView`** vezérlőre egyaránt jellemzőek. Az egyszerűség kedvéért a következő bekezdésekben a **`GridView`** vezérlőn mutatjuk be ezt a funkcionalitást.

A GridView vezérlő

A **GridView** vezérlő a **ListViewBase** osztályból származó listás adatok megjelenítésére szolgáló vezérlő, a Windows 8 Modern alkalmazások egyik zászlóshajója, szinte minden alkalmazásban megtalálható valamilyen formában. A 7-2 ábrán ez a vezérlő látható használat közben, teljes pompájában.



7-2 ábra:Egy GridView, csoportosítva

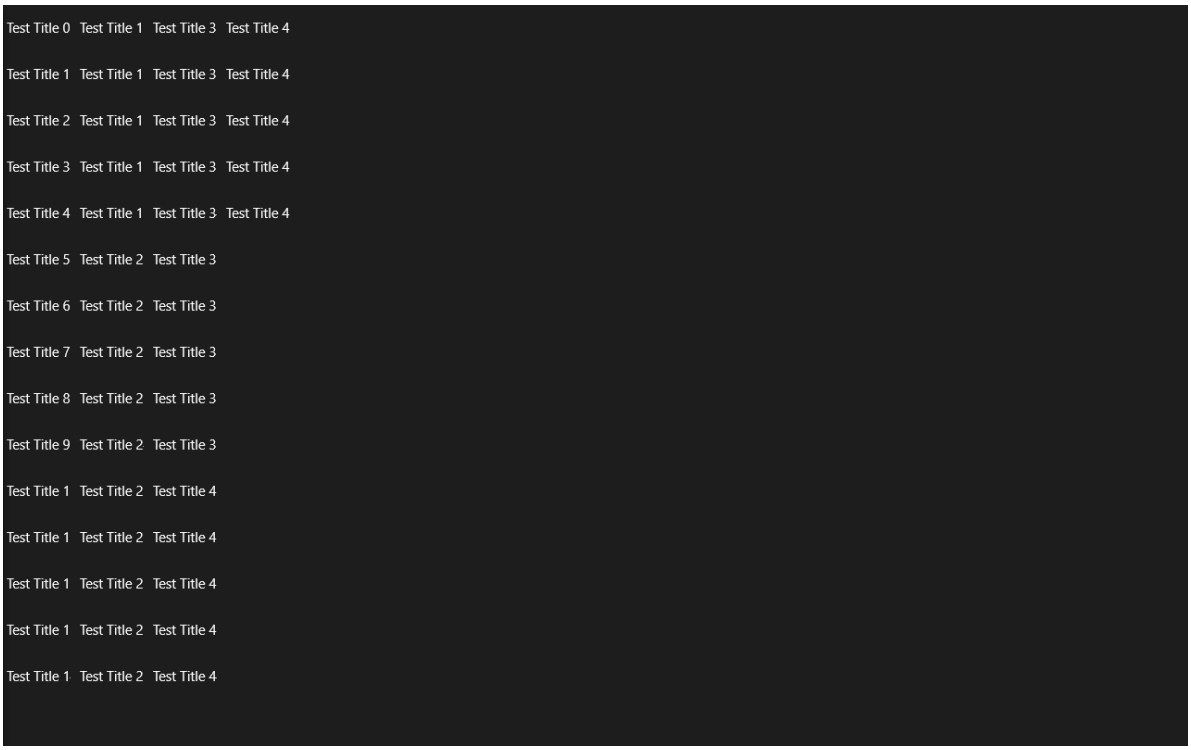
Adatok megjelenítése a GridView vezérlőben

A **GridView** vezérlő ősei között az **ItemsControl** osztály is szerepel, és ennek megfelelően minden ahhoz kapcsolódó tudásunk itt is újrahasznosítható. A **GridView**-t az **ItemsSource** tulajdonságán keresztül tölthetjük fel adatokkal. A **GridView** az elemeket saját logikája szerint rendezi el. Az alábbi példakód demonstrálja a vezérlő használatát:

```
// A felület XAML leírása
<Page
    x:Class="ComplexControlsDemo.MainPage"
    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"
    xmlns:viewModels="using:ComplexControlsDemo.ViewModels" mc:Ignorable="d">

    <Page.DataContext>
        <viewModels:MainPageViewModel/>
    </Page.DataContext>

    <Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
        <GridView ItemsSource="{Binding NewsView}" DisplayMemberPath="Title"/>
    </Grid>

</Page>

// A felület logikáját leíró C# kód
using System;
using System.Collections.ObjectModel;
```

```
using ComplexControlsDemo.Model;
using Windows.UI.Xaml.Data;

namespace ComplexControlsDemo.ViewModels
{
    public class MainPageViewModel
    {
        public ObservableCollection<News> News { get; set; }
        public ICollectionView NewsView { get; set; }

        public MainPageViewModel()
        {
            News = new ObservableCollection<News>();

            LoadData();
            Initialize();
        }

        public void LoadData()
        {
            for (int i = 0; i < 50; i++)
            {
                News.Add(new News
                            {
                                Author = "Test Author" + i,
                                Date = DateTime.Now,
                                Title = "Test Title " + i,
                                PhotoUrl = "/Assets/NewYork.jpg",
                                NewsBody = "Test Body " + i
                            });
            }
        }

        public void Initialize()
        {
            var cvs = new CollectionViewSource
                        {
                            Source = News,
                        };

            NewsView = cvs.View;
        }
    }
}
```

A fenti példakódban a **`GridView ItemsSource`** tulajdonságát a **`NewsView`** tulajdonságra adatkötjük. A **`DisplayMemberPath`** tulajdonság határozza meg, hogy a bekötött **`News`** példányok mely tulajdonságát használjuk megjelenítésre. Az eredmény a 7-3 ábrán látható.



7-3 ábra: A GridView alapértelmezett elrendezése

Layout testreszabása

A 7-3 ábra bemutatja, hogy miként helyezi el a **`GridView`** alapértelmezés szerint az elemeket. A **`GridView`** panelként **`WrapGrid`**-et használ, azaz egyforma szélességű és magasságú cellák jönnek létre minden esetben. Az elemeket pedig horizontálisan vagy vertikálisan helyezi el. Az alapértelmezett viselkedés szerint először az oszlopokat tölti ki, és ha már nincs több szabad sor, akkor lép át a következő oszlopba. A vezérlő megjelenítési stratégiája azonban könnyedén módosítható! Az **`ItemsControl`**-tól örökölt tulajdonság szerint az **`ItemsPanel`** tulajdonságot felüldefiniálva kicserélhetjük az elrendezés kezeléséhez használt panelt. Az alábbi példakód ezt a lépést demonstrálja:

```
<GridView ItemsSource="{Binding News}" DisplayMemberPath="Title">
    <GridView.ItemsPanel>
        <ItemsPanelTemplate>
            <VariableSizedWrapGrid Orientation="Horizontal" MaximumRowsOrColumns="9" />
        </ItemsPanelTemplate>
    </GridView.ItemsPanel>
</GridView>
```

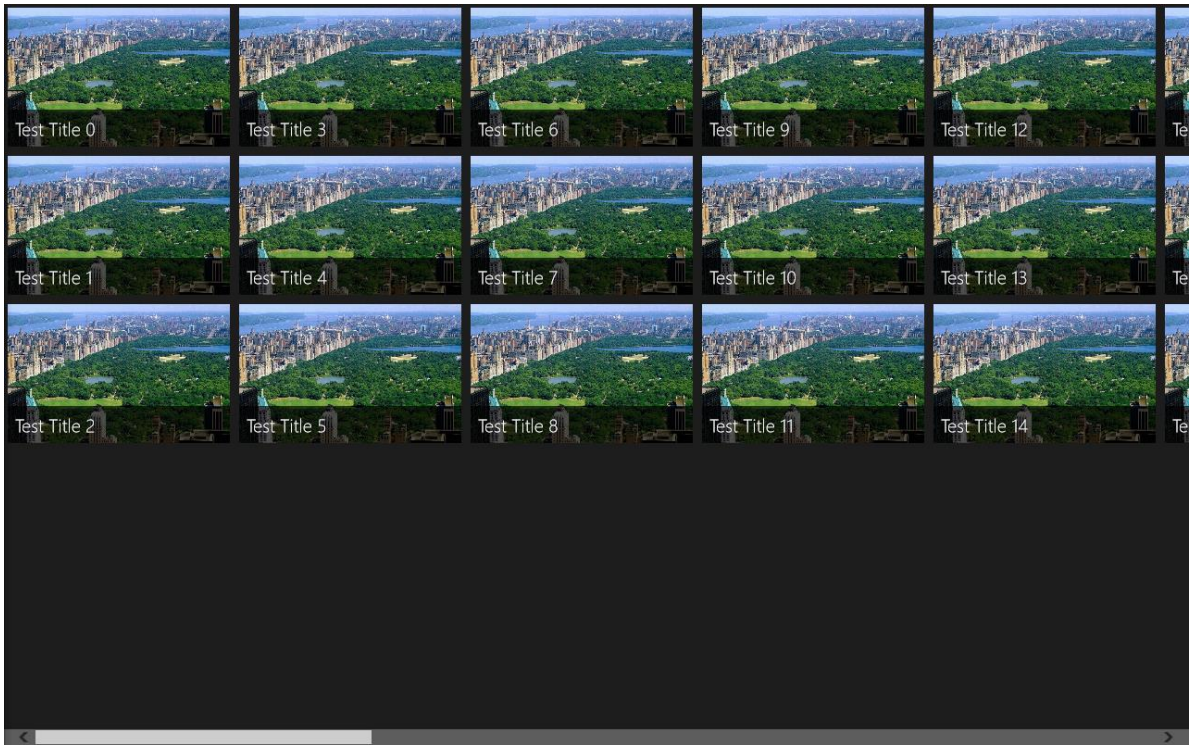
A kód egy új **`ItemsPanelTemplate`**-et rendel a **`GridView`** **`ItemsPanel`** tulajdonságához, ezzel jelezve, hogy a **`VariableSizedWrapGrid`** végzi majd az elemek elrendezését. De itt bármilyen más panel definíciója is állhatna.

Elemek testreszabása

A 7-3 ábrán látható, hogy az adataink bár szépen megjelennek, azok sem az első, sem a második elrendezéssel nem tűnnek izgalmasnak. Ahhoz, hogy ezen változtassunk, testre kell szabnunk az elemek megjelenését. Ezt szintén az **`ItemsControl`**-tól örökölt **`ItemTemplate`** tulajdonsággal tehetjük meg. Az **`ItemTemplate`** tulajdonság egy adatsablont (**`DataTemplate`**) definiál. Ez a sablon határozza meg, hogy egy adatelem a listában milyen felületi elemek (vezérlők) együtteseként jelenik meg. A sablonban található egyes vezérlőelemek adatkötöthetők az éppen megjelenítendő adategység egyes tulajdonságaihoz. Az alábbi példakód saját adatsablon definiálását mutatja be:

```
<Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
    <GridView ItemsSource="{Binding News}">
        <GridView.ItemsPanel>
            <ItemsPanelTemplate>
                <VariableSizedWrapGrid ItemHeight="160" ItemWidth="250"
                    MaximumRowsOrColumns="3" />
            </ItemsPanelTemplate>
        </GridView.ItemsPanel>

        <GridView.ItemTemplate>
            <DataTemplate>
                <Grid>
                    <Grid.RowDefinitions>
                        <RowDefinition Height="*"/>
                        <RowDefinition Height="40"/>
                    </Grid.RowDefinitions>

                    <Image Source="{Binding PhotoUrl}" Grid.RowSpan="2"
                        Stretch="UniformToFill"/>
                    <Rectangle Fill="#AA000000" Grid.Row="1"
                        HorizontalAlignment="Stretch" VerticalAlignment="Stretch"/>
                    <TextBlock Text="{Binding Title}" Grid.Row="1"
                        HorizontalAlignment="Left" VerticalAlignment="Center"
                        Margin="8,0,0,0" FontSize="20" FontFamily="Segoe UI"
                        FontWeight="Light"/>
                </Grid>
            </DataTemplate>
        </GridView.ItemTemplate>
    </GridView>
</Grid>
```

Ha kicsit közelebbről megnézzük a kódot, látható, hogy apró módosítások is bekerültek a kiegészítésen túl. Így az **`ItemsPanelTemplate`**-ben található **`VariableSizedWrapGrid`**-en meghatároztuk, hogy az elemek mérete fixen 250x160 pixel lehet, valamint azt, hogy a vezérlő maximum 3 sorból állhat. Az **`ItemTemplate`** belsejében egy **`Grid`**et definiálunk, amelyben a cikkhez tartozó képet és a címet jelenítjük meg. A testreszabás eredménye a 7-4 ábrán látható.



7-4 ábra: GridView saját `ItemTemplate`-ekkel

Haladó testreszabás

Ha visszalapozol a 7-1 ábrához, láthatod, hogy a fő hír kiemelésre került. Tekintve, hogy a **GridView** belsejében található panel egy **VariableSizedWrapGrid**, átalakíthatjuk úgy a kódunkat, hogy bizonyos elemek átnyúlhassanak sorokon es oszlopokon. Ez egy viszonylag gyakori UI minta, amit sokan alkalmaznak. Sajnálatos módon ahhoz, hogy elérjük a kívánt hatást, jelenleg alkalmaznunk kell pár trükköt. A trükk lényege, hogy felüldefiniáljuk a **GridView** egy metódusát, ami azért felelős, hogy az adat és az **ItemTemplate**-ben található vezérlők megfelelően összehangolódjanak, és némi vizsgálatot követően a vezérlőkön beállítsuk a **VariableSizedWrapGrid.RowSpan**, illetve a **VariableSizedWrapGrid.ColumnSpan** tulajdonságokat.

Az első lépés a modell kiterjesztése az egyszerű és a fő hírek megkülönböztetésére. Az alábbi kódblokk ezeket a lépéseket demonstrálja.

Az **IItemType.cs** file tartalma:

```
public interface IItemType
{
    ItemType ItemType { get; set;}

}

public enum ItemType
{
    Normal,
    Main
}
```

A módosított **News.cs** tartalma:

```
public class News : IItemType
{
    public string Title { get; set; }
    public string PhotoUrl { get; set; }
    public string NewsBody { get; set; }
    public DateTime Date { get; set; }
    public string Author { get; set; }
    public ItemType ItemType { get; set; }
}
```

A módosított **MainPageViewModel.cs** tartalma:

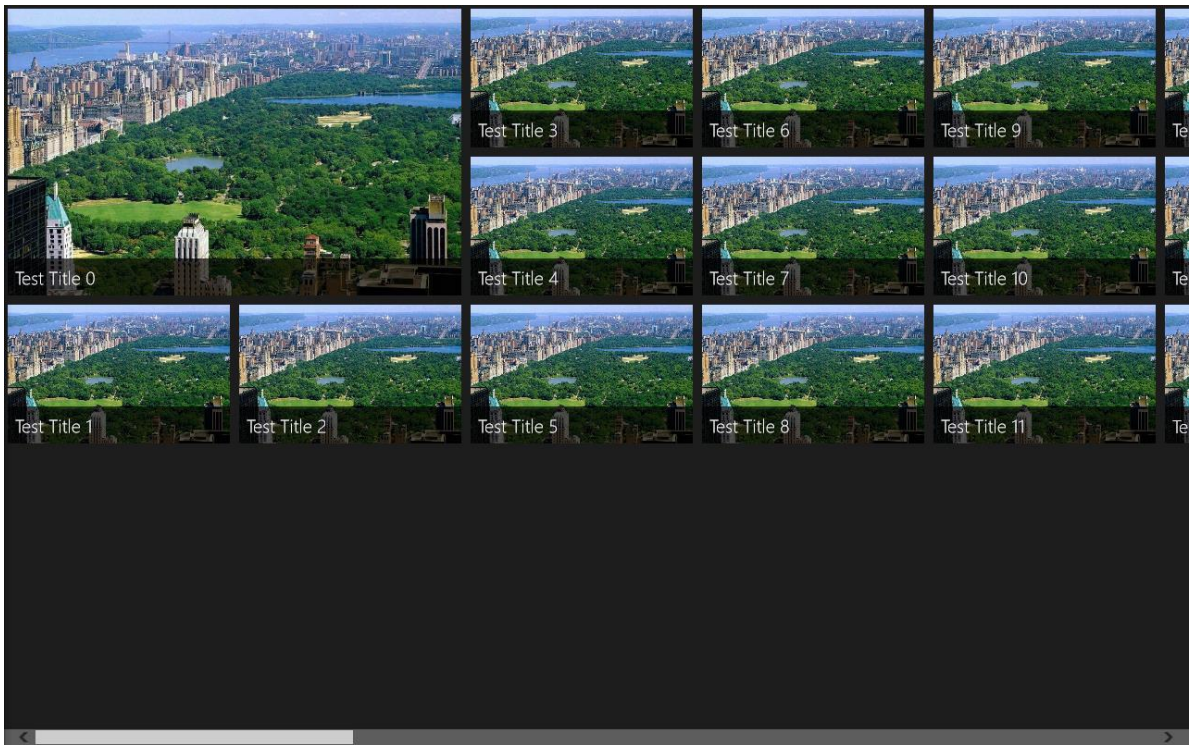
```
public class MainPageViewModel
{
    ...

    public void LoadData()
    {
        for (int i = 0; i < 50; i++)
        {
            News.Add(new News
                            {
                                Author = "Test Author" + i,
                                Date = DateTime.Now,
                                Title = "Test Title " + i,
                                PhotoUrl = "/Assets/NewYork.jpg",
                                NewsBody = "Test Body " + i
                            });
        }

        News.First().ItemType = ItemType.Main;
    }
```

```
    ...
}
```

Végül a **GridView** osztályt is ki kell egészítenünk, hogy a **News** példány **ItemType** tulajdonsága alapján beállítsuk a sorokon és az oszlopokon való átnyúlást. Az alábbi példakód ezt a lépést demonstrálja:

```
public class NewsGridView : GridView
{
    protected override void PrepareContainerForItemOverride(Windows.UI.Xaml.DependencyObject element,
        object item)
    {
        var dataItem = item as IItemType;

        if (dataItem != null && dataItem.ItemType == ItemType.Main)
        {
            element.SetValue(VariableSizedWrapGrid.RowSpanProperty, 2);
            element.SetValue(VariableSizedWrapGrid.ColumnSpanProperty, 2);
        }

        base.PrepareContainerForItemOverride(element, item);
    }
}
```

Ennek eredményeként, amikor egy elemet, illetve annak konténerét a **NewsGridView** el szeretné készíteni, ellenőrizzük, hogy az adatkötött hír (**item**) **ItemType** tulajdonsága **Main**-e. Ha igen, akkor a kapcsolódó vezérlő vagy panel (**element**) objektumon állítjuk be a **VariableSizedWrapGrid** panel **RowSpan** és **ColumnSpan** csatolt tulajdonságait. Az eredmény a 7-5 ábrán látható.



7-5 ábra: A Gridview elemei közül az első átnyúlik sorokon és oszlopokon

Elemek kiválasztása

A **GridView**-ra kattintva azt tapasztalhatjuk, hogy az egyes elemeket ki lehet választani. Ez a lehetőség a **SelectionMode** tulajdonságtól függ, amely az alábbi értékeket veheti fel:

- **None** – Az elemek kiválasztása nem támogatott
- **Single** – Egyszerre csak egy elem lehet kiválasztva
- **Extended** – Több elem együttes kijelölése is lehetséges, a CTRL vagy a SHIFT billentyűk lenyomása mellett.
- **Multiple** – Az elemre kattintás vagy annak megérintése kiválasztja az elemet. Az elem kiválasztás megszüntetése csak újabb kattintással vagy érintéssel lehetséges.

Csoportok kezelése

A 7-2 ábrán látható, hogy a **GridView** elemei csoportokba vannak szervezve. A cikkek kategóriájuk szerint csoportosítva vannak. Ez a megjelenítés rendkívül gyakori **GridView** vezérlők használatánál. A fejezet elején a legfontosabb információt, a **CollectionViewSource** használatát már tárgyaltuk, így itt csupán a **GridView** vezérlő testreszabása kerül a fókuszba.

Az első és legfontosabb előfeltétel, hogy az adatforrás csoportosítva legyen, a **CollectionViewSource** pedig ennek megfelelően felparaméterezve. Ennek eléréséhez a „Csoportosítás a **CollectionViewSource** segítségével” című bekezdésben leírt adatmodellt használjuk.

A csoporthoz tartozó megjelenítési beállításokat a **GridView GroupStyle** tulajdonságán keresztül érhetjük el. A **GroupStyle** segítségével meghatározhatjuk, hogy milyen legyen az elemek elrendezése a csoporton belül, hogyan nézzen ki a csoportot magába foglaló konténer, milyen elemek legyenek a csoport fejlécében stb.

Az alábbi példakód a kategória – hírek csoportosítását jeleníti meg:

```
<Controls:NewsGridView ItemsSource="{Binding CategoriesView}" SelectionMode="Multiple">
    <GridView.ItemsPanel>
        <ItemsPanelTemplate>
            <StackPanel Orientation="Horizontal"/>
        </ItemsPanelTemplate>
    </GridView.ItemsPanel>
    <GridView.ItemTemplate>
        <DataTemplate>
            <Grid>
                <Grid.RowDefinitions>
                    <RowDefinition Height="*"/>
                    <RowDefinition Height="40"/>
                </Grid.RowDefinitions>

                <Image Source="{Binding PhotoUrl}" Grid.RowSpan="2"
                    Stretch="UniformToFill"/>
                <Rectangle Fill="#AA000000" Grid.Row="1" HorizontalAlignment="Stretch"
                    VerticalAlignment="Stretch"/>
                <TextBlock Text="{Binding Title}" Grid.Row="1" HorizontalAlignment="Left"
                    VerticalAlignment="Center"
                    Margin="8,0,0,0" FontSize="20" FontFamily="Segoe UI"
                    FontWeight="Light"/>
            </Grid>
        </DataTemplate>
    </GridView.ItemTemplate>

    <GridView.GroupStyle>
        <GroupStyle HidesIfEmpty="True">
            <GroupStyle.HeaderTemplate>
                <DataTemplate>
                    <TextBlock Text="{Binding CategoryName}"
                        Style="{StaticResource GroupHeaderTextStyle}"/>
                </DataTemplate>
            </GroupStyle.HeaderTemplate>

            <GroupStyle.Panel>
                <ItemsPanelTemplate>
```

```
                    <VariableSizedWrapGrid ItemHeight="160" ItemWidth="250"
                        MaximumRowsOrColumns="3" />
                </ItemsPanelTemplate>
            </GroupStyle.Panel>
        </GroupStyle>
    </GridView.GroupStyle>
</Controls:NewsGridView>
```

Ha az adatokat csoportosítva akarjuk megjeleníteni, akkor azokat a csoportok listájára kell adatkötnünk, nem pedig az elemekre. Ezzel a váltással azonban egy fontos módosítást is el kell végeznünk! Korábban a **`GridView ItemsPanel`** tulajdonsága azt határozta meg, hogy az egyes elemek, jelen esetben hírek milyen elrendezésben szerepelnek majd. A csoportosítás után viszont ez a panel nem közvetlenül az adatelemeket (a híreket) tartalmazza, hanem azok csoportjait. Így a **`GridView ItemsPanel`**jében található panel a csoportok elrendezését határozza meg. A csoporton belüli elemek elrendezését majd a **`GroupStyle`** tulajdonsággal kell meghatározni. Jelen példában a **`StackPanel`**t választottuk horizontális elrendezéssel, így a csoportok egymás mellé fognak kerülni. A fenti példakódban a **`GroupStyle HidesIfEmtpy`** tulajdonsága, illetve annak **`true`** értéke biztosítja, hogy ne jelenjen meg olyan csoport, amiben nincsenek elemek, azaz nem lesz olyan kategória, amiben nincs hír.

A **`GroupStyle HeaderTemplate`** tulajdonsága segítségével határozzuk meg, hogy a csoport fejlécében mit jelenítünk meg. Jelen esetben a kategória nevét írjuk ki minden csoport fölé.

Végül a **`GroupStyle Panel`** tulajdonságával azt határozzuk meg, hogy a csoporton belül az elemek milyen elrendezés szerint legyenek elhelyezve.

Igényalapú adatletöltés

A **`ListViewBase`** egyik legizgalmasabb újdonsága az igényalapú adatletöltés kezdeményezése. Korábban figyelni kellett a listás vezérlő **`ScrollBar`**jának állapotát, és amikor az a végéhez közeledett, automatikusan be kellett tölteni az új elemeket, majd miután az adatok megérkeztek, be kellett szúrni őket a megfelelő helyre. A **`ListViewBase`** az **`ISupportIncrementalLoading`** interfész segítségével képes jelezni az adatforrás számára, hogy újabb adatokat kell betölteni, egyfajta folytatólagos lapozás módszerével. Ehhez csupán annyit kell tennünk, hogy az adatforrásnak implementálnia kell az **`IsupportIncrementalLoading`** interfészt. Az interfész két tagot definiál:

- **`HasMoreItems`** – Boolean tulajdonság, amely azt jelzi, hogy van-e még adat az adatforrásban, lehet-e továbblapozni
- **`LoadMoreItemsAsync`** – Aszinkron metódus a további elemek letöltésére

Az alábbi példakód az interfész lehetséges megvalósítását demonstrálja:

```
public class NewsDataSource : ObservableCollection<News>, ISupportIncrementalLoading
{
    private int index = 0;

    public bool HasMoreItems
    {
        get { return true; }
    }

    public Windows.Foundation.IAsyncOperation<LoadMoreItemsResult> LoadMoreItemsAsync(uint count)
    {
        index++;

        return Task.Run(() =>
        {
            var coreDispatcher = Window.Current.Dispatcher;
            coreDispatcher.RunAsync(CoreDispatcherPriority.Normal, () =>
            {
                for (var i = 0; i < count; i++)
                {
```

```
                    this.Add(new News
                            {
                                Author = "Test Author" + index.ToString() + i,
                                Date = DateTime.Now,
                                Title = "Test Title " + i,
                                PhotoUrl = "/Assets/NewYork.jpg",
                                NewsBody = "Test Body " + i
                            });
                }
            });

            return new LoadMoreItemsResult() { Count = count };
        }).AsAsyncOperation();
    }
}
```

Ez a kód nem teljesen életszerű, ugyanis a végtelenségig lapoz, illetve generál elemeket, így a **HasMoreItems** tulajdonság mindig **true** értékkel tér vissza, a **LoadMoreItemsAsync** metódus pedig mindig újabb és újabb elemeket tud betölteni. A metódus törzsében a betöltött elemeket azonnal hozzáadjuk a listához, visszatérési értékként pedig egy **LoadMoreItemsResult** típusba ágyazzuk azt az információt, hogy hány elemet sikerült ténylegesen letöltenünk.

Jelen verzióban a legegyszerűbb az **ObservableCollection** osztályból származtatni, így azonnal értékül rendelhetjük egy **ListViewBase** típusú vezérlő **ItemsSource** tulajdonságához. Sajnos a csoportosításról le kell mondanunk ebben az esetben, ugyanis a **CollectionViewSource** osztály **sealed**, így nem kiterjeszthető, jelenleg pedig nem támogatja ezt a funkciót.

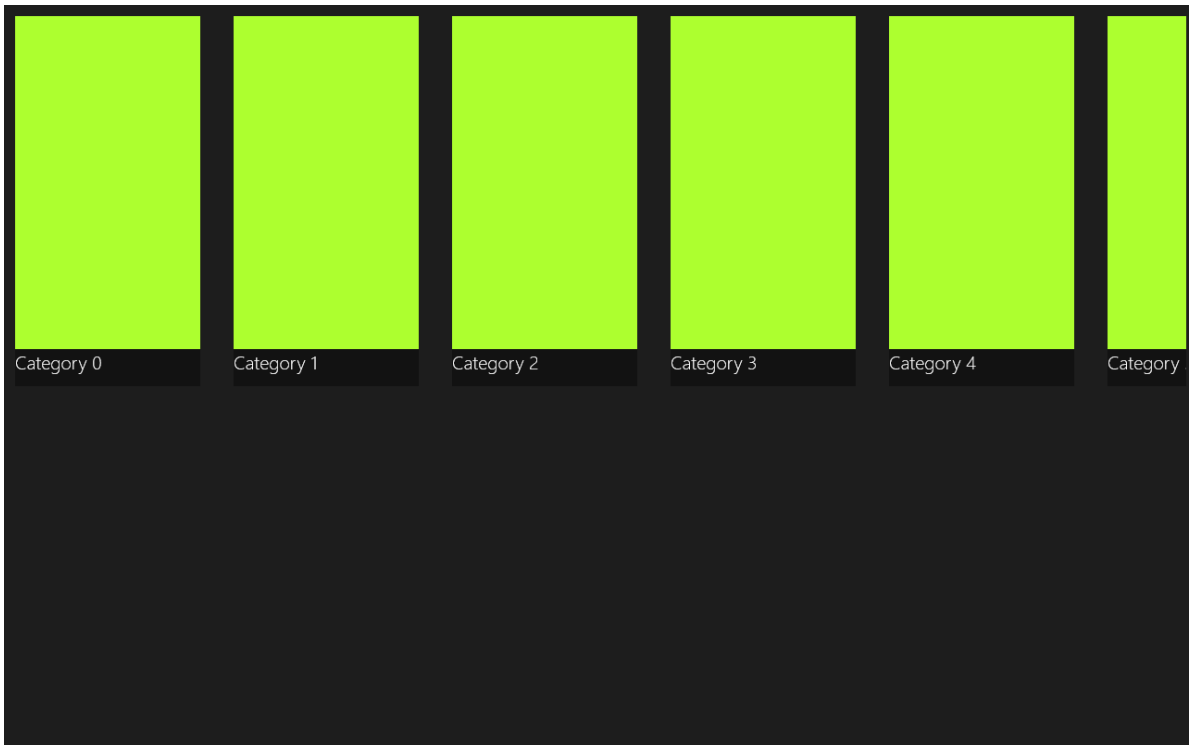
A ListView vezérlő

A **ListView** vezérlő szinte minden jellegzetességében ugyanaz, mint a **GridView** vezérlő. Akárcsak a **GridView**, ez is a **ListViewBase** vezérlőből származik, így a viselkedésük közös. A két vezérlő az alapértelmezett megjelenésük, illetve az elemek elrendezésében különbözik.

Az alábbi példakódban a kategóriákhoz rendelünk egy speciális megjelenítést:

```
<ListView ItemsSource="{Binding CategoriesView.CollectionGroups}" SelectionMode="None"
VerticalAlignment="Center">
    <ListView.ItemTemplate>
        <DataTemplate>
            <Grid Background="GreenYellow" Height="400" Width="200" Margin="5">
                <Grid.RowDefinitions>
                    <RowDefinition Height="*"/>
                    <RowDefinition Height="40"/>
                </Grid.RowDefinitions>

                <Rectangle Fill="#FF121212" Grid.Row="1"/>
                <TextBlock Text="{Binding Group.CategoryName}" FontSize="20"
                    FontFamily="Segoe UI" FontWeight="Light" Grid.Row="1"/>
            </Grid>
        </DataTemplate>
    </ListView.ItemTemplate>
    <ListView.ItemsPanel>
        <ItemsPanelTemplate>
            <VirtualizingStackPanel Orientation="Horizontal" />
        </ItemsPanelTemplate>
    </ListView.ItemsPanel>
</ListView>
```

Vegyük észre, hogy a fenti példakódban az **ItemsSource** tulajdonságot nem a **CategoriesView**-ra, hanem annak **CollectionGroups** tulajdonságára kötjük! Így a lista elemei valójában a kategóriák lesznek, nem a kapcsolódó hírek. Az eredményt a 7-6 ábra mutatja.



7-6 ábra: Listview saját ItemTemplate-tel

A SemanticZoom használata

A **SemanticZoom** kétségkívül a legizgalmasabb vezérlő a Windows 8 SDK-ban! Segítségével ugyanarra az adatra – bár ez nem követelmény – két különböző nézetet tudunk definiálni, a **ZoomedInView** és a **ZoomedOutView** nézeteket. A két nézet között pedig *pinch* és *zoom* gesztusokkal lehet váltani. Hasonlóan működik a Windows 8 Start oldal is. Próbáld ki bátran!

A **SemanticZoom** vezérlő használata egészen egyszerű. Az alábbi példakódon láthatjuk a vezérlő definíciójának vázlatát:

```
<SemanticZoom>
    <SemanticZoom.ZoomedInView>
        <!--Ide jön a részletes nézet-->
    </SemanticZoom.ZoomedInView>
    <SemanticZoom.ZoomedOutView>
        <!--//Ide jön az áttekintő -->
    </SemanticZoom.ZoomedOutView>
</SemanticZoom>
```

Készítsünk egy olyan vizualizációt, amely a részletes nézetben a kategóriák listáját mutatja, az áttekintő nézetben pedig a csoportosított híreket! Az alábbi példakód ezt demonstrálja:

```
<SemanticZoom>
    <SemanticZoom.ZoomedInView>
        <Controls:NewsGridView ItemsSource="{Binding CategoriesView}"
            SelectionMode="Multiple">
            <GridView.ItemsPanel>
                <ItemsPanelTemplate>
                    <StackPanel Orientation="Horizontal"/>
```

```
            </ItemsPanelTemplate>
        </GridView.ItemsPanel>
        <GridView.ItemTemplate>
            <DataTemplate>
                <Grid>
                    <Grid.RowDefinitions>
                        <RowDefinition Height="*"/>
                        <RowDefinition Height="40"/>
                    </Grid.RowDefinitions>

                    <Image Source="{Binding PhotoUrl}" Grid.RowSpan="2"
                        Stretch="UniformToFill"/>
                    <Rectangle Fill="#AA000000" Grid.Row="1"
                        HorizontalAlignment="Stretch" VerticalAlignment="Stretch"/>
                    <TextBlock Text="{Binding Title}" Grid.Row="1"
                        HorizontalAlignment="Left" VerticalAlignment="Center"
                        Margin="8,0,0,0" FontSize="20" FontFamily="Segoe UI"
                        FontWeight="Light"/>
                </Grid>
            </DataTemplate>
        </GridView.ItemTemplate>

        <GridView.GroupStyle>
            <GroupStyle HidesIfEmpty="True">
                <GroupStyle.HeaderTemplate>
                    <DataTemplate>
                        <TextBlock Text="{Binding CategoryName}"
                            Style="{StaticResource GroupHeaderTextStyle}"/>
                    </DataTemplate>
                </GroupStyle.HeaderTemplate>

                <GroupStyle.Panel>
                    <ItemsPanelTemplate>
                        <VariableSizedWrapGrid ItemHeight="160" ItemWidth="250"
                            MaximumRowsOrColumns="3" />
                    </ItemsPanelTemplate>
                </GroupStyle.Panel>

            </GroupStyle>
        </GridView.GroupStyle>
    </Controls:NewsGridView>
</SemanticZoom.ZoomedInView>
<SemanticZoom.ZoomedOutView>
    <ListView ItemsSource="{Binding CategoriesView.CollectionGroups}"
        SelectionMode="None" VerticalAlignment="Center">
        <ListView.ItemTemplate>
            <DataTemplate>
                <Grid Background="GreenYellow" Height="400" Width="200" Margin="5">
                    <Grid.RowDefinitions>
                        <RowDefinition Height="*"/>
                        <RowDefinition Height="40"/>
                    </Grid.RowDefinitions>

                    <Rectangle Fill="#FF121212" Grid.Row="1"/>
                    <TextBlock Text="{Binding Group.CategoryName}" FontSize="20"
                        FontFamily="Segoe UI" FontWeight="Light" Grid.Row="1"/>
                </Grid>
            </DataTemplate>
        </ListView.ItemTemplate>
        <ListView.ItemsPanel>
            <ItemsPanelTemplate>
                <VirtualizingStackPanel Orientation="Horizontal" />
            </ItemsPanelTemplate>
        </ListView.ItemsPanel>
    </ListView>
```

```
    </SemanticZoom.ZoomedOutView>
</SemanticZoom>
```

Ebben a példában a korábban megvalósított **GridView** komponensünket használtuk részletező nézetben, a **ListView**-t pedig áttekintő nézetben. Mivel mindkettő ugyanazon az adatforráson dolgozik, így ha áttekintő nézetben kiválasztunk egy kategóriát, a **SemanticZoom** átváltáskor a **GridView**-t a kiválasztott kategóriához pozicionálja. **SemanticZoom** nézetként jelenleg csak a **GridView** és a **ListView** vezérlők használhatók, persze saját komponenseket is implementálhatunk, ehhez a megfelelő interfészeket kell megvalósítanunk.

Speciális listakezelés a FlipView vezérlővel

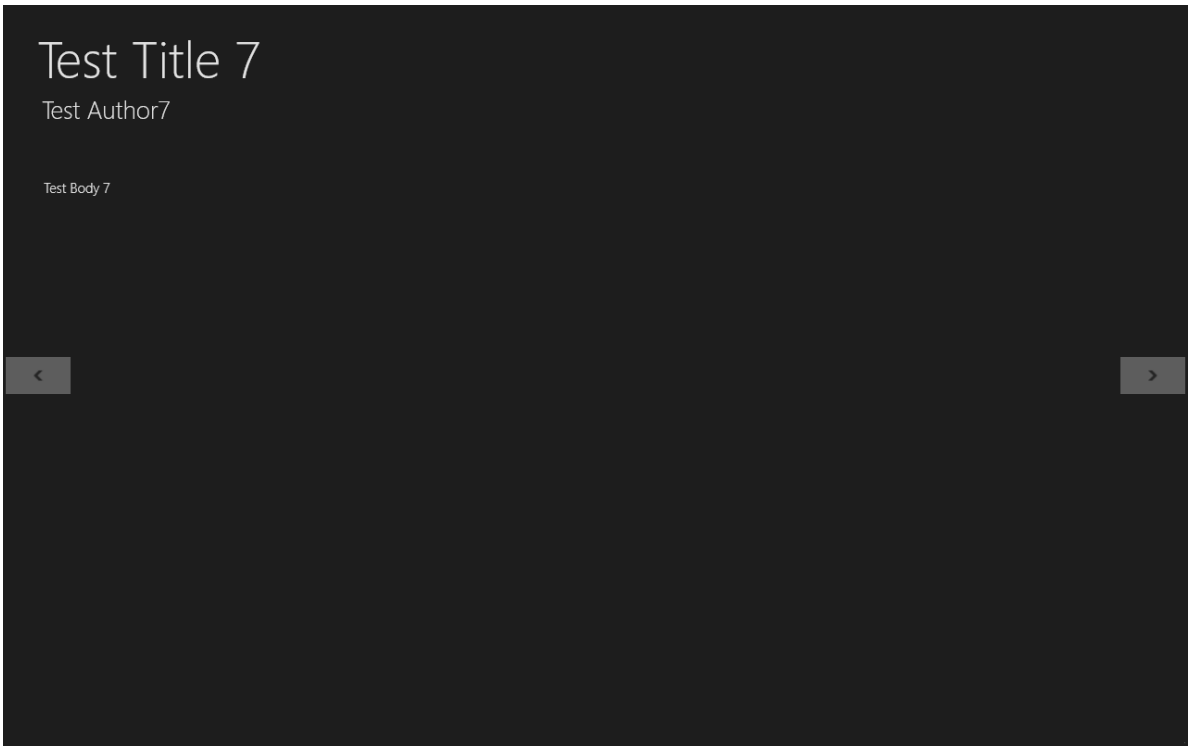
Habár a **FlipView** listás vezérlő, mindenképp kilóg azok sorából. Először is a **FlipView** nem **ListViewBase** leszármazott, és ez máris fontos különbség. Másodszor a **FlipView** nem elemek listáját jeleníti meg egyszerre, hanem a listából mindig csak egyet, az aktuálisan kiválasztott, pozicionált elemet.

A **FlipView** automatikusan biztosítja a navigációt az elemek között egérrel és érintésgesztusokkal egyaránt. A következő kódrészlet a **FlipView** használatát mutatja be:

```
<FlipView ItemsSource="{Binding News}"
    SelectedItem="{Binding SelectedItem, Mode=TwoWay}">
    <FlipView.ItemTemplate>
        <DataTemplate>
            <Grid>
                <TextBlock Text="{Binding Title}"
                    Style="{StaticResource PageHeaderTextStyle}"
                    VerticalAlignment="Top" Margin="36,36,0,0" />
                <TextBlock Text="{Binding Author}"
                    Style="{StaticResource PageSubheaderTextStyle}"
                    VerticalAlignment="Top" Margin="40,90,0,0"/>
                <TextBlock Text="{Binding NewsBody}"
                    Style="{StaticResource BodyTextStyle}"
                    VerticalAlignment="Top" Margin="42,180,0,0"/>
            </Grid>
        </DataTemplate>
    </FlipView.ItemTemplate>
</FlipView>
```

A fenti vezérlő mindig egy elemet jelenít meg, azon belül annak címét (**Title**), szerzőjét (**Author**), illetve szövegtörzsét (**NewsBody**).

A **SelectedItem** adatkötése sokat segít abban, hogy a **FlipView**-t a megfelelő elemre tudjuk pozicionálni. A 7-7 ábrán az eredmény látható.



7-7 ábra: FlipView vezérlő saját ItemTemplate-tel

Összegzés

Ebben a fejezetben megismerkedhettünk a Windows 8 sajátos, ám igen gyakran használt vezérlőivel. Ezek lelke a **`ListViewBase`** ősosztály, amely a felhasználói élmény szempontjából olyan kiemelkedő funkciókat biztosít, mint az adatok igényalapú letöltése, a csoportosítás, az adat- és elem-virtualizáció, a teljes testreszabhatóság és a kiválasztott elemek kezelése. A **`SemanticZoom`** és a **`FlipView`** vezérlők a felhasználói élményt és az adatok prezentációjának sokszínűségét tovább növelik. Használjuk bátran ezeket a vezérlőket, és építsünk minél inkább ezek képességeire, hogy felhasználóink igazi Windows 8 stílusú élményben részesülhessenek!

8. Windows 8 alkalmazásfejlesztés HTML5 és JavaScript segítségével

Ebben a fejezetben az alábbi témákat ismerheted meg:

- Hogyan használhatod fel a meglévő webfejlesztői ismereteidet Windows 8 alkalmazások fejlesztéséhez?
- HTML5/JavaScript-alapú Windows 8 alkalmazások működése
- Alapvető vezérlők, vezérlők testreszabása
- Fejlesztőeszközök használata

Bevezetés

A HTML5 a következő, jelentősen átdolgozott változata a HTML-nek (Hypertext Markup Language), a web fő jelölőnyelvének. Manapság azonban HTML5 alatt már nemcsak a leíró nyelvnek a legújabb változatát értjük, hanem egy komplett platformot, ami az alábbi komponensekből áll össze:

- HTML5
- CSS3
- JavaScript

A CSS a webes fejlesztés egyik alapvető komponense, hiszen segítségével megjelenést és külalakot adhatunk weboldalainknak. Két fő alkotóeleme van: azok a szabályok, amelyeknek alapján megkereshetjük a dokumentumaink bizonyos elemeit (CSS szelektorok), valamint azok a stílusok, amelyeket a megtalált elemekre alkalmazhatunk. A CSS3 egyik legnagyobb előnye az, hogy modulokból áll, és ezeket a modulokat egymástól függetlenül fejlesztik. Ilyen modul például a *Background & Borders*, a *Colors*, a *Selectors* és még számos egyéb. A modularizált felépítésének köszönhetően az egyes újításokat csomagban is meg lehet valósítani, így nem kell egyszerre implementálni a teljes szabványt. Ennek köszönhetően a funkciók hamarabb eljuthatnak a felhasználókhoz.

A JavaScript egy dinamikus, gyengén típusos szkriptnyelv. Először 1997–99 között szabványosította az ECMA „ECMAScript" néven, a könyv írásának pillanatában az aktuális verzió az 5.1. A JavaScript legérdekesebb újdonságai közé tartozik az **`Object`** konstruktor kibővítése, amelynek segítségével könnyedén tudunk saját objektumokat létrehozni és másolni, és az objektumaink tulajdonságait is egyszerűbben kezelhetjük az új *getter* és *setter* funkciókkal.

Út a HTML5-ig

Korábban a különböző plugin függőségek miatt az emberek nem ugyanazt a felhasználói élményt kapták az egyre inkább elterjedtebb hordozható eszközökön (táblagép, mobiltelefon). A HTML5 létrehozásánál az a cél lebegett az alkotók szeme előtt, hogy egy új RIA (Rich Internet Application) platformot hozzanak létre, mely mindenféle plugin és egyéb komponensek használata nélkül működik, hasonló élményt biztosítva a felhasználónak asztali számítógépeken, táblagépeken vagy akár mobiltelefonon. A HTML ökoszisztéma ma reneszánszát éli, naponta jelennek meg új keretrendszerek, melyekkel gyorsan és hatékonyan tudunk webes alkalmazásokat vagy akár keresztplatformos, Android, iOS és Windows Phone rendszereken egyaránt futtatható, közös kódbázissal rendelkező mobil alkalmazásokat készíteni.

A HTML5 elődjéhez képest nagyon sok újítást vonultat fel, ilyen a beépített *GeoLocation API*, amely a helymeghatározást segíti, valamint az egyik legnagyobb újítás, a *Canvas*, amelynek segítségével dinamikusan, programozható módon jeleníthetünk meg kétdimenziós grafikákat, alakzatokat. A Canvas az összetettebb animációk programozására is használható, így könnyedén készíthetünk vele játékokat, videólejátszókat, illetve reklámokat. A HTML5 logóját a 8-1 ábrán láthatod.

8-1 ábra: A HTML5 logója

Egy weboldalt leíró alap HTML struktúra így néz ki:

```
<!DOCTYPE html>
<html lang="hu">
    <head>
    </head>
    <body>
    </body>
</html>
```

A kezdőtag után jön a **`html`** nyitó és záró tag. Ezek közé kerül minden, kivéve a külső **`script`**eket. A háttér információk a **`head`** tagok közé kerülnek (pl. **`meta`** és **`script`** tagok), majd a **`body`** tagok között kap helyet az oldal és néhány **`script`**.

A HTML5/JavaScript szerepe a Windows 8 fejlesztői platformon

A nagyvállalatok közül a Microsoft is teljes mellszélességgel beállt a HTML5-öt támogatók lelkes sorába, és teljesen egyenrangú fejlesztési platformként beemelte a meglévő nyelvek mellé a natív alkalmazásfejlesztés lehetőségét a HTML5/JavaScript meglévő eszköztárainak a segítségével.

Egy HTML5-öt használó alkalmazás jóval több, mint egy weboldal, így egészen más szempontokat kell figyelembe venni a tervezésénél. Az alkalmazásoknak nincs kerete, kihasználhatják a teljes képernyőt. A hangsúly a tartalmon van, a tartalom adja a dizájn nagy részét. Nincsenek háromdimenziós vagy épp tükröződő üveghatást keltő effektek, az alkalmazások „digitálisan autentikusak”.

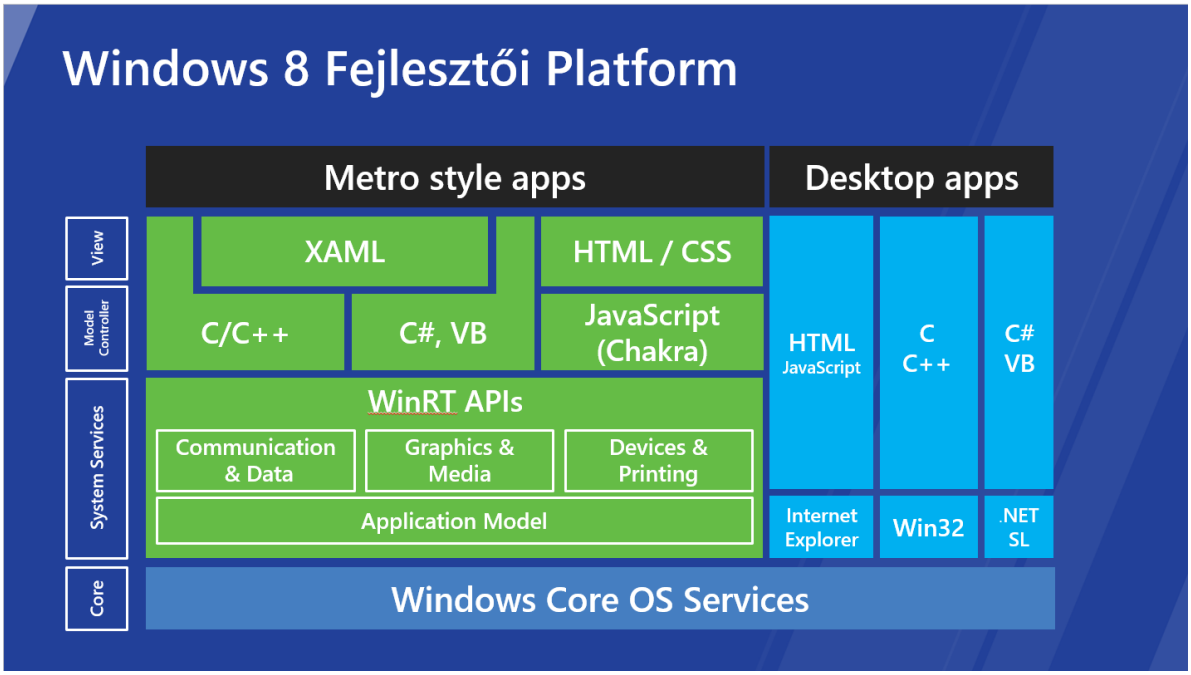
> *Fontos megjegyezni, hogy manapság divatos úgy hirdetni a HTML5-öt, amivel egyszerre lehet mobil, web, és desktop alkalmazásokat fejleszteni, azonban a Windows 8 esetében ez nincs. A Microsoft sok újítást eszközölt a HTML és CSS szabványokon, így hiába íródik egy Windows 8 stílusú alkalmazás HTML5/JavaScript eszközökkel, az nem képes sem weboldalként, sem pedig mobil alkalmazásként működni!*

A Microsoft elkötelezett a HTML5 szabványosításának irányában, nagyon sok bővítéssel látta el a szabványt, amit ki kellett egészíteni ahhoz, hogy használható legyen érintőképernyős eszközökön (érintési gesztusok kezelése), illetve – Microsoft specifikus CSS attribútumokat adtak hozzá (`-ms-grid`, `-ms-flexbox`).

> *A HTML5 szabvány a könyv írásának pillanatában még nem készült el, a szakértők szerint 2014-re várható annak véglegesítése.*

A Windows 8 alkalmazások működése

Ahogy a 8-2 ábra mutatja, a megújult fejlesztői platformon pontosan ugyanazokat a Windows Runtime API-kat és szolgáltatásokat érheti el egy fejlesztő, aki HTML5/JavaScript eszközök segítségével fejleszt alkalmazást, mint aki C# vagy C++ nyelven teszi ugyanezt. Természetesen a „régi" Windows alkalmazások is futtathatóak lesznek.

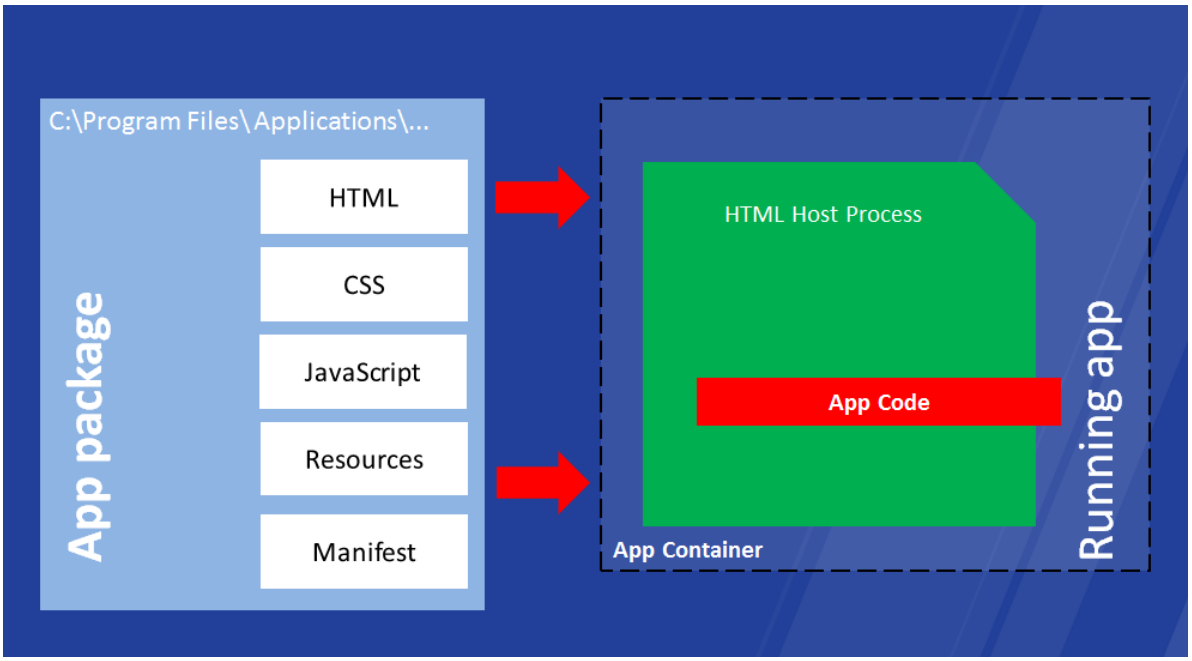


8-2 ábra: A Windows 8 fejlesztői platform

App Container

Az alkalmazásaink teljes izoláltságban élnek, saját, zárt tárterülettel rendelkeznek, így a saját területükön kívül nem látnak semmit a gépünkből, és ezért nem áll fenn annak a veszélye, hogy valaki egy kártékony kóddal az alkalmazásunk vagy akár a teljes rendszer működését ellehetetlenítse.

Az *App Package* tartalmazza az alkalmazáshoz szükséges összes erőforrást, a HTML/JavaScript/CSS fájlokat, a szükséges képeket, valamint a *manifest* fájlt. Az alkalmazás futás közben egy *App Container* nevezetű zárt környezetben fut. Biztonsági megfontolásból néhány elérhető szolgáltatás hozzáférhetőségét előre be kell állítani, és első használatnál a rendszer mindig megkérdezi a felhasználót, hogy engedélyezi-e a szolgáltatás használatát (pl. webkamera bekapcsolása, helymeghatározás). Az App Container felépítése a 8-3 ábrán látható.



8-3 ábra: Az App Container felépítése

Szerencsére azért van egy nagyon hatékony eszközünk arra, hogy ennek ellenére szinte tetszőleges részéhez hozzáférjünk a fájlrendszernek, fájlokat tárolhassunk, mappákat nyithassunk meg, és menthessünk el. Rendelkezésünkre áll a ***`FileOpenPicker`*** *osztály, mely a* ***`Windows.Storage.Pickers`*** *névtérben található. A fájlok elmentésére a* ***`FileSavePicker`*** *szolgál.*

Local context ↔ web context

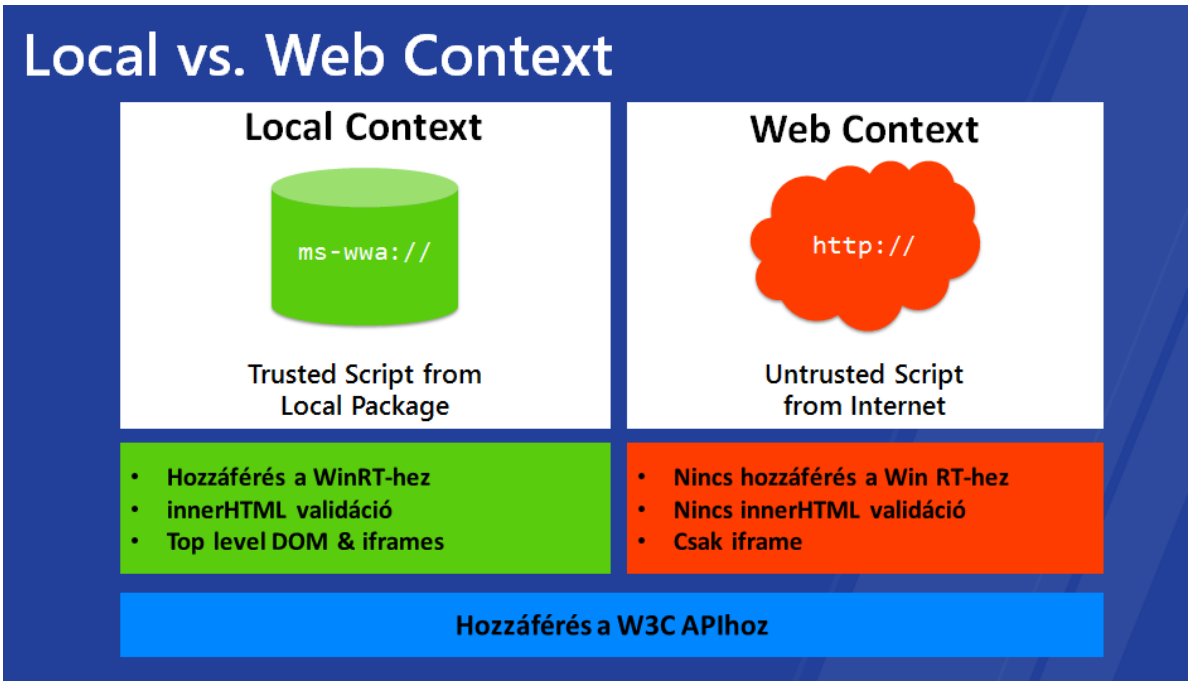
Mivel az alkalmazásunk menüpontjai és oldalai HTML oldalak, és azokban internetről származó kódot is használhatunk, így megkülönböztetjük a *local context*et (helyi környezet) és *web context*et (webes környezet). Ha az alkalmazásunk saját oldalait használjuk, akkor az local contextben fog futni, azonban ha kívülről beágyazunk egy weboldalról származó elemet, akkor annak biztonsági okok miatt csak korlátozott hozzáférése lesz a rendszerünkhöz. Hasonló megfontolások miatt külső internetről származó tartalmat csakis **`iframe`** elembe ágyazhatunk bele. A főbb különbségek a 8-4 ábrán láthatók, illetve azokat a 8-1 táblázat is összefoglalja.

Nézzünk erre egy konkrét példát! Ha az alkalmazásunkban szeretnénk megnyitni egy weboldalt, akkor használhatjuk az alábbi kódot, mely automatikusan elindítja a böngészőt, és abban fogja az oldalt megnyitni:

```
<p><a href="http://www.devportal.hu">Devportal</a></p>
```

Ha azonban azt szeretnénk, hogy a weboldal mindenképpen a saját alkalmazásunkon belül nyíljon meg, akkor az alábbi kódot használva egy egyszerű **`iframe`** beágyazással ezt megtehetjük:

```
<p><a href="http://www.devportal.hu" target="targetFrame" >Devportal</a></p>

<iframe name="targetFrame">
</iframe>
```



8-4 ábra: Local context ↔web context

Táblázat 8-1: Local context és web context összehasonlítás

Hozzáférés	Local Context	Web Context
Windows Runtime	Igen	Nem
Windows Library for JavaScript	Igen	Igen, korlátozásokkal
JavaScript URI-k	Nem	Igen
Külső szkript referenciák (`<script src="http://*" />`)	Nem	Igen
`window.close`	Igen	Nem
Cross Domain XHR	Igen	Nem

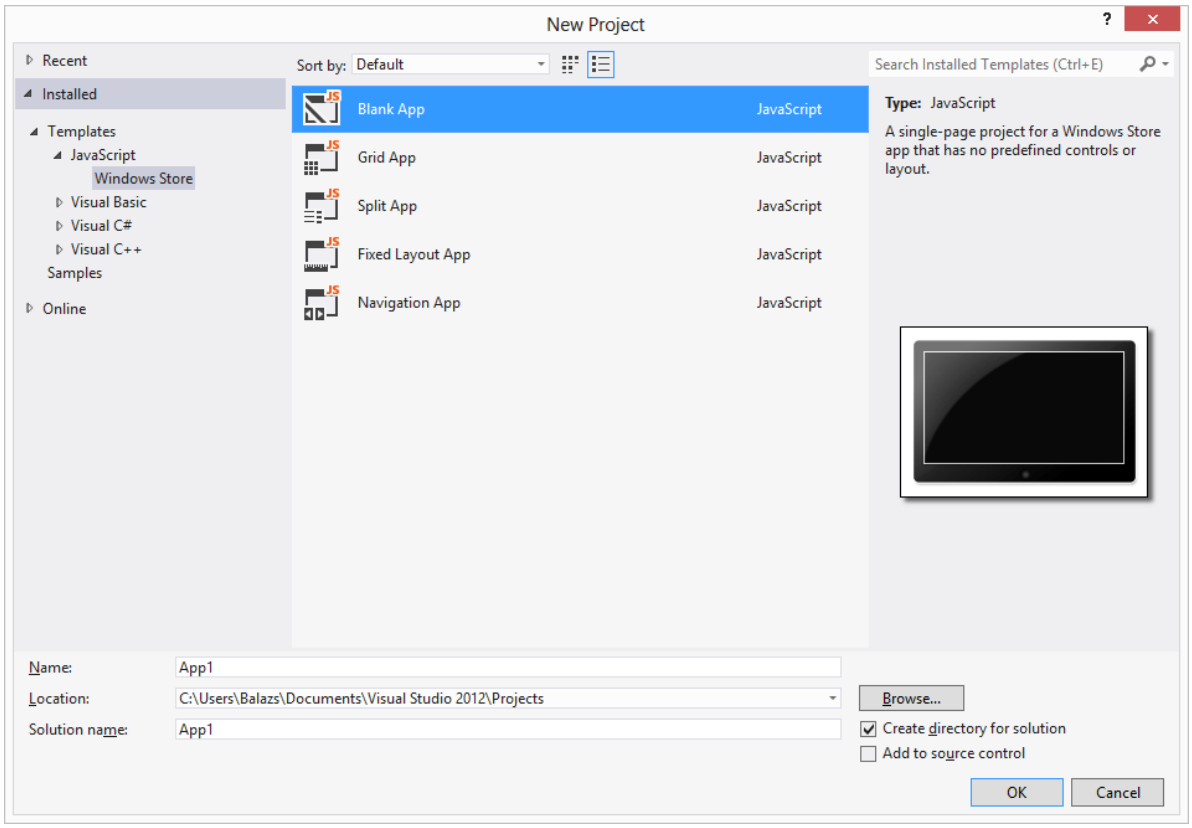
Ismerkedés a fejlesztőeszközökkel

Az alkalmazások fejlesztéséhez a Visual Studio 2012-t használhatjuk, mely webes szempontból jelentősen megerősödött: immáron teljes körű IntelliSense támogatás érhető el a HTML5/CSS3/JavaScript hármas számára is, valamint beépített projektsablonokat biztosít a fejlesztők számára a leggyakoribb projekttípusokhoz.

„Hello World”

A fejlesztőeszközökkel való ismerkedést elősegítendő készítsd el a már klasszikus, kicsit kibővített „Hello World” programot!

Indítsd el a Visual Studio 2012-t, és hozz létre egy új projektet a File | New |Project paranccsal! A New Project dialógusban válaszd a JavaScript projektek közül a Windows Store kategóriában lévő Blank App sablont (8-5 ábra), majd kattints az OK gombra!

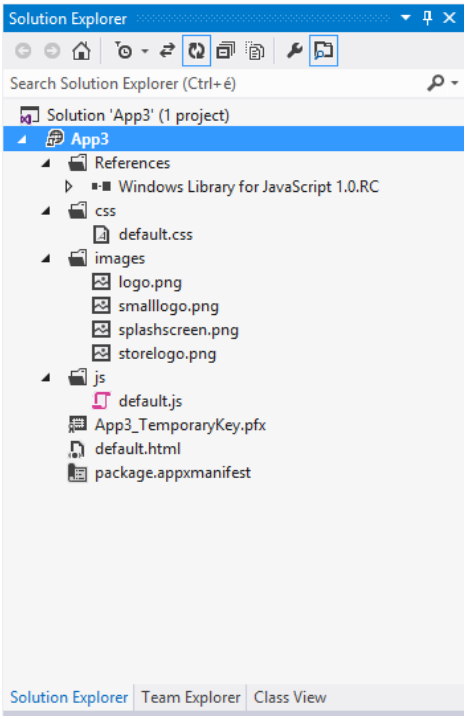


8-5 ábra: JavaScript project típusok

A program szerkezete

Miután létrehoztad a projektet, a képernyő jobb oldalán láthatod a Solution Explorer ablakot (8-6 ábra). Itt találhatók az alkalmazás működéséhez szükséges fájlok:

- **`default.html`:** Az alkalmazásunk kezdőoldala, ez töltődik be, miután elindult az alkalmazás. Referenciákat tartalmaz a **`default.js`**-ra, és az alkalmazás saját CSS fájljához.
- **`/js/default.js`:** JavaScript kód, amely leírja, hogyan viselkedjen az alkalmazás, miután elindult.
- **`/css/default.css`:** Az alkalmazásunk kinézetét leíró CSS fájl.
- **`/References/Windows Library for JavaScript`:** Beépített stílusok, vezérlők és egyéb eszközök gyűjteménye, amelyek hatékony alkalmazások megírásában segítenek.
- **`package.appmanifest`:** Ez a fájl írja le alkalmazásunk képességeit. Itt kell beállítanunk azt, ha hozzá szeretnénk férni az eszközünk valamelyik érzékelőjéhez a saját alkalmazásunkban – például a helymeghatározáshoz vagy a webkamerához.
- **`/images`:** Ebben a mappában találhatóak azok a képek, amelyek az alkalmazáshoz szükségesek. A két logó fájl jelenik majd meg az alkalmazás lapkáján (kis, illetve normál méret, illetve opcionálisan egy széles logót is hozzáadhatunk). A **`splashscreen.png`** az alkalmazás betöltődésekor megjelenő kép, a **`storelogo.png`** pedig a Windows Store-ban az alkalmazásunkhoz társított kép.



8-6 ábra: A projekt szerkezete a Solution Explorerben

A 8-1, a 8-2 és a 8-3 kódlisták a projekt **`default.html`**, **`default.js`**, illetve **`default.css`** fájljainak alapértelmezett – a projekt generálása utáni – tartalmát mutatják be.

8-1 kódlista: a `default.html` tartalma

```
<!DOCTYPE html>
<html>
<head>
     <meta charset="utf-8" />
     <title>App1</title>

     <!-- WinJS references -->
     <link href="//Microsoft.WinJS.1.0.RC/css/ui-dark.css" rel="stylesheet" />
     <script src="//Microsoft.WinJS.1.0.RC/js/base.js"></script>
     <script src="//Microsoft.WinJS.1.0.RC/js/ui.js"></script>

     <!–App1 references -->
     <link href="/css/default.css" rel="stylesheet" />
     <script src="/js/default.js"></script>
</head>
<body>
<p>
   Content goes here
</p>
</body>
</html>
```

8-2. kódlista: a `default.js` fájl tartalma

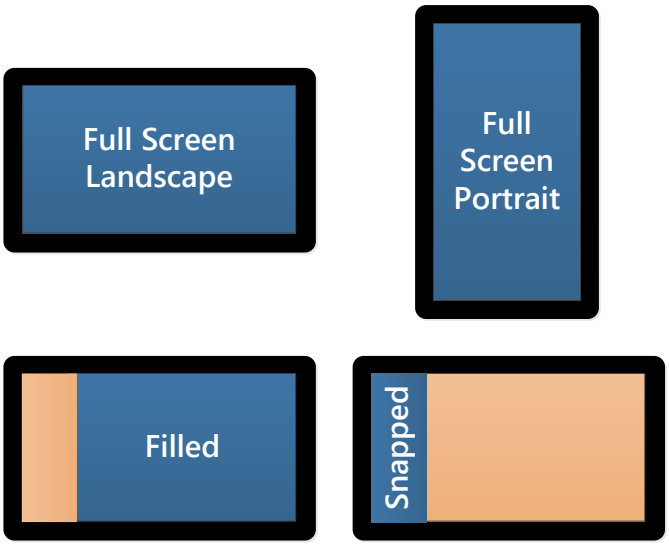
```
//For an introduction to the Blank template, see the following documentation:
// http://go.microsoft.com/fwlink/?LinkId=232509

(function () {
    "use strict";
     var app = WinJS.Application;
     var activation=Windows.ApplicationModel.Activation;
     WinJS.strictProcessing();
```

```
      app.onactivated = function (args) {
          if (args.detail.kind === activation.ActivationKind.launch) {
              if (args.detail.previousExecutionState !==
                  activation.ApplicationExecutionState.terminated) {
              // TODO: This application has been newly launched. Initialize
              // your application here.
           } else{
              // TODO: This application has been reactivated from suspension.
              // Restore application state here.
           }

           args.setPromise(WinJS.UI.processAll());
      }
};

app.oncheckpoint = function (args) {
   // TODO: This application is about to be suspended. Save any state
   // that needs to persist across suspensions here. You might use the
   // WinJS.Application.sessionState object, which is automatically
   // saved and restored across suspension. If you need to complete an
   // asynchronous operation before your application is suspended, call
   // args.setPromise().
};

app.start();

})();
```

8-3 kódlista: a `default.css` fájl tartalma

```
body {

}

@media screen and (-ms-view-state: fullscreen-landscape) {

}

@media screen and (-ms-view-state: filled) {

}

@media screen and (-ms-view-state: snapped) {

}

@media screen and (-ms-view-state: fullscreen-portrait) {

}
```

A Windows 8 egyik jelentős újdonsága, hogy egy alkalmazás több állapotot is felvehet a teljes képernyős módon kívül, és ez nagy kihívás elé állítja a fejlesztőket, akik igényes alkalmazásokat szeretnének készíteni. Lehetőség van arra, hogy két alkalmazás párhuzamosan fusson egymás mellett. Ilyenkor a kisebbik az úgynevezett "*Snapped*" állapotot, a nagyobbik pedig a "*Filled*" állapotot veszi fel (8-7 ábra). Mindezek mellett pedig nem szabad megfeledkeznünk az álló állapot lehetőségéről sem, így ezeket egy alapos fejlesztőnek kezelnie kell. Ehhez nyújt lehetőséget a CSS3-ban megjelent *Media Queries*, mely lehetővé teszi, hogy az egyes megjelenítendő méretekhez részben vagy teljesen eltérő stílusokat definiáljunk, és ezen keresztül különböző típusú eszközökön eltérő felhasználói élményt és vizuális megjelenést biztosítsunk az alkalmazásunkban. A Media Queries nagy előnye, hogy mindezt JavaScript használata nélkül, pusztán deklaratív módszerekkel teszi számunkra elérhetővé.

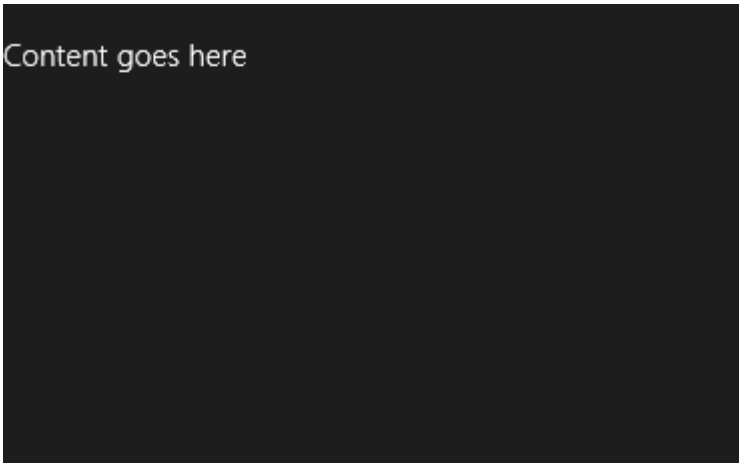


8-7 ábra: Alkalmazás lehetséges elhelyezkedései

*Mit jelent a "**`use strict`**" sor a **`default.js`** fájlban? A strict mód szigorúbb programozási stílust tesz lehetővé, „kidobálva" azokat a részeket a nyelvből, melyek csak problémákat okoztak. Ilyen megkötés például, hogy a változókat használat előtt deklarálni kell a **`var`** paranccsal, valamint a függvényeknek nem lehet több ugyanolyan nevű paramétere. Bővebben az alábbi linken olvashatsz erről: http://msdn.microsoft.com/en-us/library/br230269(v=vs.94).aspx*

A program módosítása

Miután már átnézted az alapvető fájlokat, futtasd le az alkalmazást az F5 billentyű lenyomásával (8-8 ábra)!



8-8 ábra: Az első futtatás

Mivel a program a jelenlegi állapotában nem csinál semmi hasznosat, ezért zárd be azt az Alt+F4-gyel! Kapcsolj át a Visual Studióra, majd módosítsd a **`default.html`** fájlt! Add hozzá a megfelelő üdvözlő szövegeket, illetve kérd be a felhasználó nevét! Helyezz el az oldalon egy beviteli mezőt, valamint egy „Hello" feliratú gombot! Az üdvözlés kiíratását tedd be egy külön **`div`** tagba, amint ezeknek a módosításoknak az eredményét a 8-4 kódlista is mutatja!

8-4 kódlista: a `default.html` módosítása:

```
<!DOCTYPE html>
<html>
<head>
     <meta charset="utf-8" />
     <title>App1</title>
```

```
    <!-- WinJS references -->
    <link href="//Microsoft.WinJS.1.0.RC/css/ui-dark.css" rel="stylesheet" />
    <script src="//Microsoft.WinJS.1.0.RC/js/base.js"></script>
    <script src="//Microsoft.WinJS.1.0.RC/js/ui.js"></script>

    <!—App1 references -->
    <link href="/css/default.css" rel="stylesheet" />
    <script src="/js/default.js"></script>
</head>
<body>
    <h1>Hello, world!</h1>
    <p>Hogy hívnak?</p>
    <input id="nameInput" type="text" />
    <button id="helloButton">"Hello"</button>
    <div id="greetingOutput"></div>
</body>

</html>
```

Ha most újra lefuttatod az alkalmazást, akkor a 8-9 ábrán látható eredményt kapod.

8-9 ábra: Megjelent a gomb és a beviteli mező

A gomb jelenleg még nem működik, a működésre bírásához meg kell írnunk az eseménykezelő függvényét JavaScriptben. Az eseménykezelő megkapja a felhasználó által beírt nevet a **`nameInput`** vezérlőből, és felhasználja azt a **`greetingOutput div`** elemben megjelenő kimenethez.

> *Felmerülhet a kérdés, hogy meg kell-e különböztetnünk a különféle inputokat attól függően, hogy az felhasználó egérrel kattintott-e rá, vagy az érintőképernyőn érintette-e meg a megfelelő gombot. Jó hír, hogy ez a teher lekerült a vállunkról, elég a **`click`** eseményt használnunk, mert az bármilyen input esetén működik.*

Az eseménykezelő létrehozásához nyisd meg a **`default.js`** fájlt, és az **`app.oncheckpoint`** eseménykezelő után, de még az **`app.start`** előtt hozd létre a **`buttonClickHandler`** nevű eseménykezelő függvényt! Hozz létre abban egy **`userName`** nevű változót, melynek add értékül a **`nameInput`** tartalmát, majd írd ki az üdvözlést a **`greetingOutput div`** elembe!

```
function buttonClickHandler(eventInfo) {
    var userName = document.getElementById("nameInput").value;
    var greetingString = "Hello, " + userName + "!";
    document.getElementById("greetingOutput").innerText = greetingString;
}
```

Most, hogy létrehoztad a megfelelő eseménykezelőt, már csak fel kell rá iratkoznod – ezt az **`addEventListener`** segítségével tudod megtenni. Az eseménykezelőre célszerű akkor feliratkozni, amikor az alkalmazásunk aktív állapotban van. Szerencsére a Visual Studio a **`default.js`** fájlban legenerálta számunkra az alapértelmezett életciklus kezelését, így nekünk nem kell ezzel foglalkozni. Adj a meglévő kódhoz egy **`helloButton`** nevű változót, amellyel aztán feliratkozhatsz a gomb eseményére:

```
app.onactivated = function (args) {
    if (args.detail.kind === activation.ActivationKind.launch) {
        if (args.detail.previousExecutionState !==
            activation.ApplicationExecutionState.terminated) {
            // TODO: This application has been newly launched. Initialize
            // your application here.
            } else {
                // TODO: This application has been reactivated from suspension.
                // Restore application state here.
            }
            args.setPromise(WinJS.UI.processAll());
            // Retrieve the button and register our event handler.
           var helloButton = document.getElementById("helloButton");
           helloButton.addEventListener("click", buttonClickHandler, false);
    }
}
```

Ha ezután elindítod az alkalmazást, és beírod a saját neved, majd a Hello gombra kattintasz, akkor alul megjelenik az üdvözlés, amint azt a 8-10 ábra is mutatja.

8-10 ábra: Működik a program!

Természetesen nem muszáj ragaszkodnunk az alapértelmezett sötét témához, a CSS fájl kis módosításával pillanatok alatt átválthatunk a világos témára! Ehhez nyisd meg a **`default.html`** fájlt, majd az alábbi sorban a **`ui-dark.css`**-t írd át **`ui-light.css`**-re.

```
<!-- WinJS references -->
<link href="//Microsoft.WinJS.1.0.RC/css/ui-light.css" rel="stylesheet" />
```

Ha most futtatod az alkalmazást, annak felülete már világos árnyalatú, amint azt a 8-11 ábra is mutatja.

8-11 ábra: A világos stílus használata

Alapvető vezérlők

A Windows 8-ban nagyon sok beépített vezérlő található, amelyek a kiváló felhasználói élményt nyújtó alkalmazások hatékony fejlesztésében segítenek bennünket. Ezek természetesen tökéletesen belesimulnak a Windows 8 modern dizájnjába, beépített animációkat tartalmaznak, és felkészítették őket az többérintéses vezérlésre is. Ezeket a WinJS-ben található vezérlőket HTML, CSS és JavaScript segítségével írták, így könnyedén integrálhatóak a többi DOM elemmel. Tipikus példák ezekre a **`DatePicker`**, a **`Rating`** és a **`ToggleSwitch`** vezérlők, melyek a 8-12 ábrán láthatók.

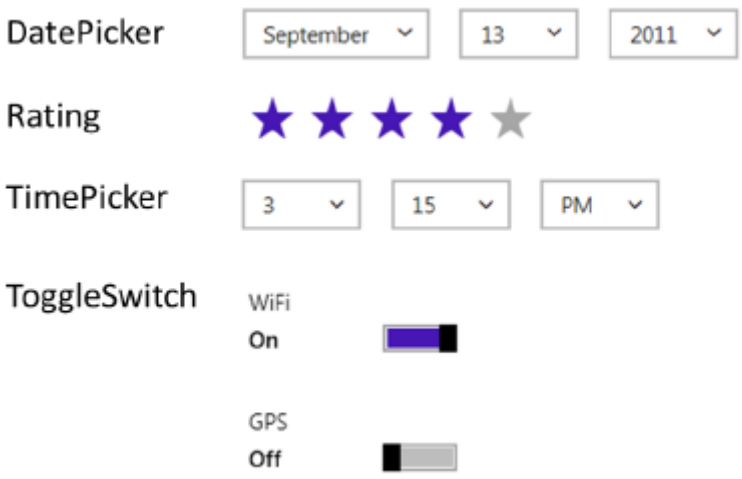
A vezérlőkről általánosságban elmondható, hogy a HTML jelölésben a **`data-win-control`** és a **`data-win-options`** attribútumokat csatoljuk a gyökér elemhez – amely tipikusan **`div`** vagy **`span`**.

A **`Rating`** vezérlő esetében ez így néz ki:

```
<div data-win-control="WinJS.UI.Rating"></div>
```

A vezérlő példányosítása pedig így történik:

```
var object = new WinJS.UI.Rating(element, options);
```



8-12 ábra: WinJS vezérlők működés közben

A fejezetnek nem célja bemutatni az összes vezérlőt, azonban az érdeklődők számára a Windows 8 HTML/JavaScript vezérlők teljes listája elérhető a http://msdn.microsoft.com/en-us/library/windows/apps/hh465453.aspx weboldalon.

Vezérlők testreszabása

A beépített vezérlők esetén természetesen felülírhatók a Windows Library for JavaScript stíluslapok, így igazán egyedi kinézetű vezérlőket is létrehozhatunk, amelyekkel még különlegesebb lehet az alkalmazásunk. Nézzünk erre egy egyszerűbb példát egy szövegmező segítségével! A CSS fájlba az alábbi stílusdefiníciót kell beírnunk:

```
input[type=text]
{
    width: 400px;
}.
```

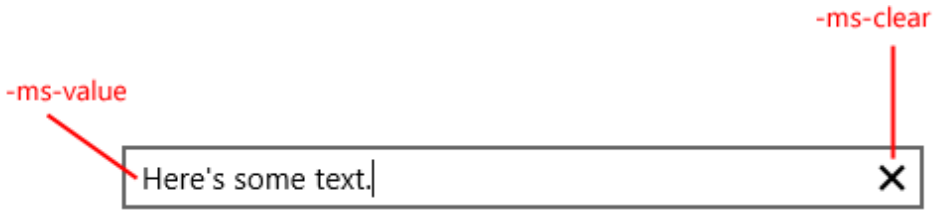
Ennek eredményét a 8-13 ábra mutatja be.

8-13 ábra: Egy szövegmező 400px szélességgel

A legtöbb vezérlő több komponensre is szétbontható. A 8-13 ábrán látható, hogy a szövegmezőnek két része van, a benne található szöveg (*value*) és a törlés gomb (*Clear button*), amint azt a 8-14 ábra mutatja.

8-14 ábra: Komponensek

A Windows 8 alkalmazások esetében ezeket a komponenseket ellátták egyedi névvel a CSS „álelemek” (CSS Pseudo Elements) használatával, melynek segítségével az adott szelektorhoz speciális effektusokat rendelhetünk (8-15 ábra).



8-15 ábra: CSS pszeudóelemek

Ha szeretnénk a beviteli mező törlésére szolgáló X gomb köré egy narancssárga keretet rakni, akkor az **`-ms-clear`** használatával ezt az alábbiak szerint tudjuk megtenni a CSS kódban (az eredmény a 8-16 ábrán látható):

```
input[type=text]::-ms-clear
{
    border: 2px solid orange
}
```

8-16 ábra: Az `-ms-clear` használatának eredménye

Az egyéni stílusok beállítása az alábbi szintaxis szerint történik:

```
element selector::part name { /* Ide jön a stílus leírása */ }
```

Érzékelők használata

Ahhoz, hogy egy gép érzékelhesse a környezetét – a fizikai valóságot – , érzékelőkre van szüksége. A gép szenzoraiból érkező adatokat kihasználva az alkalmazásunk hatékonyan alkalmazkodhat a körülményekhez. Készíthetünk például magas kontrasztú témát, mely napfényben is jól látható, vagy automatikusan átválthatunk egy sötét témára, ha a gép sötét környezetben van.

Az érzékelők a hordozható gépeknél lehetnek érdekesek, mivel a legtöbb ilyen eszköznél az alapfelszereltség részét képezi a fényérzékelő, a mozgásérzékelők és a GPS vevőegység is.

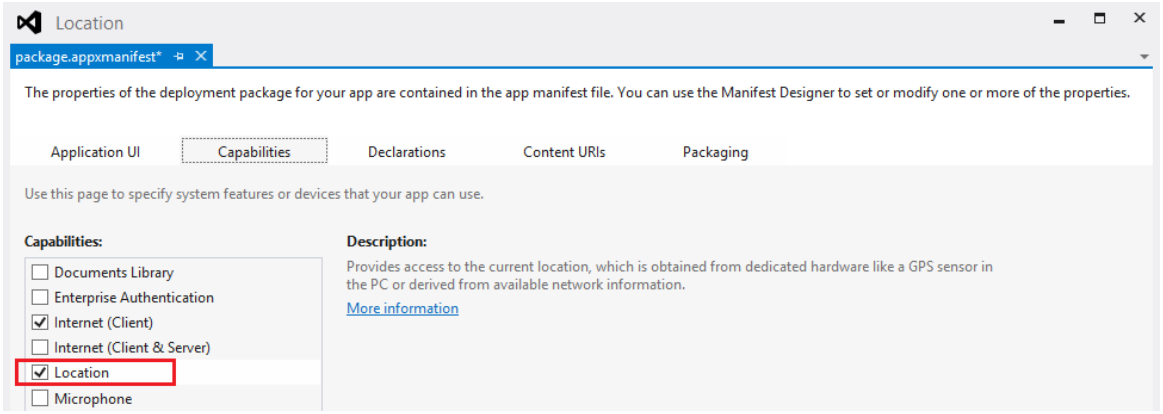
Helymeghatározás

Tekintsük át egy konkrét példán keresztül az egyik leggyakrabban használt eszközt, a helymeghatározást! A Windows 8 helymeghatározása több elemből épül fel. A rendszer elsődlegesen a már bekapcsolt hálózati antennáit próbálja meg felhasználni a helyzetének meghatározására úgy, hogy megpróbálja a körülötte lévő *hotspot*ok alapján bemérni a saját pozícióját. Ez az esetek jelentős részében elfogadható pontosságot biztosít amellett, hogy kevesebb energiát és kevesebb időt igényel, mint a GPS használata. Ráadásul városi környezetben potenciálisan nagyobb lefedettséget tud biztosítani, mint a GPS, ami – valljuk be – épületen belül nem megfelelő hatékonysággal működik. Ettől a szenzortól visszakapjuk pozíciónk szélességi és hosszúsági koordinátáit, tengerszint feletti magasságunkat, mozgásunk irányát, valamint a sebességet. Emellett az API össze van kötve a Bing Maps szolgáltatással is, így lehetőségünk van például koordinátából címet és címből koordinátát képezni, vagy éppen útvonalat tervezni két pont között. Szerencsére azonban a helymeghatározás eszközének kiválasztása teljesen automatikusan történik, így fejlesztőként egyszerűen használható, amint azt ki is próbálhatod.

Hozz létre ismét egy új projectet, szintén a Blank App sablon segítségével! Állítsd be, hogy az alkalmazás hozzáférhessen a helymeghatározáshoz: ehhez kattints duplán a **`package.appxmanifest`** fájlra, majd ott a **`Capabilities`** fülön jelöld ki a Location mezőt (8-17 ábra)! Ezután nyisd meg a **`default.html`** fájlt, és írd bele az alábbi kódot:

```
<!DOCTYPE html>
<html>
<head>
<title>Windows Geolocation Example</title>
<link rel="stylesheet" href="/winjs/css/ui-dark.css" />
<script type = "text/javascript" >

var loc = null;
function getloc() {
    if (loc == null) {
        loc = new Windows.Devices.Geolocation.Geolocator();
    }
    if (loc != null) {
        loc.getGeopositionAsync().then(getPositionHandler, errorHandler);
    }
}

function getPositionHandler(pos) {
    document.getElementById('latitude').innerHTML = pos.coordinate.latitude;
    document.getElementById('longitude').innerHTML = pos.coordinate.longitude;
    document.getElementById('accuracy').innerHTML = pos.coordinate.accuracy;
    document.getElementById('geolocatorStatus').innerHTML=
        getStatusString(loc.locationStatus);
}
```

```
function errorHandler(e) {
    document.getElementById('errormsg').innerHTML = e.message
    document.getElementById('geolocatorStatus').innerHTML =
        getStatusString(loc.locationStatus);
}

function getStatusString(locStatus) {
    switch (locStatus) {
        case Windows.Devices.Geolocation.PositionStatus.ready:
            return "Location is available.";
            break;
        case Windows.Devices.Geolocation.PositionStatus.initializing:
            return"A GPS device is still initializing.";
            break;
        case Windows.Devices.Geolocation.PositionStatus.noData:
            return "Data from location services is currently unavailable.";
            break;
        case Windows.Devices.Geolocation.PositionStatus.disabled:
            return "Your location is currently turned off.";
            break;
        case Windows.Devices.Geolocation.PositionStatus.notInitialized:
             return "The app has not requested location data.";
             break;
        case Windows.Devices.Geolocation.PositionStatus.notAvailable:
             return "You do not have the required location services on your system.";
             break;
        default:
             break;
   }
}

</script>
</head>
<body>
<p>Click "Get Location" to get geolocation data.<br />
<input type="button" onclick="getloc()" value="Get Location" /><br />

Latitude: <span id="latitude"></span><br />
Longitude: <span id="longitude"></span><br />
Accuracy (in meters): <span id="accuracy"></span><br /><br />
Location Status:<span id="geolocatorStatus"></span><br />
Error Message: <span id="errormsg"></span><br />

</p>
</body>
</html>
```



8-17 ábra: A Location kijelölése a `package.appxmanifest` szerkesztőjében

Az osztály, ami a helymeghatározásban segít, a **Windows.Devices.Geolocation** névtérben található **Geolocator.** Egy új példányt a **new** kulcsszóval példányosíthatunk belőle. Két fontos információt kérdezhetünk le belőle, az egyik az érzékelő állapotára utal, a másik pedig az aktuális koordinátánkra.

Az előbbit lekérdezhetjük szinkron módon a **LocationStatus** tulajdonságon keresztül vagy feliratkozhatunk a **StatusChanged** eseményre. Mindkét esetben egy **PositionStatus** típusú felsorolás egy értékét fogjuk visszakapni, amely az alábbiak valamelyike lehet:

- **Disabled**, ha a felhasználó nem engedélyezte az alkalmazásnak a helymeghatározást.
- **Initializing**, ha a mérés forrása a GPS, de még nem "lát" elegendő műholdat.
- **NoData,** ha a mérés forrása nem GPS, és még nem tudott használható koordinátákkal előállni.
- **NotAvailable,** ha valami okból kifolyólag semmilyen módon nem lenne képes az eszköz behatárolni magát, pl. nincs hálózat.
- **NotInitialized,** ha még meg sem próbáltunk semmiféle adatot lekérdezni – normál esetben ez az alapállapot.
- **Ready,** ha van rendelkezésre álló adat.

Az aktuális pozíció koordinátáit manuálisan a **GetGeopositionAsync()** segítségével kérdezhetjük le, vagy akár eseményvezérelten, a **PositionChanged** eseményre feliratkozva.

Mindkét esetben egy **GeoPosition** típusú objektumot kapunk, mely egy **Coordinate** tulajdonságban a szélességi és hosszúsági fokokat (**Latitude, Longitude**) tartalmazza, illetve a magasságot (**Altitude**), a sebességet (**Speed**), a haladási irányt (**Heading**) és a **TimeStamp** értéket – ez utóbbi a mérés időpontját adja meg.

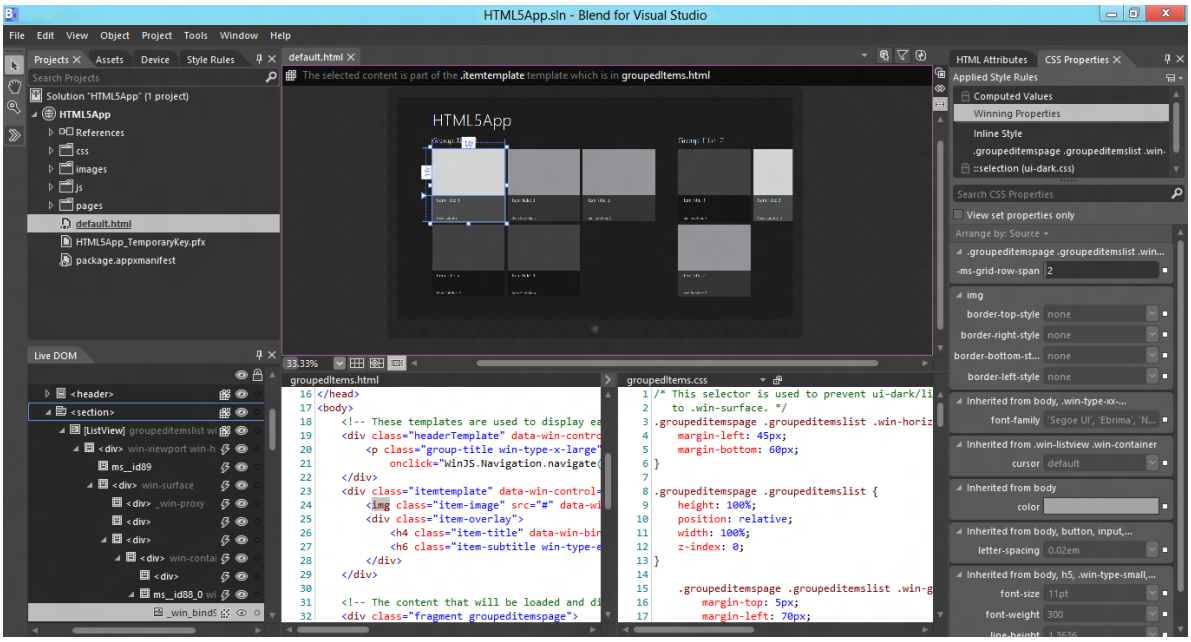
A **GeoPosition** rendelkezik még egy **CivicAddress** tulajdonsággal is, mely segítségével lehetőségünk van arra, hogy hagyományos címként adja vissza a pozíciónkat (**Country, State, City, PostalCode** jellemzőkkel).

Blend for HTML

Egy nagyobb alkalmazás felületének elkészítéséhez hatalmas mennyiségű HTML és CSS kódot kell írnunk. Az ilyen jellegű fejlesztés szöveges szerkesztővel nem hatékony, lassú a munkafolyamat – éppen a vizualitás hiánya miatt. Ezen segít a már régóta létező, korábban Expression Blend nevet viselő eszköz, melynek immár létezik HTML-es alkalmazásokhoz készített verziója. A Blenddel csak Windows 8 stílusú alkalmazásokat hozhatunk létre, weboldalakat sajnos nem. A 8-18 ábrán egy jellemző képernyőt láthatunk egy Windows 8 stílusú JavaScript alkalmazás vizuális szerkesztéséről.

A Blend segítségével könnyedén meg tudjuk tervezni az alkalmazásunk kinézetét, mert ez az eszköz a dizájn koncepciójára fókuszál, nem pedig a szövegszerkesztésre és a szintaxisra! Könnyedén létrehozhatunk animációkat és grafikákat, valamint jól használható, informatív felhasználói felületeket. A kódolás és dizájn elkülönítését is ragyogóan támogatja, mivel a dizájner nyugodtan dolgozhat a fejlesztőtől különállóan az alkalmazás felületén, amíg a fejlesztő a Visual Studio jól megszokott környezetében írja a mögöttes JavaScript kódot. Természetesen Blendben is lehet kódot írni, azonban a kényelem miatt érdemesebb erre a Visual Studiót használni.

Az eszköz legfontosabb újdonsága az úgynevezett interaktív mód. Egy HTML5 alkalmazás fejlesztése annyiban bonyolultabb egy weboldalnál, hogy magasabb fokú interakciót vár a felhasználótól. A fejlesztők számára nagy kihívás, hogy az alkalmazások fejlesztése közben olyan állapotokkal kell dolgozniuk, amelyek a felhasználók által kiváltott interakciók eredményeképpen állnak elő. Hagyományos esetben inaktív módon megtervezzük az alkalmazás működését, majd lefordítjuk azt, és minden egyes apró változtatásnál a módosítás, fordítás és ellenőrzés lépéseit ismételjük. Az interaktív mód ezt igyekszik kiküszöbölni: a fejlesztőnek lehetősége van az alkalmazást a tervezőfelületről futtatni, és bármelyik kívánt állapotban megállíthatja azt alkalmazást. Az adott ponton elvégezheti a szükséges módosítást, majd annak végrehajtása után a megállított állapotból folytathatja tovább a munkát.



8-18 ábra: A Blend felülete

Összegzés

A Microsoft teljesen új lehetőségként a Windows 8 platform részévé tette a HTML5 alapú natív alkalmazásfejlesztést, amit meglehetősen sok fejlesztőeszközzel és dokumentációval igyekszik támogatni. A Windows Store alkalmazásainak jelentős része – talán pont ezért nem meglepő módon – HTML5/JavaScript alapokon működik, ezzel is jelezve, hogy ez nem egy „megtűrt” fejlesztési irány, hanem teljes mértékben támogatott megoldás. Azok a szakemberek, akik korábban webes fejlesztőként foglalkoztak már a HTML-lel és a kapcsolódó technológiákkal, viszonylag rövid betanulási idő után natív alkalmazásfejlesztővé válhatnak, és a Windows Store-ba feltöltött alkalmazásaikkal részesei lehetnek a Windows 8 várható sikereinek.

9. Alkalmazások integrálása a Windows 8 szolgáltatásaival

Ebben a fejezetben az alábbi témákat ismerheted meg:

- Mit jelent az új életciklus-modell és hogyan működik?
- Milyen módon tudunk adatokat elérni és tárolni alkalmazásainkban?
- Hogyan tudunk saját Lapkákat készíteni és egyedivé varázsolni őket?
- Hogyan tudjuk az új hardverkörnyezet előnyeit kihasználni?

Bár a fejezet címe általánosítva próbál rámutatni a következő oldalak tartalmára, nemsokára meglátod, hogy számos különböző aspektusát ismerheted meg a Windows 8 alkalmazásfejlesztésének. Egymástól jelentősen különböző témaköröket kell érintenünk, azonban egy közös van bennük: azok mind a Windows 8 platform szolgáltatásaira épülnek.

A Windows 8 a korábbi Windows operációs rendszerektől eltérően nem csupán a már hagyományosnak mondható környezeteket hivatott támogatni, hanem az informatika aktuális vívmányának, az egyre nagyobb teret hódító táblagépnek (vagy más néven *tablet*nek) a követelményeit is elemi szinten ismeri és képes kiszolgálni. Miért van szükség arra, hogy a rendszer már tervezésétől fogva igazodjon az új környezethez? Az első alfejezetben részletesen megnézzük, milyen kihívásokkal kell szembenéznie egy operációs rendszernek, amely a hordozható eszközök új generációját hivatott támogatni. Bevezetésre került egy új életciklus-modell, amely ismerős lehet számodra is, ha fejlesztettél már a Windows Phone platformon. Fontos megismerned és átlátnod ezt a koncepciót, mivel saját alkalmazásaid működését és architektúráját jelentősen befolyásolhatja majd a későbbiekben!

A Microsoft hosszú évek óta biztonsági szempontokat szem előtt tartva implementálja rendszereit, így a Windows 8 kifejlesztése során is nagy hangsúlyt kapott a felhasználói környezet biztonsága. Windows 8 alkalmazások egy úgynevezett homokozóban (*sandbox*) futnak, amely izolálja egymástól azokat, és védelmet nyújt magának az operációs rendszernek is. Ennek következtében megváltozott a fájlok elérése, illetve speciális fájltípusokhoz tartozóan a célmappák kezelése is. Részletesen megnézzük, hogy mire is kell figyelnünk a fejlesztés során. Ezután a parancsikonok világából kilépve életre keltjük az alkalmazásunkat, már jóval azelőtt, hogy a felhasználó azt futtatná. Megnézzük, hogyan hozhatunk létre saját csempét, amely az alkalmazásunk telepítése után az új Start menüben jelenik meg.

Végezetül elérünk ahhoz a témakörhöz, melynek teljes kibontakozását és részletezését a következő fejezetben látjuk majd csak. A táblagépek speciális hardvereit készülünk felhasználni, amelyek programjainkat még inkább képesek a felhasználó fizikai környezetébe olvasztani, ezzel sokkal interaktívabb és célratörőbb felhasználói élményt kialakítva.

Lássunk is hozzá!

A Windows 8 életciklus-modellje

A Windows 8 újdonságai nem csupán a felhasználók számára jelentenek egy teljes arzenálnyi megismernivalót, a fejlesztők világában is jelentős változások történtek. A táblagépeken való futtatásra tervezett modern Windows 8 stílusú alkalmazásoknak számos megszorítással kell élniük.

Miért van ezekre szükség? Gondoljunk csak bele: egy táblagép esetében nem fogadható el az erőforrások teljes kimerítése, mert ezzel csak rövid üzemidőt lehetne garantálni. A megszokott perifériakörnyezet hiányában az érintőképernyőhöz igazított, teljes képernyős alkalmazások közül csak az látható, amelyet a felhasználó éppen futtat, és így nem célszerű hagyni, hogy a háttérben számos más tevékenységet végezve

a rendszer a felhasználó tudta nélkül használja el az erőforrásokat. Ahhoz, hogy a fenti követelményeknek a Windows 8 meg tudjon felelni, a Microsoftnak egy új életciklus-modellt kellett bevezetnie, amely egy állapotgépszerű működést ír elő.

> *Nem lehet eléggé hangsúlyozni, ezért a fejezeten keresztülhaladva több alkalommal is találkozhatsz a következő kijelentéssel: A Windows 8 stíluselveit követő alkalmazások nem az egyetlen lehetőséget jelentik a platformra való fejlesztés során! A hagyományos asztali alkalmazásmodellt követő programok továbbra is futtathatók és fejleszthetők Windows 8 alatt, azonban a fejezet célja az új, modern stílusú alkalmazásfejlesztés megismertetése. A következőkben leírt életciklus-modell kizárólag erre a programtípusra vonatkozik!*

Amíg a felhasználó egy alkalmazással dolgozik, azt nem zárja be és „lepihenni" sem hagyja, addig arra úgy tekintünk, hogy éppen fut. A memóriaterülete él, az erőforrásokat használhatja és kommunikálhat a felhasználóval. Azonban ha a felhasználó áttér egy másik alkalmazásra, az addig futtatott programot az operációs rendszer „pihenteti". Annak memóriaterületében továbbra is ott van az alkalmazás, így majd ahhoz később visszatérve a „pihentetés" előtti állapota helyreállítható. Ez azonban nincs minden esetben így! Ha túl régen használtuk az alkalmazást, vagy az operációs rendszer úgy találja, hogy az operatív memória fogyóban van, akkor véglegesen bezárhatja az alkalmazásunkat, amely az állapot mentésének hiányában elveszítheti a felhasználó teljes munkafolyamatát! Az új életciklus-modellt a 9-1 ábra mutatja be.

9-1 ábra: Az új életciklus-modell állapotai

Elméletben már tudjuk, hogyan kerülnek végrehajtásra a Windows 8 alkalmazások. Azt is meg kell néznünk, hogy a fent említett dolgok hogyan csapódnak le a kódban! Ha létrehozunk egy Windows Store alkalmazást (vagy elővesszük egy már működő alkalmazás kódját), akkor nyissuk meg az **`App.xaml.cs`** fájlt! Ez az alkalmazásunk gyökerének (belépési pontjának) képzelhető el, és az alkalmazásszintű tevékenységeket itt tudjuk felügyelni. Két eseménykezelőt fogunk itt találni: az **`OnLaunched`** és az **`OnSuspending`** metódusokat. Ahhoz, hogy ezek működését megismerhesd, készíts két üres metódust, amelyek az alkalmazásállapot mentéséért (**`SaveState`**) és későbbi visszatöltéséért (**`LoadState`**) fognak felelni!

```
private void SaveState()
{
    // Itt mentjük az állapotot
}

private void LoadState()
{
    // itt tölthetjük vissza az elmentett állapotot
}

protected override void OnLaunched(LaunchActivatedEventArgs args)
{
    Frame rootFrame = Window.Current.Content as Frame;

    if (rootFrame == null)
    {
        rootFrame = new Frame();

        if (args.PreviousExecutionState == ApplicationExecutionState.Terminated)
        {
            LoadState();
```

```
        }

        Window.Current.Content = rootFrame;
    }

    if (rootFrame.Content == null)
    {
        if (!rootFrame.Navigate(typeof(MainPage), args.Arguments))
        {
            throw new Exception("Failed to create initial page");
        }
    }

    Window.Current.Activate();
}

private void OnSuspending(object sender, SuspendingEventArgs e)
{
    var deferral = e.SuspendingOperation.GetDeferral();

    SaveState();

    deferral.Complete();
}
```

Vegyük sorra, mi is történik az **OnLaunched** eseményvezérlőn belül! Amikor az alkalmazásunkat elindítja a felhasználó vagy visszatér hozzá, az **OnLaunched** esemény mindig le fog játszódni. Első dolgunk meghatározni, hogy melyik felülethez kell visszatérni a programnak. Ha „pihentetett" állapotból indult az alkalmazás, nincs más dolgunk, mint a memóriában található felületelemet visszahelyezni, és aktiválni az alkalmazásablakot. Ha nem ez történt, akkor joggal feltételezzük, hogy már volt futtatva, de lezárásra (terminálásra) került, mert túl sok memóriát használt, vagy hosszú ideig nem tért vissza ahhoz a felhasználó. Megvizsgálhatjuk az **ApplicationExecutionState** felsorolt típus segítségével, hogy a leállításnak mi volt a pontos oka, majd a saját készítésű **LoadState** metódus segítségével betöltjük a tárból az elmentett állapotot. Az is egy lehetőség, hogy még soha nem futott az alkalmazás. Ebben az esetben létrehozzuk a számára szükséges gyökérelemet, majd a főképernyőre navigálunk a navigációs kontextus segítségével.

Amint a felhasználó elhagyni szándékozik a programot, vagy egy ideje nem használta a táblagépet, és ezért a rendszer lekapcsolja a képernyőt, lejátszódik az **OnSuspending** esemény. Ha nem akarjuk kockáztatni a felhasználói munkamenetének konzisztenciáját és esélyt adni a lezárás miatti adatvesztésre, ebben az eseményben kell az állapot elmentéséről gondoskodnunk.

Nagyon vigyázzunk, mert az **OnSuspending** esemény maximum 5 másodpercig futhat! Ha ennyi idő nem volt elegendő az adatok mentésére, az operációs rendszer akkor is lezárja az alkalmazásunkat! Érdemes a munkafolyamat közben, apróbb részletekben menteni a releváns adatokat, így a „kilépés" idejére már nem marad komolyabb tennivalónk, és a lehetséges hibák számát is csökkenthetjük.

Az alkalmazás lehetséges állapotainak listáját és használati eseteiket a 9-1 táblázat tartalmazza.

9-1 táblázat: A Windows 8 stílusú alkalmazások állapotai

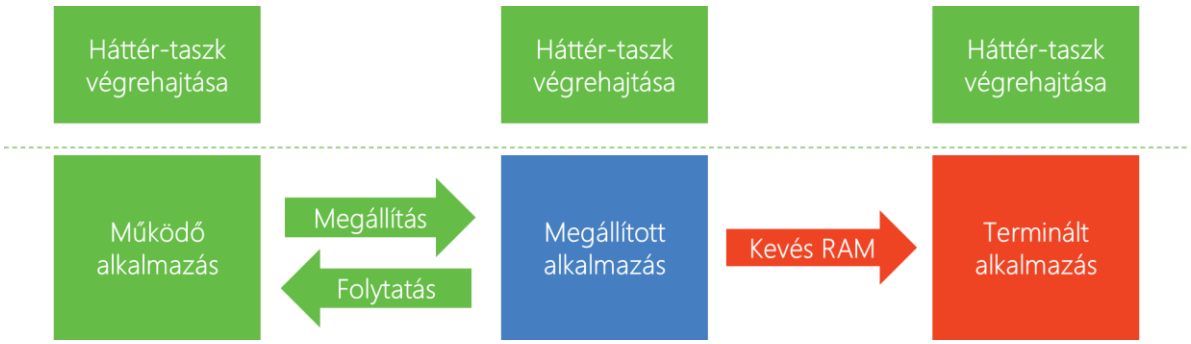
Állapot	Mikor következik be?	Hogyan kezeljük?
NotRunning	Először fut az alkalmazás: ■ A Windows Store-ból lett telepítve ■ A Task Manager segítségével zárták be az alkalmazást ■ Felhasználói ki-/bejelentkezés történt ■ A számítógépet újraindították	A szokásos módon inicializáljunk, és jelenítsük meg az alapértelmezett felhasználói felületet!

Állapot	Mikor következik be?	Hogyan kezeljük?
	Alt+F4 segítségével vagy a bezárás gesztussal (táblagép esetében) lett bezárva a program (10 másodpercen belül újra el lett indítva).	
`Running`	A programot a csempéje vagy valamelyik integrációs esemény segítségével elindították.	Kezeljük le az **`Activation`** eseményt!
`Suspended`	A program pihentetésre került, mivel nem történt egy ideje felhasználói interakció.	Kezeljük le az **`Activation`** eseményt!
`Terminated`	A rendszer kezd kifogyni az erőforrásaiból (tápellátás, szabad operatív memóriaterület stb.), így kénytelen az alkalmazásunkat pihentetett állapotból véglegesen leállítani.	A mentett adatokat állítsuk vissza, törekedjünk a felhasználói munkafolyamat szakadásmentes, hiánytalan visszaállítására! Használjuk ehhez az alkalmazásszintű eseményeket!
`ClosedByUser`	Több mint 10 másodperc telt el az Alt+F4 vagy bezárás gesztus alapú bezárási események óta.	Inicializáljunk a szokásos módon, jelenítsük meg az alapértelmezett felhasználói felületet, és indítsunk egy friss munkafolyamatot!

Az életciklus folyamattal kapcsolatban mindig tartsuk fejben a következő fontos tényezőket:

- A pihentetett állapotból lezárásra kerülő alkalmazásokat nem figyelmezteti a Windows 8. A bezárás oka csak a következő futtatáskor derül ki.
- A Windows 8 stílusú alkalmazásokat úgy tervezték, hogy nem számolunk a felhasználó általi valódi bezárással (Alt+F4), inkább az operációs rendszerre bízzuk az alkalmazások kezelését. Ha mégis valódi bezárás történt, külön lekérdezhető az **`ApplicationExecutionState`** felsorolt típus segítségével.
- Az alkalmazás eltávolításával egy időben az alkalmazáshoz tartozó lokális adatok is eltávolításra kerülnek.
- Az alkalmazás a *Contract & Extension* megközelítés segítségével (erről részleteket a következő fejezetben találhatsz) mélyebben integrálható, amely azt jelenti, hogy az konkrét céllal is indítható (pl. fénykép készítése, fájl megnyitása, keresés indítása stb.). A speciális esetek lekezelése mind az **`OnLaunced`** eseményen keresztül történik.

Nem beszéltünk még egy speciális esetről. Mi történik, ha az alkalmazásnak bizonyos műveleteket akkor is el kell végeznie, amikor nem fut előtérben? Erre a Microsoft teljes háttérfeladat infrastruktúrát hozott létre. Egy alkalmazás feliratkozhat különböző háttérfeladatok futtatására, és az operációs rendszer segítségét kérheti az ütemezésben. A háttérfeladatokat a következő fejezet mutatja be részletesen. A működését addig is előrevetíti a 9-2 ábra.

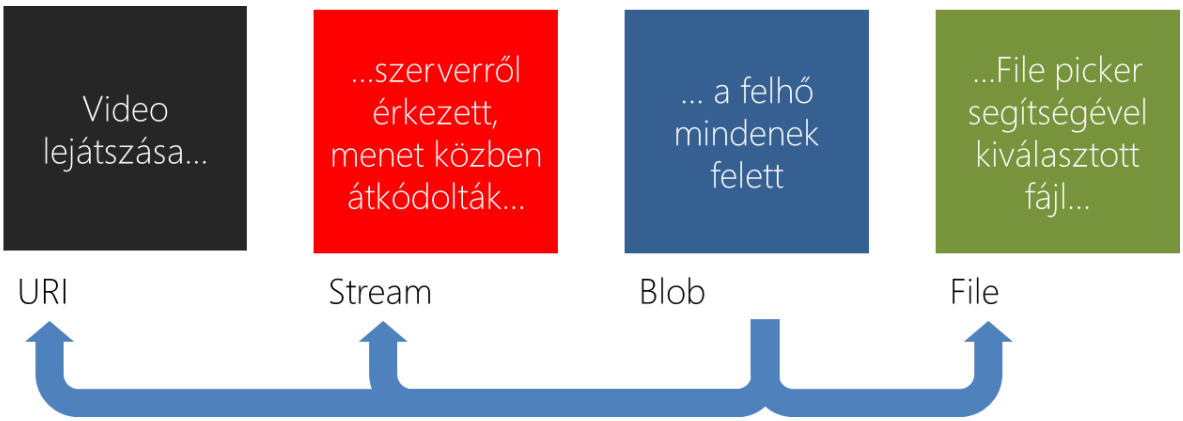


9-2 ábra: Feladatok végrehajtása a háttérben

Fájlok elérése

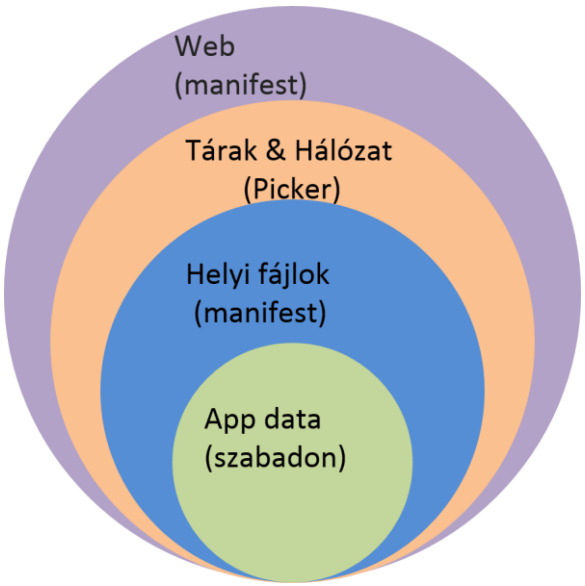
Amikor Windows 8 alkalmazások fájlkezeléséről beszélünk, bele kell gondolnunk, hogy az adott állomány hányféle különböző helyről származhat! Gondoljunk csak egy egyszerű videóra! Ha az interneten keresztül szeretnénk azt elérni, akkor egy URL/URI cím segítségével azonosítjuk. A programnak, amelyik használni kívánja, le kell tudnia tölteni azt, értelmeznie és kezelnie kell az adott formátumot. Ha a videót különböző sávszélességgel rendelkező, esetleg más technikai környezetbe tartozó felhasználók számára is elérhetővé szeretnénk tenni, akkor valamilyen *streaming* forrást fogunk megjelölni, ahonnan a videót egy vagy több video állomány formájában visszakapjuk. Ugyanezek az állítások igazak akkor is, ha valamilyen felhőalapú szolgáltatást veszünk igénybe a fájlok tárolásához, hiszen azok nem minden esetben rendelkeznek egy, a felhasználó gépén található másolattal.

A Windows 8 (és ezzel a Windows Runtime) a modern stílusú alkalmazásokhoz illeszkedő aszinkron objektumok és műveletek segítségével egységesíti a fájlkezelést és adatelérést, amint azt a 9-3 ábra mutatja.



9-3 ábra: Ugyanaz az adat különböző forrásokból érkezhet

A Windows 8 esetében az alkalmazásaink egymástól és az operációs rendszer részeitől szeparáltan futnak a homokozón (sandbox) belül. Ez a környezet szigorúan felügyeli, hogy a programunk hova is szeretne kitekinteni fájlok után kutakodva. A 9-4 ábra halmazokon reprezentálja, hogy a fájlokat elhelyezkedésük alapján kategorizálva hogyan érhetjük el egy Windows 8 stílusú alkalmazáson belül.



9-4 ábra: Fájlok elérési lehetőségei a tárolási helytől függően

Az alkalmazásunkhoz csomagolt fájlok (úgynevezett *Application Data*) szabadon hozzáférhetők, nem kell jogosultsági kérelmeket leadnunk hozzájuk. Ide tartoznak az alkalmazáshoz csomagolt képek, zenék, XML és egyéb típusú fájlok.

Windows 8 alatt (ahogyan már elődjénél is) a felhasználó speciális mappákkal rendelkezik, amelyekben zenéket, videókat, dokumentumok és egyéb más tartalmakat tárolhat. Ezek a mappák a tartalmukkal együtt programból – a felhasználó közbeavatkozása nélkül – kezelhetők, azonban az alkalmazáshoz tartozó *manifest* állományon keresztül a progamnak ezt az igényét jeleznie kell. Ez az igény tartalmazza, hogy melyik mappához szeretnénk hozzáférni, milyen célból, és milyen típusú fájlokat szeretnénk azon belül elérni. A manifestben megjelölt adatok az alkalmazás fordítása és csomagolása során bekerülnek az alkalmazásba, és a Windows Store-ba való feltöltésükkor azonnal meg is jelenik a program által felhasználni kívánt funkciók listáján. Amikor egy jövőbeli vásárlónk majd le kívánja tölteni a programunkat, ott fog szerepelni az alkalmazás adatlapján, hogy milyen komponenseket kell annak elérnie az operációs rendszerből. Jelen esetben ez a felhasználó mappák kezelésére fog vonatkozni.

A fájlrendszerhez, a felhasználó számítógépéhez tartozó merevlemezekhez és egyéb fájltárakhoz programozott módon hozzáférni nincs lehetőségünk, és erről minden esetben tájékoztatnunk kell a felhasználót. Ebben a *Picker Contract* segít nekünk, amelynek a működését nemsokára megismerjük.

Webes tartalmak letöltéséhez és eléréséhez megint csak a manifest fájlon keresztül kell engedélyt kérnünk. Ha az engedélyt megkaptuk, akkor a hivatkozott tartalmak már programból kezelhetők.

Fájlok kiválasztása a Picker Contract-ok segítségével

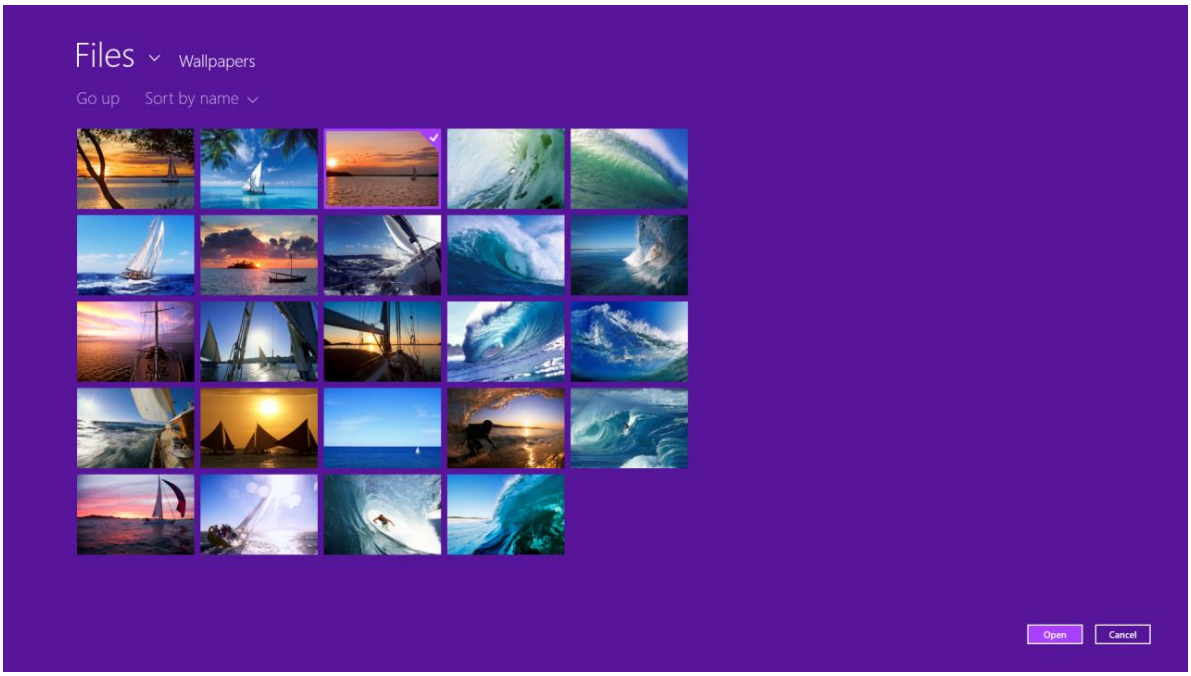
Programozott módon nem választhatunk ki fájlokat a helyi fájlrendszerből vagy a hálózatról, csak és kizárólag a speciális funkciót ellátó mappában (Képek, Dokumentumok stb.) találhatókat vagy a programhoz tartozókat. Az egyetlen lehetőségünk a felhasználó segítségül hívása. A **`FileOpenPicker`** és a **`FileSavePicker`** objektumok segítségével az operációs rendszerbe integrált fájlok kiválasztására alkalmas felületet elő tudjuk varázsolni saját alkalmazásunkon belül. Használata rendkívül egyszerű. Az alábbi kódrészlet egy képfájl megnyitására alkalmas metódust mutat be:

```
async void OpenFile()
{
    FileOpenPicker filePicker = new FileOpenPicker();
    filePicker.SuggestedStartLocation = PickerLocationId.PicturesLibrary;
    filePicker.ViewMode = PickerViewMode.Thumbnail;
    filePicker.FileTypeFilter.Add( ".jpg" );
    filePicker.FileTypeFilter.Add( ".png" );

    StorageFile file = await filePicker.PickSingleFileAsync();
    if ( file != null )
    {
        using ( IRandomAccessStream fileStream =
                                    await file.OpenAsync( FileAccessMode.Read ) )
        {
            BitmapImage image = new BitmapImage();
            image.SetSource( fileStream );

            // kép további feldolgozása, vagy megjelenítése
        }
    }
}
```

Először létrehoztunk egy új objektumpéldányt, melynek megadjuk a **`SuggestedStartLocation`** tulajdonságon keresztül, hogy az aktuálisan bejelentkezett felhasználó Képek mappáját mutassa. A nézeti módhoz beállítjuk, hogy apró előnézeti képeket mutasson, melyek felső sarkában egy kis jelölő segítségével kijelölhető a fájl (listás nézet is lehetséges a **`PickerViewMode.List`** elem használatával). A paraméterek közül talán a legfontosabb a **`FileTypeFilter`** gyűjtemény, amely a kezelni kívánt fájlok kiterjesztéseit tartalmazza egyszerű szöveges formájában. A fenti példában a **`.jpg`** és **`.png`** fájlokat szeretnénk használni.

Fájlok megnyitásakor erőforrásokkal dolgozunk, melyekről már tudjuk, hogy Windows 8 alkalmazások esetében nem érhetők el szinkron módon várakoztatással, hanem aszinkron híváson keresztül majd a háttérben zajlik le a folyamat. A **`PickSingleFileAsync()`** metódus hívásával egyetlen fájl aszinkron kiválasztására adhatjuk le igényünket. Ahhoz, hogy az **`await`** operátor segítségével hívható legyen, magát az **`OpenFile()`** metódusunkat is el kellett látnunk az **`async`** módosítóval.

Ezután egyszerű ellenőrzés következik arra nézve, hogy a felhasználó nem a Mégse (Cancel) gombot választotta-e. Ha nem, akkor nincs más dolgunk, mint egy stream segítségével aszinkron módon megnyitni a fájlt. Jelen esetben egy a Windows Runtime által feldolgozható képet állítottunk elő a fájl tartalmából, amelyet könnyedén ki is rajzolhatnánk a felhasználói felületre. A példakódhoz tartozó programot futás közben a 9-5 ábra mutatja be. A fájlok mentése is hasonlóan egyszerű a **`FileSavePicker`** segítségével.



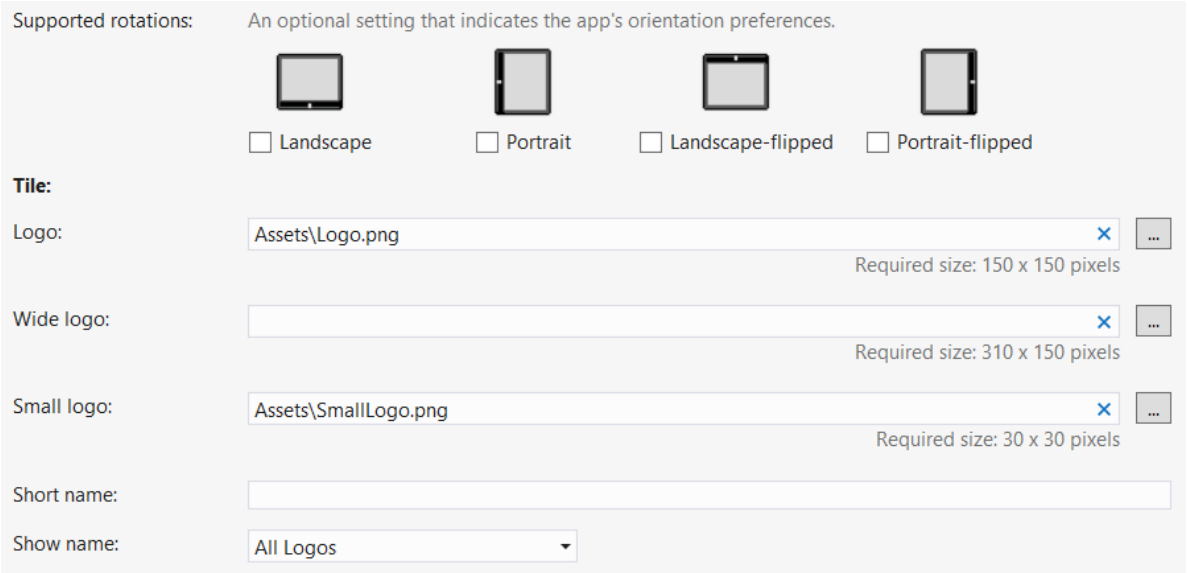
9-5 ábra: A FileOpenPicker contract futtatás közben

Csempék kezelése

A legelső dolog, amit a felhasználók megpillantanak a Windows 8 esetében, az új Start képernyő. Hamar ráébrednek, hogy az ikonok, melyeket a felületen látnak, nem csupán apró képek – ezeket korábban parancsikonoknak hívtuk –, hanem az adott alkalmazások miniatűr reprezentációi. Izegnek-mozognak, különböző tartalmak láthatók rajtuk, valamint időről időre változik a megjelenésük. A naptáralkalmazás az aktuális napra vonatkozó ünnepet, illetve ismerőseink születésnapját mutatja. Az időjárás alkalmazás az aktuális hőmérsékletadatokat jelzi, továbbá figyelmeztet bennünket, hogy milyen időjárásra készüljünk, ha kilépünk a lakásból. Nem biztos, hogy a fent említett programok közül bármelyiket is futtatta már a felhasználó, mégis azok akár a program futtatása előtt megjelenítik az alkalmazáshoz tartozó fontosabb információkat, ráadásul rögtön a Start képernyő felületén.

Ilyen *csempéket* mi magunk is készíthetünk. Egyszerű ikont tartalmazóktól egészen a legfrissebb hírcsatornákat megjelenítő interaktív felületig mi magunk is sokféle dolgot építhetünk.

Alapvetően kétféle csempeváltozat létezik: a négyzetes (150 × 150 pixel) és a széles (310 × 150 pixel). Az itt leírt méretek az alapértelmezett megjelenéshez tartozó háttérkép méretét is jelölik egyben, amely JPEG és PNG formátumokban elfogadott, maximálisan 150 KByte fájlmérettel. Az alkalmazásunkhoz tartozó ikon kicserélése nagyon egyszerű feladat, nincs más dolgunk, mint duplán a **`Package.appxmanifest`** fájlra kattintani, majd a megjelenő panelen a Logo és a Wide logo mezőknek értéket adni (9-6 ábra).



9-6 ábra: A lapka háttérképének kiválasztása

A csempéken látható információ dinamikussá alakítására több lehetőségünk is van, az egyszerűbbik eset bemutatásával fogjuk kezdeni. A lapka jobb alsó sarkában – függetlenül attól, hogy négyzetes vagy széles módba van állítva – meg tudunk jeleníteni apró ikonokat vagy számokat 1-től 99-ig. Gondoljunk csak egy zenelejátszó alkalmazásra, amely a zene megállításakor a jól ismert szüneteltetés (paused) jelet helyezi el a felületen. Esetleg példának hozható egy üzenetkezelő alkalmazás is, amely számokkal jelzi számunkra az olvasatlan üzenetek számát.

```
void ShowPauseBadge()
{
    XmlDocument badgeXml =
        BadgeUpdateManager.GetTemplateContent( BadgeTemplateType.BadgeGlyph );

    XmlElement badgeElement = (XmlElement)badgeXml.SelectSingleNode( "/badge" );
    badgeElement.SetAttribute( "value", "paused" );

    BadgeNotification badge = new BadgeNotification( badgeXml );
    BadgeUpdateManager.CreateBadgeUpdaterForApplication().Update( badge );
}
```

A **`ShowPauseBadge()`** metódus szerkezete nagyon egyszerű. A csempék kezelése mindig sablonok segítségével történik, ahol a sablon egy XML szerkezet. A jelenlegi esetben a választási lehetőségek száma kettőre korlátozódik: vagy ikonokat, vagy számokat megjelenítő XML-t kell választanunk a **`BadgeTemplateType`** felsorolt típus segítségével. Ezután megkeressük az XML fában a **`badge`** ágat, amely rendelkezik egy **`value`** attribútummal. Annak értéket adunk, és a **`BadgeUpdateManager`** osztály segítségével frissítjük a csempét. Az ikonok megnevezéseit a 9-2 táblázat tartalmazza.

9-2 táblázat: A csempéken megjeleníthető ikonok

Value attribútum	Ikon	XML leírás
`none`	Nincs ikon megjelenítve	**`<badge value="none"/>`**
`activity`		**`<badge value="activity"/>`**
`alert`		**`<badge value="alert"/>`**
`available`		**`<badge value="available"/>`**

Value attribútum	Ikon	XML leírás
away		<badge value="away"/>
busy		<badge value="busy"/>
newMessage		<badge value="newMessage"/>
paused		<badge value="paused"/>
playing		<badge value="playing"/>
unavailabe		<badge value="unavailable"/>
error		<badge value="error"/>
attention		<badge value="attention"/>

A számok megjelenítése is hasonlóan egyszerű. Vigyázzunk a **BadgeTemplateType** felsorolás ide vonatkozó **BadgeNumber** elemének a használatára, valamint a **value** attribútum helyes beállítására! A **ShowPlayingCount()** metódus a számjelzéses csempefrissítést mutatja be.

```
void ShowPlayingCount()
{
    XmlDocument badgeXml =
                BadgeUpdateManager.GetTemplateContent( BadgeTemplateType.BadgeNumber );

    XmlElement badgeElement = (XmlElement)badgeXml.SelectSingleNode( "/badge" );
    badgeElement.SetAttribute( "value", "3" );

    BadgeNotification badge = new BadgeNotification( badgeXml );
    BadgeUpdateManager.CreateBadgeUpdaterForApplication().Update( badge );

}
```

Természetesen rengeteg eset van, amikor a számozás vagy az apró ikonok megjelenítése nem bizonyul elegendőnek. Lehetőségünk van a csempe teljes kinézetének átalakítására képek segítségével, illetve dinamikus szöveges tartalmak megjelenítésére is. Hasonlóan az előző példához, itt is sablonokat fogunk alkalmazni. A sablonok száma itt azonban már jóval magasabb, mint az előző esetben.

Meg kell különböztetnünk azt a két esetet, amikor a csempe négyzetes, illetve szélesre állított módban van, majd ennek függvényében a megfelelő sablon paramétereit kell beállítanunk! A következő kódminta egy olyan frissítő metódust vezet be, amely mind a két esetre fel van készítve, a csempére kiírt üzenetet pedig 10 másodpercig képes megjeleníteni.

```
void UpdateTile()
{
    XmlDocument tileXml =
      TileUpdateManager.GetTemplateContent( TileTemplateType.TileWideImageAndText01 );

    XmlNodeList tileTextAttributes = tileXml.GetElementsByTagName( "text" );
    tileTextAttributes[0].InnerText = "A csempére kiírt üzenet!";

    XmlNodeList tileImageAttributes = tileXml.GetElementsByTagName( "image" );
    ((XmlElement)tileImageAttributes[0]).
         SetAttribute( "src", "ms-appx:///images/basetile.png" );
    ((XmlElement)tileImageAttributes[0]).SetAttribute( "alt", "red graphic" );
```

```
    XmlDocument squareTileXml =
      TileUpdateManager.GetTemplateContent( TileTemplateType.TileSquareText04 );
    XmlNodeList squareTileTextAttributes = squareTileXml.GetElementsByTagName( "text" );
    squareTileTextAttributes[0].
      AppendChild(squareTileXml.CreateTextNode( "Hello World!" ));
    IXmlNode node = tileXml.ImportNode(
      squareTileXml.GetElementsByTagName( "binding").Item( 0 ), true );
    tileXml.GetElementsByTagName("visual").Item(0).AppendChild(node);

    TileNotification tileNotification = new TileNotification( tileXml );
    tileNotification.ExpirationTime = DateTimeOffset.UtcNow.AddSeconds( 10 );

    TileUpdateManager.CreateTileUpdaterForApplication().Update( tileNotification );
}
```

Először is két különböző sablont hozunk létre. A **`tileXml`** változóban tárolt változat a széles megjelenítési módba váltott lapkát támogatja. Elhelyezünk az XML-ben egy szöveget, majd a csempéhez tartozó képet is megváltoztatjuk az **`image`** ág **`src`** tulajdonságán keresztül (szigorúan vigyázva a megfelelő erőforrás-hivatkozásra).

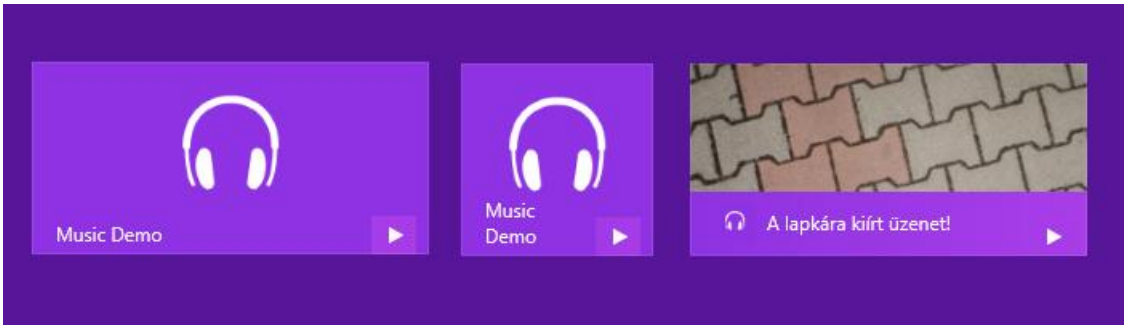
A következő lépésben a négyzetes megjelenítési módhoz tartozó sablont módosítjuk az előzőhöz egészen hasonlóan. Miután mind a két sablon a rendelkezésünkre áll, össze kell azokat fűznünk, mivel egyetlen XML fa keretein belül szeretnénk átadni a **`TileUpdateManager`** objektumnak.

Ha az összefűzés megtörtént, beállíthatjuk még az értesítés módját is. A csempén megjelenő üzenetet mindössze 10 másodpercig szeretnénk láthatóvá tenni, ezért a **`TileNotification`** objektumpéldányunk **`ExpirationTime`** tulajdonságának is értéket adunk. Ezután valóban nincs más dolgunk, mint a **`TileUpdateManager`** segítségével beütemezni a frissítést.

A fenti változtatások a következő XML sablont fogják létrehozni – természetesen mi ezt nem látjuk, mindössze a fenti kódok hatása fog érvényesülni:

```
<tile>
    <visual lang=”en-US>
        <binding template=”TileWideImageAndText01”>
            <image id=”1” src=”ms-appx:///images/basetile.png”/>
            <text id=”1”>A lapkára kiírt üzenet!</text>
        </binding>

        <binding template=”TileSquareText04”>
            <text id=”1”>Hello World!</text>
        </binding>
    </visual>
</tile>
```

A 9-7 ábra a különböző csempeállapotokban készített pillanatfelvételeket tartalmazza.



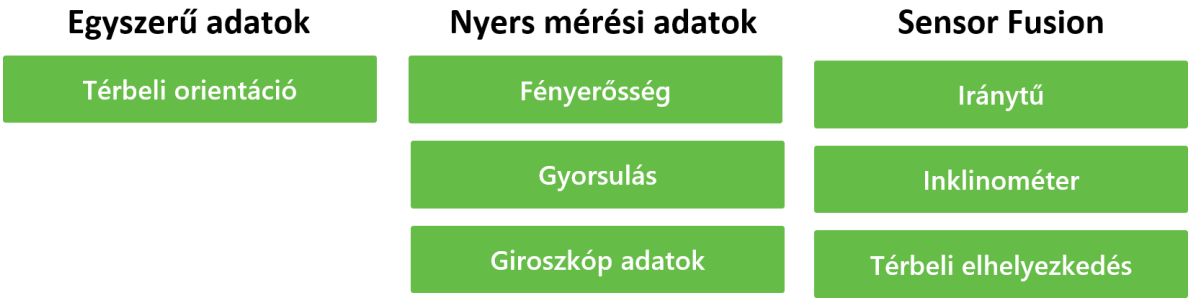
9-7 ábra: A különböző csempeállapotok a program futtatása során

Szenzorok kezelése

A Windows 8 a különböző hardverek és szenzorok kezelését teljesen új szintre emelte. A táblagépeknek köszönhetően a felhasználó munkakörnyezete jelentősen átalakult. Nem olyan méretes számítógéppel dolgozunk, amelyhez a szoba bútorzatát és berendezését igazítjuk, hanem egy könnyed kis eszközről beszélünk, amely egész nap segítheti a felhasználót – különböző kihívásoknak megfelelve.

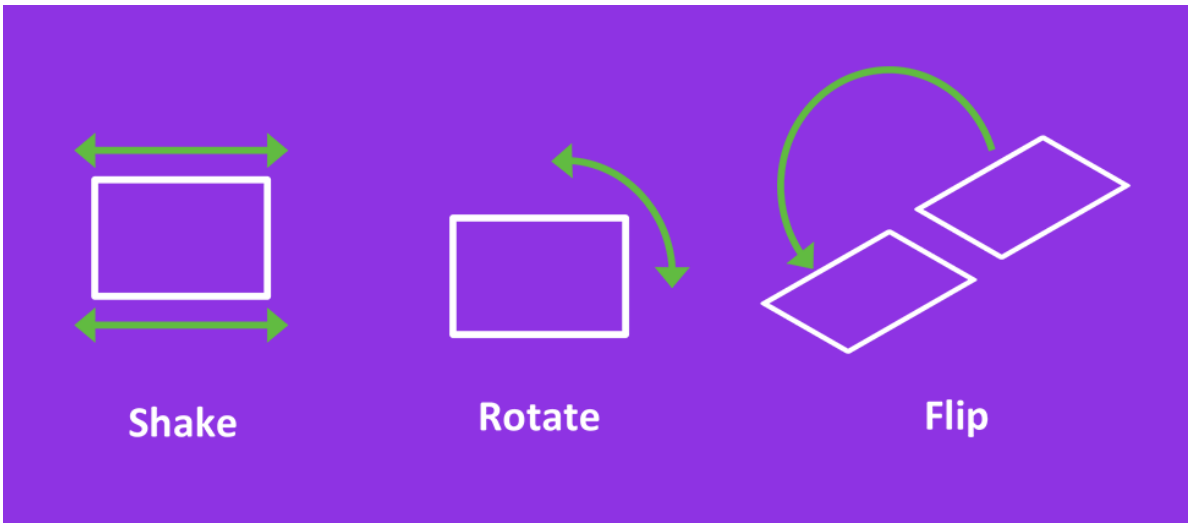
A folyamatos környezetváltozáshoz igazodnunk kell, a fényviszonyoknak megfelelő megvilágítást kell adnunk a kijelzőnek, az alkalmazásoknak az aktuális térbeli elhelyezkedésnek megfelelő nézetet kell biztosítanunk, a különböző információk kiszolgálásához pedig tudnunk kell, hogy éppen merre jár a felhasználó.

A táblagépek rengeteg olyan szenzorral vannak felszerelve, amelyekkel eddig nem igazán találkozhattunk. A szenzorok különböző típusú adatokat szolgáltatnak vissza, és különböző mérési elvárásokkal rendelkeznek. Ha magunknak kellene ebben az információáradatban kiigazodnunk, elég sok munka állna előttünk már egészen egyszerű alkalmazások megírása során is. Azonban a Windows 8 platform szinten támogatja az eszközök kezelését, az általuk mért adatokat pedig a legkényelmesebb formában teszi számunkra elérhetővé. Nem mindig szükséges tudnunk, hogy melyik fizikai eszköz mérte a legutóbbi adatokat, csupán azt kell megmondanunk, hogy mire vagyunk kíváncsiak. A 9-8 ábra a különböző mérési lehetőségeket mutatja be.



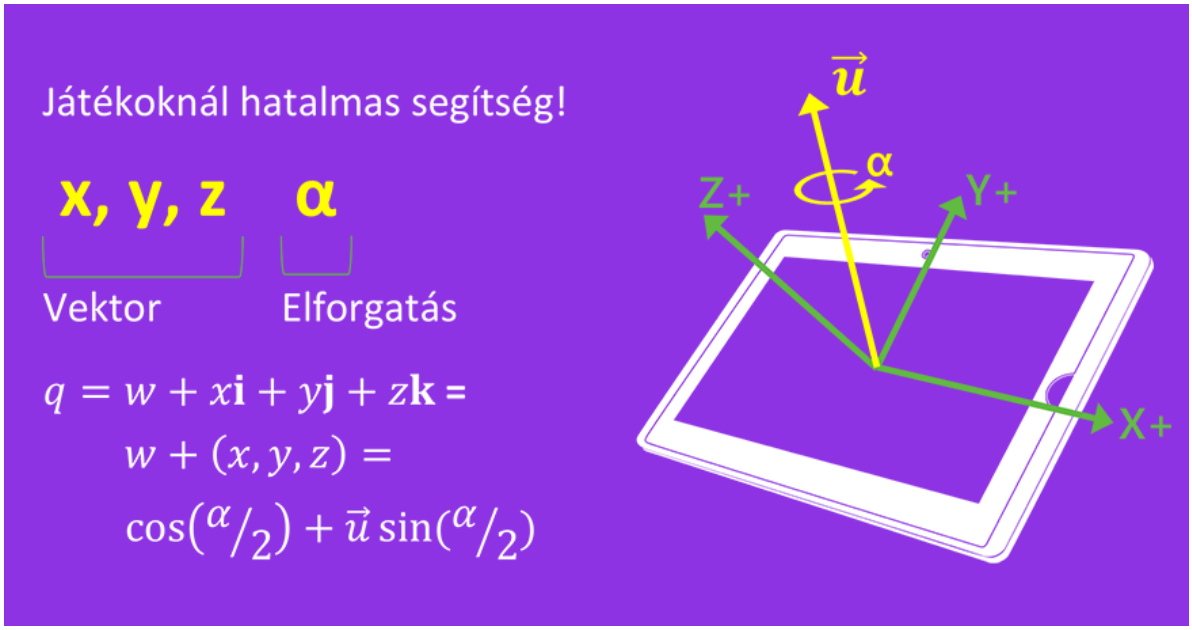
9-8 ábra: A Windows 8 által szolgáltatott mérési lehetőségek

Vegyük a következő egyszerű példát: szükségünk van az adott eszköz térbeli elhelyezkedésére, azonban ez a mi esetünkben kettős fogalomnak számít. Nem minden esetben szükséges tudnunk, hogy a Descartes-féle koordinátarendszerben különböző tengelyekre nézve mekkora szöggel fordították el a táblagépet, vagy adott tengelyen hány egységgel mozdult el. Mindösszesen arra van szükségünk, hogy tudjuk: álló vagy fekvő üzemmódban használja-e a felhasználó a gépezetet, ezzel az adott orientációhoz legjobban illeszkedő felhasználói felületet jeleníthetjük meg számára. Az egyszerű orientációs adatokra mutat példákat a 9-9 ábra.



9-9 ábra: Az elhelyezkedéssel összefüggő néhány lehetséges esemény

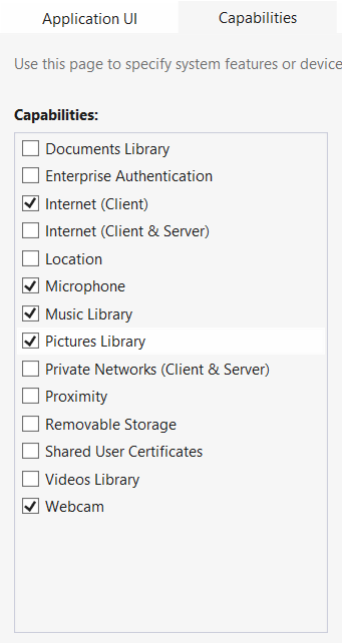
Következő példának nézzük a számítógépes játékokat! Elképzelhető, hogy egy olyan autószimulátort írunk, ahol a felhasználó a táblagép forgatásával kormányozhatja autóját. Itt nem elegendő azt tudnunk, hogy éppen fekvő vagy álló állapotban van-e az eszköz, helyette nagyon pontos szögekkel és iránymenti gyorsulásokkal kell számolnunk. Különböző szenzorok szükségesek a két említett egység kiméréséhez, és a visszaszolgáltatott adatokat valahogy fel kell dolgoznunk. A Windows 8 bevezeti a *sensor fusion* fogalmát, amely lehetővé teszi, hogy a pontos hardvereszközök ismerete nélkül mérjünk összetett adatokat, esetünkben különböző vektorértékeket és szögelfordulásokat. A visszaérkező információ pedig a játékokban a legkönnyebben felhasználható formában kerül vissza, úgynevezett *kvanterniók*ra leképezve. A 9-10 ábra vizuális formában is bemutatja a leírtakat.



9-10 ábra: A mérések egyszerűsödése Windows 8 platform esetében

Egy egyszerű példa: kamera és mikrofon használata

A következő példában egy kamerát és mikrofont használó programrészt fogunk elkészíteni. Nincs szükségünk másra, mint egy Windows Store alkalmazássablonra Visual Studióban. Említettem többször is a fejezet során, hogy a homokozónak és a Windows Store elvárásoknak köszönhetően nem férhetünk hozzá szabadon a különböző hardverelemekhez, az alkalmazás által igényelt komponenseket meg kell jelölnünk a **`Package.appxmanifest`** fájlban. Kattintsunk is rá kettőt, majd a Capabilities fület kiválasztva jelöljük ki az alábbi elemeket: Microphone, Music Library, Pictures Library, Webcam! Ahogyan azt a nevük is mutatja, a rendelkezésünkre álló audio- és videoeszközöket fogjuk használni, majd a rögzített tartalmakat a Windows 8 speciális mappáiba kimenteni. A fent leírt beállításokat a 9-11 ábra mutatja be.



9-11 ábra: A szükséges elemek megjelölése az alkalmazásleíróban

Ha készen vagyunk a konfigurációs fájl módosításával, akkor kezdjünk is neki a kódírásnak! Először a kamerakezelésért felelős kódot hozzuk létre:

```
async void CaptureCamera()
{
    try
    {
        CameraCaptureUI cameraUI = new CameraCaptureUI();
        Size ratio = new Size( 16, 9 );
        cameraUI.PhotoSettings.CroppedAspectRatio = ratio;
        cameraUI.PhotoSettings.Format = CameraCaptureUIPhotoFormat.Png;

        StorageFile imageFile =
             await cameraUI.CaptureFileAsync( CameraCaptureUIMode.Photo );

        if ( imageFile != null )
        {
            await imageFile.RenameAsync( "MediaCaptureDemo.png" );
            await imageFile.MoveAsync( KnownFolders.PicturesLibrary );
        }
    }
    catch ( Exception )
    {
        new MessageDialog( "Hiba történt! :(" ).ShowAsync();
    }
}
```

Felesleges lenne saját képrögzítő felület készítésével bonyolítani a programunkat, ahol csak tudjuk, használjuk a Windows 8 beépített eszközeit! Egy ilyen kiválasztott eszköz határozatlan időre elnavigál az alkalmazásunkból, megjelenít egy új felületet, ahol a felhasználó elvégezheti a kívánt műveletet, majd ha a folyamat jóváhagyásra került, akkor a rögzített adatokkal visszatér a programunkhoz.

Ez aszinkron folyamat, így a metódusunk is előre felkészül az aszinkron működésre az **`async`** kulcsszó megjelenésével. A **`CameraCaptureUI`** objektum használható fényképek és videofelvételek készítésére is. Esetünkben egy olyan fénykép rögzítése lesz a feladata, amely 16:9-es méretarányokkal rendelkezik. A **`CameraCaptureUI`** (és sok más Windows 8 beépített eszköz) szakított a megszokott **`Stream`** alapú adatforgalmazással, helyette közvetlenül fájlba írja a rögzített tartalmakat, amelyeket sokkal gyorsabban tudunk szükség esetén felhasználni, esetleg a felhasználó által választott helyre elmenteni – valójában csak

egy fájlmozgatásra van szükségünk –, és kevesebb memóriát is pazarol. Ebben a példában a rögzített képet a felhasználó fényképek mappájába mozgatjuk. A kivételek kezelésére egész végig figyelünk, és a program futása során előforduló hibákról a felhasználót tájékoztatjuk.

A program másik feladata a mikrofon segítségével hanganyag rögzítése, majd annak a felhasználó zene mappájába való kimentése, amint azt a következő kódrészlet is bemutatja:

```
MediaCapture mediaCapture;

async void InitAudioCapture()
{
    mediaCapture = new MediaCapture();
    MediaCaptureInitializationSettings settings =
         new MediaCaptureInitializationSettings();
    settings.StreamingCaptureMode = StreamingCaptureMode.Audio;

    await mediaCapture.InitializeAsync( settings );
}

async void CaptureAudio( bool startRecording )
{
    try
    {
        if ( startRecording )
        {
            StorageFile audioFile =
                 await KnownFolders.MusicLibrary.CreateFileAsync( "MediaCaptureDemo.mp3",
                        CreationCollisionOption.ReplaceExisting );

            MediaEncodingProfile encoding =
                 MediaEncodingProfile.CreateMp3( AudioEncodingQuality.High );
            await mediaCapture.StartRecordToStorageFileAsync( encoding, audioFile );
        }
        else
        {
            await mediaCapture.StopRecordAsync();
        }
    }
    catch ( Exception ex )
    {
        new MessageDialog( "Hiba történt! :(" ).ShowAsync();
    }
}
```

A **`MediaCapture`** objektum számos formátumot ismer (mp3, m4a stb.), amelyekbe a rögzített audioanyagot rögtön képes is kivezetni, azonban ehhez inicializálásra van szüksége. Az inicializálás során kiválaszthatjuk a hangrögzítést, továbbá – ha több mikrofonunk is van – kijelölhetjük, melyik fizikai eszközt szeretnénk használni. A **`CaptureAudio`** metódus két feladatot is ellát. A bemenő paraméterének függvényében vagy aszinkron módon elkezd hangot rögzíteni, vagy leállítja az éppen folyamatban lévő felvételt.

Ahogyan a fénykép készítésénél, a Windows 8 itt is közvetlenül fájlba vezeti ki a felvétel tartalmát, amelynek minősége (felbontása) a rögzítés megkezdésekor állítható be. Minél részletesebb a hang- illetve a videoanyag, annál nagyobb lesz a fájl mérete is.

Ezek után nincsen más dolgunk, mint valamilyen eseményhez, jelen példában a felületen található gombok lenyomásához hozzárendelni az oda tartozó metódushívásokat:

```
private void captureImage_Click_1(object sender, RoutedEventArgs e)
{
    CaptureCamera();
}

private void captureAudio_Checked_1(object sender, RoutedEventArgs e)
```

```
{
    CaptureAudio( true );

    captureAudio.Content = "Stop";
}

private void captureAudio_Unchecked_1(object sender, RoutedEventArgs e)
{
    CaptureAudio( false );

    captureAudio.Content = "Start";
}
```

Összegzés

Ebben a fejezetben áttekintettük, hogyan integrálható alkalmazásunk a Windows 8 modern stílusú szolgáltatásaival. A táblagépeknek köszönhetően a lehetőségek tárháza szinte végtelen, számos hardveres összetevő, illetve az operációs rendszer új komponensei és eszközei segítik munkánkat, hogy a felhasználóknak valódi élményt és a hosszú tanulási folyamat helyett intuíción alapuló programhasználatot tegyünk lehetővé. Rengeteg lehetőség és újdonság érthető el a fejlesztők számára, ám ezek olyan szükséges változtatásokat hoztak magukkal, mint a korlátozott jogosultsági szint, az alkalmazás mini reprezentációját életre keltő új Start képernyő, valamint az erőforráskezelést támogató életciklus-modell.

A fejezet által lefektetett elméleti és gyakorlati alapokat a következő fejezet során mélyítjük el, a teljesség igénye nélkül. A témaköröket csak felszínesen van lehetőségünk érinteni, első, saját Windows 8 alkalmazásod kifejlesztése során a Microsoft fejlesztői tudástára, az MSDN jelentős segítséget fog nyújtani tudásod kiterjesztésére.

10. Haladó Windows 8 integrációs ismeretek

Ebben a fejezetben az alábbi témákat ismerheted meg:

- Hogyan tudunk az új életciklus-modellben háttérfolyamatokat létrehozni?
- Hogyan tudjuk integrálni alkalmazásunkat a Windows 8 beépített keresőjével?
- Hogyan készítsünk beállításokat kezelő felületet a programunkhoz?
- Milyen módon kérdezzük le az aktuális földrajzi helyzetünket, illetve hogyan használjuk ezt az információt térképen való pozicionáláshoz?

Nehézségét tekintve ez a fejezet nem különbözik az előzőtől vagy a könyv többi fejezetétől, a haladó szó inkább azt fejezi ki, hogy az operációs rendszer szolgáltatásait és integrációs lehetőségeit egyre mélyebben ismerjük meg. Az alapozó fejezetben megnéztük, hogy mit jelent az új életciklus-modell, azonban egy lényegi nézőpontra nem tértünk ki: mi történik, ha az alkalmazás nincs előtérben?

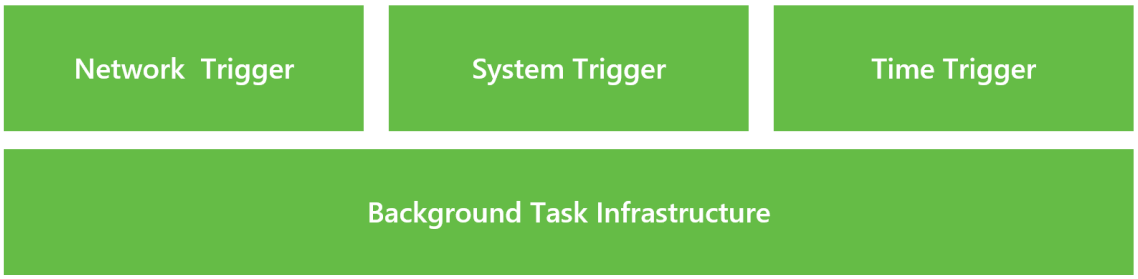
A Windows 8 keresője a Start képernyő és számos beépített alkalmazás részét képezi, amit a felhasználó hamar felfedez az új rendszer megismerése során. Saját alkalmazásainkat is felkészíthetjük, hogy együttműködjenek ezzel a keresővel. Sőt, az alkalmazás beállításait sem kell a programon belül kezelnünk, az operációs rendszer biztosít teret a beállításhoz szükséges felületek kiemelt helyen való megjelenítéséhez is.

Egy újabb példát is láthatunk a szenzorok használatára, amely most a földrajzi pozíciónkat fogja meghatározni bizonyos szolgáltatások és szenzorcsoportok segítségével, majd az információt egy Bing térképen jeleníti meg.

Lássunk is hozzá!

Háttérfolyamatok létrehozása

Ahogyan azt az előző fejezetben már láttuk, az alkalmazásnak egy új működési modellhez kell igazodnia, nem végezhetünk bármit csak úgy a kedvünk szerint. Amikor a felhasználó nem használja az adott gépet, vagy átvált egy másik Windows 8 alkalmazásra, az aktuálisan futtatott programot az operációs rendszer „pihenteti". Bármit is csinált addig – az alkalmazás állapota a memóriában egy ideig megőrződik–, ekkor teljesen passzív státuszba lép. Ha valamilyen tevékenységet a program a háttérben szeretne végrehajtani, amikor egy másik program van az előtérben, egy háttérfolyamatot kell létrehoznunk. Ez a háttérfolyamat nem csinálhat bármit, adott típusú tevékenységeket végezhet, amint azt a 10-1 ábra mutatja be.



10-1 ábra: A háttérfolyamatok típusai

A *trigger*ek az ütemezést segítik. Tegyük fel, hogy a program szeretne hálózati forgalmat generálni a háttérben, így regisztrál egy *Network Trigger*t, amely akkor fog ténylegesen lefutni, amikor az operációs rendszer ütemezési sorában zöld jelzést kap. Ha elvégezte a feladatát, a rendszer továbblép a sorban, és a következő feladatot kezdi el végrehajtani. A legfontosabb *System Trigger* típusokat a 10-1-es táblázat foglalja össze.

10-1 táblázat: A fontos System Trigger típusok

A trigger neve	Mikor váltódik ki?
`InternetAvailable`	Az internetkapcsolat létrejön.
`NetworkStateChange`	A hálózat állapota megváltozik.
`OnlineIdConnectedStateChange`	A felhasználói fiók azonosítója megváltozott.
`SmsReceived`	SMS érkezett (csak megfelelő hardver jelenlétében elérhető).
`TimeZoneChange`	Az időzóna megváltozott (pl. téli-nyári időszámítás közötti átállás).

Saját taszkot létrehozni nagyon egyszerű. Meglévő alkalmazásunk mellé létrehozunk egy új Windows Runtime Component típusú projektet. A létrejött projektben található **`Class1.cs`** fájlt nevezzük át, és implementáljuk az így létrehozott osztályban az **`IBackgroundTask`** interfészt, amely a **`Run`** metódus keretét definiálja:

```
public sealed class CustomBackgroundTask : IBackgroundTask
{
    public void Run(IBackgroundTaskInstance taskInstance)
    {
        var deferralObject = taskInstance.GetDeferral();

        //itt futhat a feladat kódja, akár aszinkron módon is

        deferralObject.Complete();
    }
}
```

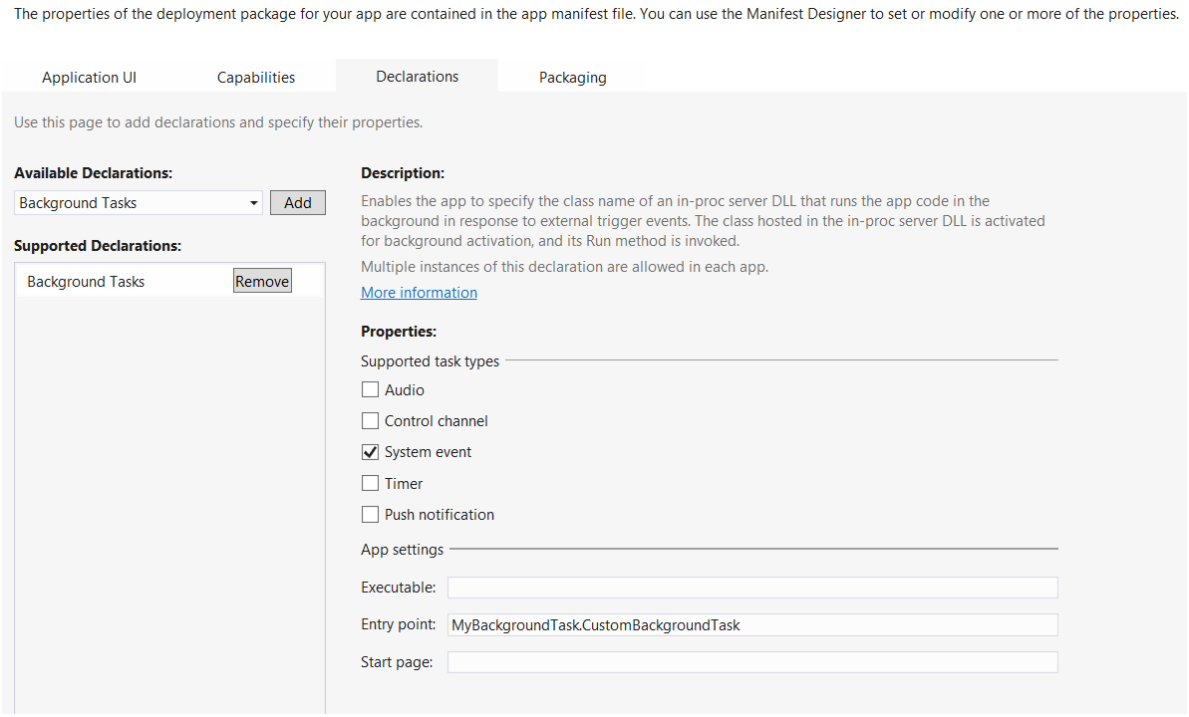
Következő lépésben fel kell vennünk az alkalmazásunkba egy hivatkozást erre az új Windows Runtime Component projektre, majd az alábbi kód segítségével regisztrálni a háttérfeladatot az alkalmazás indítása során:

```
var taskRegistered = false;
var exampleTaskName = "CustomBackgroundTask";

foreach ( var task in
          Windows.ApplicationModel.Background.BackgroundTaskRegistration.AllTasks )
{
    if ( task.Value.Name == exampleTaskName )
    {
        taskRegistered = true;
        break;
    }
}

if ( taskRegistered == false )
{
    var builder = new BackgroundTaskBuilder();

    builder.Name = exampleTaskName;
    builder.TaskEntryPoint = "MyBackgroundTask.CustomBackgroundTask";
```

```
    builder.SetTrigger( new SystemTrigger( SystemTriggerType.TimeZoneChange, false ) );
    builder.Register();
}
```

A kód név alapján megvizsgálja, hogy az adott feladat regisztrálva van-e már. Ha nem, akkor regisztrálja azt. A feladatra annak nevével, illetve a ***`Névtér.PontosObjektumNév`*** szintaktikával lehet hivatkozni. Miután megadjuk, hogy milyen feltétel esetén fusson le a fent implementált kódunk (időzóna megváltozása), egyszerűen a **`Register`** metódus hívásával beregisztráljuk.

Még egy apró lépés elválaszt bennünket a céltól. A **`Package.appxmanifest`** fájlon kettőt kattintva válasszuk a Declarations fület, majd a legördülő listából adjuk hozzá a Background Task bejegyzést! Meg kell adnunk a háttérfeladat típusát és annak belépési pontját a **`Névtér.PontosObjektumNév`** szintaktikával. A kódhoz tartozó beállításokat a 10-2 ábra mutatja be.

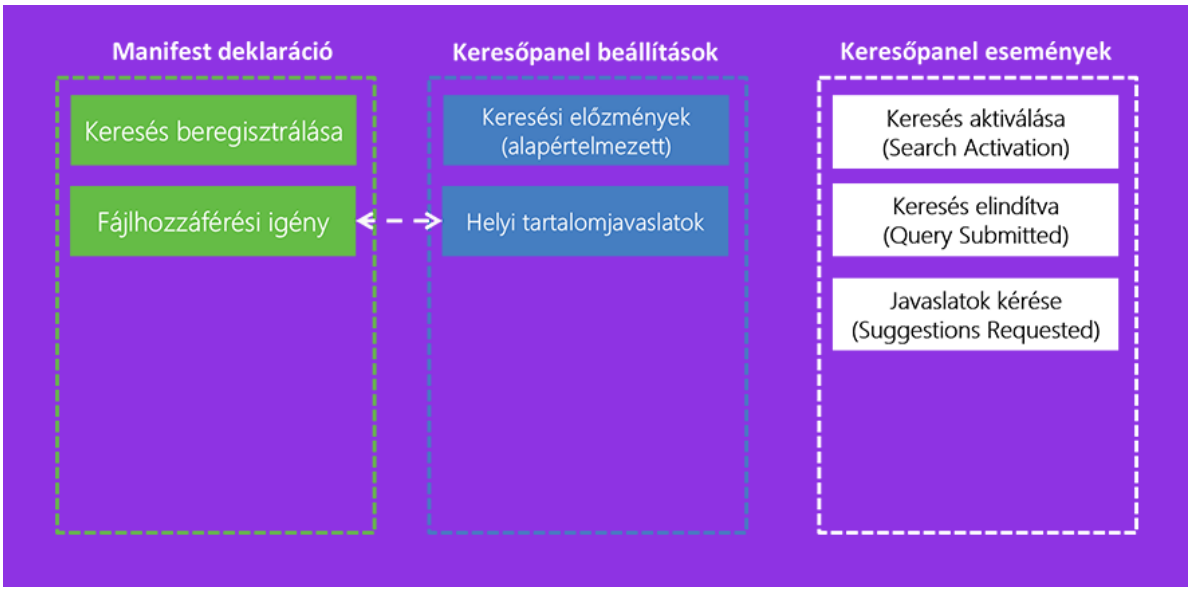


10-2 ábra: A Background Task konfigurációja

Integráció a keresővel (Search Contract)

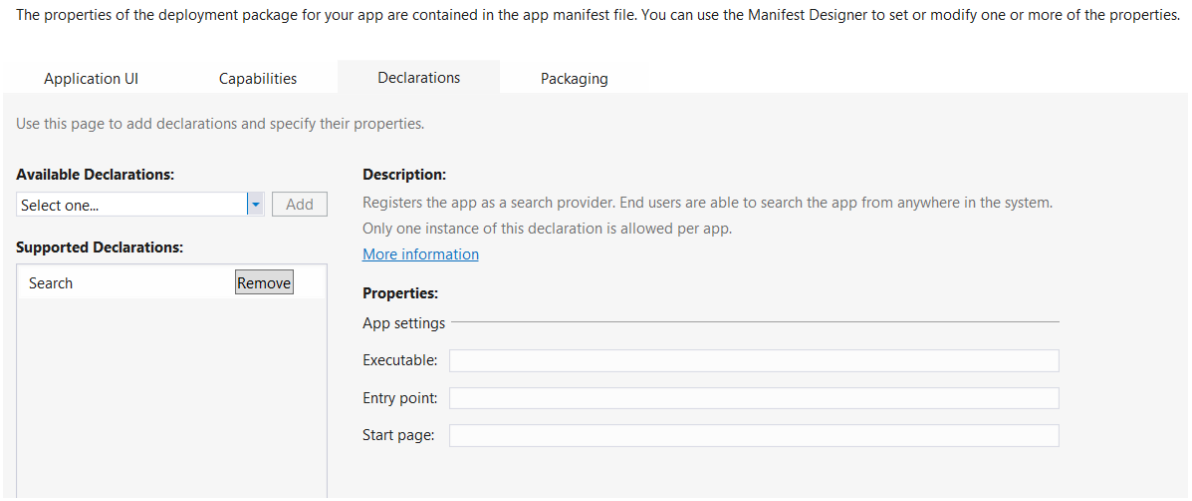
Amikor az egeret levisszük a képernyő jobb alsó vagy jobb felső sarkába, vagy a képernyő jobb széléről behúzás gesztusát használjuk, megjelenik a Charms bar. A sáv legfelső részében a keresőt találjuk, amely képes együttműködni alkalmazásunkkal – annak beállítása és egy kevés kódírás után. Többé nem szükséges saját felületeket és keresőmotorokat írnunk magunknak, vagy egy külső gyártótól származó kódkönyvtárat felhasználni, mivel az operációs rendszer magába foglalja ezt a funkciót is! A 10-3 ábráról leolvashatók a szükséges lépések, melyek az integrációhoz vezetnek, ha pedig az alkalmazás jelezte az operációs rendszernek, hogy a kereséshez adatokat szolgáltatni és feldolgozni is képes, akkor kezelheti a kereséshez kapcsolódó releváns eseményeket is.

A mintakódok kipróbálásához nem szükséges új projektet létrehoznod, nyugodtan használhatod valamelyik meglévőt.



10-3 ábra: Integráció a Windows 8 keresőjével és a kereső eseményei

A keresőintegrációt a *manifest* fájlon keresztül kell jeleznünk. Kattints duplán a **`Package.appxmanifest`** fájlra, majd menj a Declarations fülre! Az Available Declarations legördülő listából válaszd a Search elemet! Amint kiválasztottad, megjelenik a 10-4 ábrán látható felület, ahol további paramétereket adhatsz meg. Itt lehet definiálni a végrehajtási egységet, a belépési pontot és azt a felületet, amit a keresés indításkor meg szeretnél jeleníteni. Ebben az esetben ezeket nem szükséges kitölteni, zárd be a felületet!



10-4 ábra: A keresés konfigurációs lehetőségei

A keresés használatát az alábbi kódpélda mutatja be:

```
public sealed partial class MainPage : Page
{
    private SearchPane _searchPane;

    private static readonly string[] suggestionList =
        {
            "Szeged", "Debrecen", "Budapest", "Pécs", "Miskolc",
        };

    public MainPage()
    {
        this.InitializeComponent();
        _searchPane = SearchPane.GetForCurrentView();
    }
```

```
    protected override void OnNavigatedTo(NavigationEventArgs e)
    {
        _searchPane.SuggestionsRequested += _searchPane_SuggestionsRequested;
        _searchPane.QuerySubmitted += _searchPane_QuerySubmitted;
    }

    protected override void OnNavigatedFrom(NavigationEventArgs e)
    {
        _searchPane.SuggestionsRequested -= _searchPane_SuggestionsRequested;
        _searchPane.QuerySubmitted -= _searchPane_QuerySubmitted;
    }
}
```

Osztály szinten szükségünk lesz egy **SearchPane** típusú objektumpéldányra, valamint egy listára, amely a javaslatokat tartalmazza, így megkönnyítve a felhasználónak a keresést. A példában szereplő lista statikus, azonban ez éles helyzetben tartalmazhatna dinamikusan változó adatokat is, amelyek jöhetnek a helyi tárolóból, vagy akár egy webszolgáltatástól is eredhetnek. A keresőfelület nézetekhez rendelten külön-külön konfigurálható, így mi a **_searchPane** változónkat az alkalmazás adott **Page** objektumához rendeljük az inicializálás során.

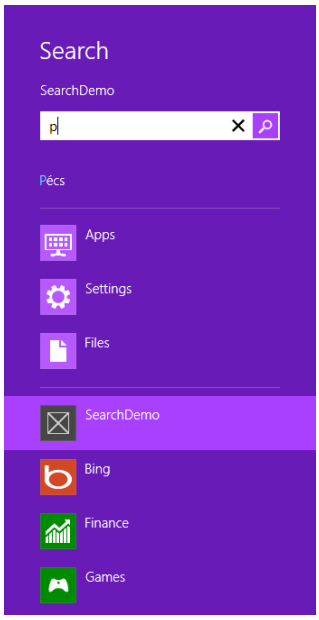
Két eseményt használunk fel. Az egyik azért felelős, hogy a keresőmezőbe való gépelés során a javaslatokkal könnyítse a keresést, a másik pedig a keresési kifejezés nyugtázásáért. Amikor az oldalra navigálunk, feliratkozunk az eseményekre, majd az oldalról való távozás során eltávolítjuk a hivatkozásainkat, ezzel az esetleges későbbi hibákat kiküszöbölve:

```
void _searchPane_SuggestionsRequested(SearchPane sender,
                                      SearchPaneSuggestionsRequestedEventArgs args)
{
    string query = args.QueryText;

    if ( string.IsNullOrEmpty( query ) )
    {
    }
    else
    {
        var request = args.Request;

        foreach ( string suggestion in suggestionList )
        {
            if ( suggestion.StartsWith( query,
                                        StringComparison.CurrentCultureIgnoreCase ) )
            {
                request.SearchSuggestionCollection.AppendQuerySuggestion( suggestion );
            }
        }
    }
}

async void _searchPane_QuerySubmitted(SearchPane sender,
                                      SearchPaneQuerySubmittedEventArgs args)
{
    var query = args.QueryText;
    await new MessageDialog( "A keresett kifejezés: " + query ).ShowAsync();
}
```

A javaslatok kibocsátása nagyon egyszerű! Megnézzük, hogy a felhasználó által a keresőmezőbe írt kifejezés értelmezhető-e, és ha igen, bejárjuk az osztályhoz tartozó statikus listát, és kiválogatjuk azokat az elemeket, melyek a begépelt kifejezéssel egyezést mutatnak. Az egyezés vizsgálata során nem foglalkozunk a kis- és nagybetűk megkülönböztetésével, továbbá a javaslatok maximális számát sem korlátoztuk, bár erre éles esetben szükség lehet. Ha egyezés volt, az adott elemet átadjuk a Windows 8 keresőjének.

A kifejezés kezelése még egyszerűbb feladat. A felhasználó nyugtázása után megkapjuk a kifejezést pontosan abban a formában, ahogyan azt begépelte. Ettől kezdve pedig a mi felelősségünk, hogyan használjuk azt fel.

A 10-5 ábra ezt az egyszerű keresést mutatja be működés közben.



10-5 ábra: A Windows 8 keresője az alkalmazásunkhoz igazítva

Integráció a beállítások panellel

A Charms bar tartogat számunkra még néhány hasznos képességet. Ahogyan a kereső esetében is jelentős könnyítés az egységesített környezet, úgy az alkalmazás beállításait is kivezethetjük egy hasonló felületre. Az első, rögtön szembetűnő különbség a kereséshez képest az, hogy a Settings gomb lenyomása után kiválasztva a megfelelő opciót nem rendelkezünk egy alapértelmezett megjelenéssel, az ide tartozó felületet magunknak kell létrehoznunk.

Mielőtt ennyire előreszaladnánk, tekintsük át, mire lesz szükségünk! A *Settings charm*ra az alkalmazás (vagy az adott alkalmazásfelület) betöltésekor fel kell iratkoznunk, létre kell hoznunk egy saját parancsot (*command*). Ha erre rákattint a felhasználó, a parancs végrehajtása során lehetőségünk van a saját felületünk megjelenítésére. Ennek a sávnak a szélessége két fix méretet vehet fel – 346 vagy 646 pixel –, így saját felületünknek is ehhez a méretezéshez kell idomulnia.

A mintakódokat nem feltétlenül szükséges új projektben implementálni, nyugodtan használhatjuk valamelyik előző projektünket is. Válasszunk egy meglévő Page objektumot, amelyhez beállításokat szeretnénk létrehozni!

```
public sealed partial class MainPage : Page
{
    private Popup _settingsPopup;

    public MainPage()
    {
        this.InitializeComponent();

        SettingsPane.GetForCurrentView().CommandsRequested += MainPage_CommandsRequested;
    }
}
```

Szükségünk van egy **Popup** típusú vezérlőre, ezen keresztül tudjuk majd a beállításokat tartalmazó panelt megjeleníteni illetve elrejteni. Ahogyan a neve is mutatja, ez a vezérlő semmilyen alapértelmezett

megjelenítési móddal nem rendelkezik, egyetlen feladata annak megjelenítése minden más felületelem előterében. Az osztály konstuktorában jelezzük a Windows 8 számára, hogy saját elemekkel szeretnénk bővíteni a Settings charmot:

```
void MainPage_CommandsRequested(SettingsPane sender, SettingsPaneCommandsRequestedEventArgs args)
{
    SettingsCommand command = new SettingsCommand( "sampleCommand",
        "Beállítások",
        ( x ) =>
            {
                _settingsPopup = new Popup();
                _settingsPopup.Width = 346;
                _settingsPopup.Height = Window.Current.Bounds.Height;
                _settingsPopup.IsLightDismissEnabled = true;

                _settingsPopup.Closed += _settingsPopup_Closed;
                Window.Current.Activated += Current_Activated;

                SettingsControl control = new SettingsControl();
                control.Width = 346;
                control.Height = Window.Current.Bounds.Height;
                _settingsPopup.SetValue( Canvas.LeftProperty,
                                         Window.Current.Bounds.Width - 346 );
                _settingsPopup.SetValue( Canvas.TopProperty, 0 );
                _settingsPopup.Child = control;
                _settingsPopup.IsOpen = true;
            } );

    args.Request.ApplicationCommands.Add( command );
}
```

Létrehozunk **`SettingsCommand`** típusú változót, majd az inicializálása során megadjuk az azonosítóját és a felületen megjelenő elnevezését. A harmadik paraméter egy metódus, amely lefut, amikor a felhasználó kiválasztja az adott parancselemet. Ennek a lambda kifejezéssel megadott metódusnak a tartalma hosszúnak tűnhet első pillantásra, de az nem tartalmaz semmilyen bonyolult tevékenységet. Létrehozza a **`Popup`** objektumpéldányt, majd a szélességét 346 pixelre állítja be. A felület magassága mindig akkora lesz, mint az ablak teljes magassága (az ablak paramétereit a **`Window.Current.Bounds`** objektum tartalmazza). A felület bezárására több lehetőségünk is van. Az **`IsLightDismissEnabled`** tulajdonság **`true`** értékre állításával a felhasználó a felületen kívülre kattintva a bezárási igényét fejezi ki.

Figyelnünk kell arra az eseményre is, amikor a felület bezáródott, ezzel későbbi hibákat előzhetünk meg! Mivel nincs még a **`Popup`** vezérlőnknek megjeleníthető tartalma, így hozzunk létre egy új **`SettingsControl`** objektumpéldányt (nincs ugyan még ilyen objektumunk, de később azt is elkészítjük)! A beállításokhoz nem tartozik alapértelmezett megjelenés, így nekünk kell egy sajátot létrehozni. Az újonnan létrehozott vezérlő méretezését is állítsuk be, majd jelenítsük meg a **`Popup`** vezérlőt az **`IsOpen`** tulajdonság igaz értékűre állításával!

A létrehozott parancsot (**`command`**) hozzáadjuk a Charms bar parancsokat tartalmazó konténeréhez.

Ha a felületünket bezárták, vagy akár az egész alkalmazásfelületről elnavigáltak, szükséges lehet a beállítások panel bezárása, illetve a többé nem használt eseményvezérlők eltávolítása, így több hibát is megelőzhetünk:

```
void Current_Activated(object sender, Windows.UI.Core.WindowActivatedEventArgs e)
{
    if (e.WindowActivationState == Windows.UI.Core.CoreWindowActivationState.Deactivated)
    {
        _settingsPopup.IsOpen = false;
    }
}

void _settingsPopup_Closed(object sender, object e)
```

```
{
    Window.Current.Activated -= Current_Activated;
}
```

Mivel eddig nem volt felületünk, amit használhattunk volna, hozzunk létre egy új **UserControl**t **SettingsControl** néven. Ennek XAML tartalma legyen a következő:

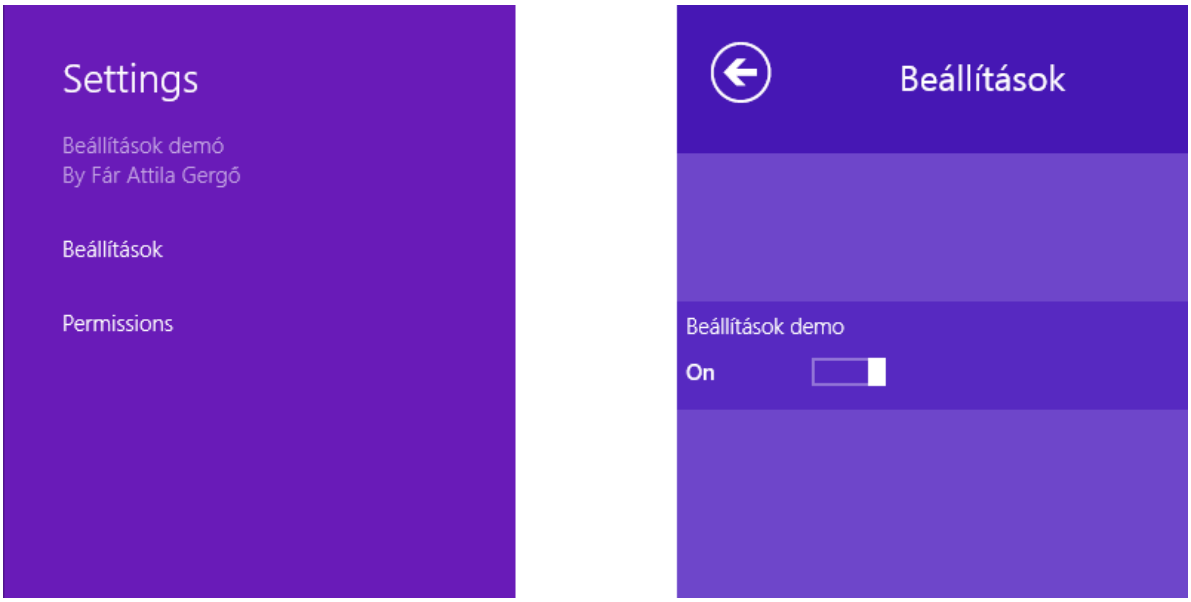
```
<Grid x:Name="layoutRoot"
        Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
    <Grid.RowDefinitions>
        <RowDefinition Height="100" />
        <RowDefinition Height="*" />
    </Grid.RowDefinitions>
    <Grid x:Name="header"
            Background="{StaticResource ComboBoxItemSelectedBackgroundThemeBrush}">
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="Auto"/>
            <ColumnDefinition/>
        </Grid.ColumnDefinitions>
        <Button x:Name="backButton"
            Content="Back"
            HorizontalAlignment="Center"
            Margin="20,0,0,0"
            Style="{StaticResource BackButtonStyle}"
            VerticalAlignment="Center"
            Width="Auto" Height="Auto"
            Foreground="{StaticResource TextBoxForegroundThemeBrush}"
            Background="{StaticResource ButtonBackgroundThemeBrush}"
            Click="backButton_Click"/>
        <TextBlock HorizontalAlignment="Center"
            Text="Beállítások"
            VerticalAlignment="Center"
            Grid.Column="1"
            FontSize="24"/>
    </Grid>
    <Grid x:Name="content" Grid.Row="1"
        Background="{StaticResource ToggleSwitchCurtainPointerOverBackgroundThemeBrush}">
        <ToggleSwitch Header="Beállítások demo"
            HorizontalAlignment="Stretch"
            VerticalAlignment="Top" IsOn="True"
            Margin="0,100,0,0"
            Background="{StaticResource ToggleSwitchCurtainBackgroundThemeBrush}"/>
    </Grid>
</Grid>
```

A mintakódban látható elemek csak demonstrációs célt szolgálnak, valódi funkcionalitásuk nincs ebben a példában. A felületen van egy visszalépés gomb is, amelyet első érzésünk alapján nem tudunk egyszerűen megvalósítani, mivel a gomb nem tudhatja, melyik az a **Popup** példány, amit be szeretnénk vele zárni. Azonban van egy egyszerű kerülőmegoldás, amellyel a külső szülőelem lekérdezhető, a hivatkozáson keresztül pedig az **IsOpen** tulajdonság megfelelően állítható:

```
private void backButton_Click(object sender, RoutedEventArgs e)
{
    if ( this.Parent.GetType() == typeof(Popup) )
    {
        ((Popup)this.Parent).IsOpen = false;
    }
}
```

A példában szereplő megvalósítás az esetek jelentős részében jól használható, saját alkalmazásunkba való beillesztése egyszerű, továbbá az MVVM tervezési minta által előírt szabályokkal is könnyedén összebékíthető.

A mintapélda eredményeként előálló beállításpanelt a 10-6 ábra mutatja.



10-6: Saját beállításpanelünk működés közben

Pozíció meghatározása szenzorokkal

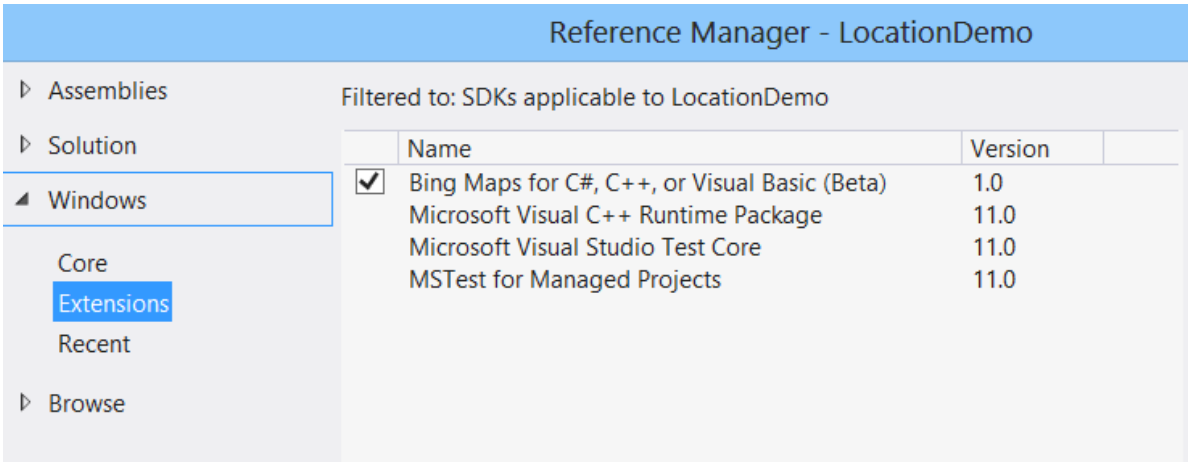
A Windows 8 különböző hardveres eszközöket használ a földrajzi helyzet meghatározásához, és ezeket más-más típusú platformszolgáltatásokkal egészíti ki. A fejlesztők számára a teljes folyamat transzparens és néhány egyszerű objektum használatára korlátozódik. Ebben a példában a helymeghatározást kiegészítjük a Bing térképszolgáltatással is, amely ugyan nem része a Windows Runtime-nak, azonban a Microsofttól származó hivatalos SDK formájában elérhető.

A könyv írásának pillanatában még nem jelent meg a végleges változat, azonban a letölthető előzetes állapotú könyvtárcsomag tökéletesen együttműködik a Windows 8 és a Visual Studio 2012 végleges változataival.

Mielőtt nekilátnánk a példa értelmezésének, szükségünk lesz az említett SDK letöltésére és telepítésére. A csomag az alábbi címről tölthető le: http://tinyurl.com/bingmapsmetrosdk

A telepítéssel az előkészületeknek még nincsen vége. Ahhoz, hogy ténylegesen használni is tudjuk, szükségünk lesz egy Bing fejlesztői kulcsra. Ezt a www.bingmapsportal.com oldalon tudjuk igényelni. Lépjünk be Microsoft Account fiókunkkal, majd hozzunk létre egy új kulcsot, amelynek típusa legyen *Trial*!

Ha feltelepítettük az SDK-t, és megvan a kulcs is, akkor egyetlen lépésre még szükség van a kódírás megkezdése előtt. Fel kell vennünk az SDK elemeire a megfelelő assembly referenciát, ahogyan az a 10-7 ábrán is látható.



10-7 ábra: A Bing Maps SDK-hoz tartozó referenciák hozzáadása a projekthez

A térképszolgáltatás használatához az alábbi kódban látható módosításokat kell hozzáadnunk ahhoz a felülethez, ahol a térképet meg akarjuk jeleníteni:

```
<Page
    x:Class="LocationDemo.MainPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:LocationDemo"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    xmlns:Maps="using:Bing.Maps"
    mc:Ignorable="d">

    <Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
        <Border BorderBrush="White"
                BorderThickness="10"
                CornerRadius="10"
                Width="800" Height="450">
            <Maps:Map x:Name="map"
                      Credentials="[Az igényelt Bing kulcs]"
                      MapType="Aerial"/>
        </Border>
    </Grid>
</Page>
```

A fenti kód egy nagyon egyszerű felületet definiál, amely nem tartalmaz mást, mint egy Bing SDK-ból származó **Map** vezérlőt és a hozzá tartozó kiemelést.

Nagyon fontos, hogy először mutassunk rá a megfelelő assembly-re, és csak utána fogjuk látni a **Map** vezérlőt, a **Maps** előtaggal ellátott névtérhivatkozáson belül. A vezérlőpéldány kapott egy nevet (map), amellyel mögöttes kódból tudunk rá hivatkozni. Ahhoz, hogy a térképet használhassuk, a vezérlőnek még át kell adnunk a portálról kapott fejlesztői kulcsot is. A vezérlő használata a mögöttes kódból igen egyszerű:

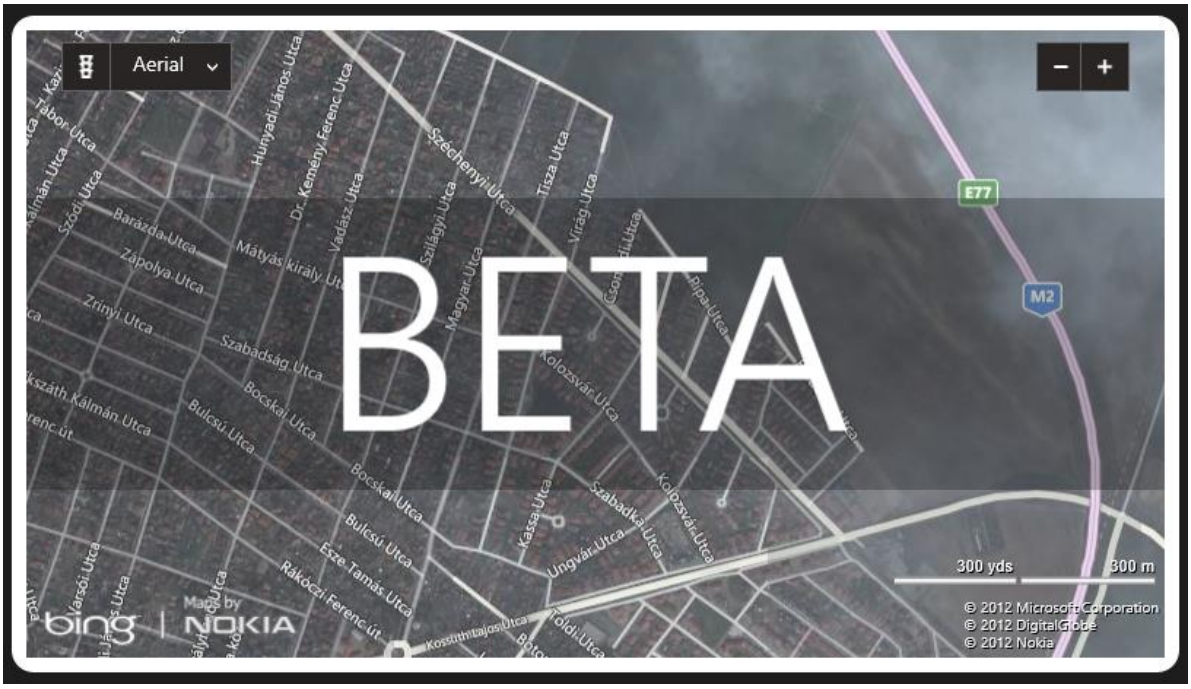
```
public sealed partial class MainPage : Page
{
    public MainPage()
    {
        this.InitializeComponent();
        this.Loaded += MainPage_Loaded;
    }

    void MainPage_Loaded(object sender, RoutedEventArgs e)
    {
        GetCurrentLocation();
```

```
    }

    private async void GetCurrentLocation()
    {
        Geolocator geolocator = new Geolocator();
        var position = await geolocator.GetGeopositionAsync();
        Location location = new Location( position.Coordinate.Latitude,
                                          position.Coordinate.Longitude );

        map.SetView( location, 15.0f );
    }
}
```

Itt megvárjuk, hogy az adott oldal teljesen betöltött állapotba kerüljön, majd a **`GetCurrentLocation()`** metódus segítségével lekérdezzük az aktuális földrajzi helyünket, és a térképet az adott pontra állítjuk. A lekérdezés – ahogyan már megszokhattuk különböző szolgáltatások használata során – teljes mértékben aszinkron módon történik. A Windows 8 **`Geolocator`** objektumát hívjuk segítségül, amely pontos működésének ismerete nélkül is könnyedén használható, a **`GetGeopositionAsync()`** metódus a szenzorok és szolgáltatások által meghatározott adatokat szolgáltatja vissza.

A térképen való megjelenítéshez a szélességi és a hosszúsági fokokra van szükségünk, melyeket egy **`Location`** típusú, Bing SDK-ból származó objektum segítségével adunk át a felületen található Map vezérlőnek.

Az alkalmazást működés közben a 10-8 ábra mutatja be.



10-8 ábra: A térképalkalmazás működés közben

Összegzés

Ebben a fejezetben kiegészítettük a Windows 8 platformszolgáltatásaihoz kapcsolódó ismereteinket. Megtanultuk, hogyan lehet az új életciklus-modellt szem előtt tartva folyamatainkat a háttérben futtatni. Kiegészítettük a programunkat, hogy az együttműködjön a Windows 8 beépített keresőjével, illetve a Charms bar beállításokért felelős felületével. Ezek után a szenzorok (illetve szenzorcsoportok) kezelésére néztünk meg egy újabb példát, amely lekérdezte aktuális helyzetünket, és a bemért adatok függvényében megmutatta pozíciónkat a térképen.

11. Webes szolgáltatások használata a Windows 8 alkalmazásokban

Ebben a fejezetben az alábbi témákat ismerheted meg:

- Webszolgáltatások elérése és használata, azaz hogyan kommunikálhat az alkalmazás egy szerverrel?
- Bevezetés a Live SDK használatába: hozzáférés a felhasználó nevéhez, képéhez, naptárához stb. egy igazán testreszabott alkalmazás megírásához
- Valós idejű kommunikáció: hogyan maradhat az alkalmazás akkor is naprakész, amikor nem fut?
- Felhő-integráció: a Windows Azure kiaknázása egy Windows 8 alkalmazásból
- Az OData protokoll: a Windows 8 alkalmazások és az adateléréshez kitalált nyílt szabvány együttműködése

Bevezetés

A Windows 8 alkalmazások és a már többéves múlttal rendelkező telefonalkalmazások üzleti modellje, felhasználói bázisa között sok hasonlóság figyelhető meg, ezért egy Windows 8 alkalmazás készítőjének érdemes tanulnia a telefonalkalmazások tapasztalataiból. Megfigyelhető közöttük például, hogy a siker egyik kulcsfontosságú eleme az alkalmazás tartalmának folyamatos naprakészen tartása. Gyakran az alkalmazás és a hardver, amin fut, csak másodhegedűsök, a felhasználót valójában az alkalmazáson keresztül elérhető tartalom érdekli.

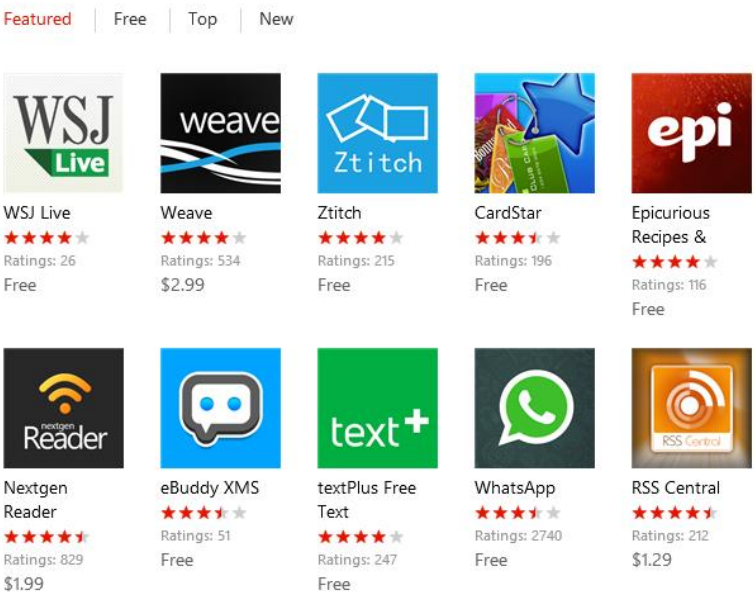
Jó példát szolgáltatnak erre a Windows Phone piacterén sikeres alkalmazások.

A 11-1 ábrán látható a Windows Phone piactér 2012. május 30-án kiemelt alkalmazásainak a listája. Ez tíz alkalmazást tartalmaz. Ebből négy online tartalom olvasására szolgál (WSJ Live, Weave, Nextgen Reader, RSS Central), három SMS- és azonnali üzenetküldő (textPlus, eBuddy XMS, WhatsApp), egy internetes adatbázisra épül (Epicurious Recipes – 30 ezer recept különféle forrásokból), és mindössze kettő támaszkodik elsősorban a telefon erőforrásaira (az akcióskupon-gyűjtő CardStar és a panorámakép-gyártó Ztitch, de természetesen ezek is hangsúlyos közösségi és online funkciókkal rendelkeznek).

Tehát a tízből nyolc alkalmazás internetes forrásoknak köszönheti interaktivitását, frissességét! Ez ugyan egy kiragadott példa volt, de jól érzékelteti a trendet, ami várhatóan a sikeres Windows 8 stílusú alkalmazásokra is jellemző lesz – hiszen miért várna a felhasználó kevesebb interaktivitást, online kapcsolatot táblagépétől, mint telefonjától? Jól érezhető tehát, hogy a Windows 8 kommunikációs lehetőségeinek ismerete alapvető eleme egy sikeres alkalmazásfejlesztő eszköztárának.

Ebben a fejezetben az interneten keresztül történő kommunikáció, illetve a felhő eszközeit ismerheted meg, amelyekkel a fent bemutatottakhoz hasonló alkalmazásokat készíthetsz.

A Windows 8 platform számos olyan helyen is tartalmaz internetes és „felhős" képességeket, ahol a fejlesztő csak közvetetten fér hozzájuk. Ilyen például a **`FilePicker`** vezérlőelem, amin keresztül a felhasználó a SkyDrive-ban lévő fájljai közül is tud válogatni, vagy a roaming alkalmazásadatok, amivel a felhasználó beállításai (szintén a felhő segítségével) szinkronizálódnak különféle számítógépei között. Ezekről a könyv más fejezeteiben esik szó.



11-1 ábra: A Windows Phone piactér kiemelt alkalmazásai 2012. május 30-án

Webszolgáltatások használata

A webszolgáltatások működése

Egy webszolgáltatás nem más, mint egy szerveren lévő függvény, amit az interneten keresztül távolról is meghívhatsz. Például lehet a szervereden egy **`GetNews()`** művelet, amit alkalmazásod a felhasználó saját számítógépéről meghív, és ekkor a Windows 8 az interneten keresztül kérést küld a szervernek. A szerver összeállítja az aktuális híreket, az interneten keresztül visszaküldi, az alkalmazáskód pedig megkapja, és tetszés szerint felhasználja őket.

Webszolgáltatások segítségével bármilyen kommunikáció lebonyolítható egy kliens és egy szerver között: egy sakkjáték lépéseinek küldözgetésétől kezdve egy költségnyilvántartó program központi adatbázisának frissítéséig.

Egy webszolgáltatás működéséhez persze egy sor technológia kell! Ezek fordítják le például a szerverhez beérkező adatcsomagokat a szerveroldali műveletet futtató programnyelv számára értelmezhető formára, és ezek küldik aztán vissza a művelet futási eredményét a szolgáltatás meghívójának.

A .NET-világban erre a Windows Communication Foundation (WCF) használható, de minden más szerveroldali technológia (pl. Java, PHP) is ad lehetőséget webszolgáltatások készítésére. Bármelyiket is használjuk, a fejlesztés menete hasonló: a programozó először is eldönti, hogy milyen műveleteket szeretne kívülről meghívhatóvá tenni, majd megírja a webszolgáltatás kódját. A szerveroldali futtatókörnyezet pedig ez alapján egy XML formátumú leíró fájlt gyárt, amelyben rögzíti, hogy a szerveren milyen kívülről hívható műveletek érhetők el.

Ezután a kliensoldal (azaz a felhasználó gépén futó alkalmazás) fejlesztése során a fejlesztő megadja ezt a leíró fájlt a fejlesztőkörnyezetnek. A fejlesztőkörnyezet ebből egy úgynevezett *proxy* osztályt generál, azaz itt lesznek a leíró fájl XML paramétereiből a kliensoldali alkalmazásban is meghívható metódusok. Ha az alkalmazás valamelyiket meghívja, akkor a proxy osztály generált kódja elküldi a kérést az interneten keresztül, és befogadja a szervertől kapott választ.

Mindez azt jelenti, hogy a webszolgáltatások komplex internetes kódjával egyáltalán nem kell bajlódni. A szerveroldali leírófájl és a kliensoldali proxy osztály generálása automatikusan történik, a fejlesztőnek csak a tényleges szerveroldali és kliensoldali kódot kell megírnia – az oda-vissza fordítást elvégzi helyette a fejlesztőkörnyezet.

Szinkron és aszinkron hívások

Ahogy a fentiekből látható, egy webszolgáltatás meghívásakor a háttérben valójában két művelet történik. A proxy osztály először küld egy kérést a szerver felé, majd – miután némi várakozás után megjött a válasz – lefordítja a válaszüzenetet a program számára érthető formára. A kliensoldali kód számára mindez tűnhet *szinkron* vagy *aszinkron* műveletnek.

Ha a proxy osztály *szinkron* működésű, akkor a két háttérműveletet egy egységként kezeli, azaz miután meghívódott a proxy osztály valamelyik metódusa, a program addig nem is halad tovább, amíg a háttérben történő műveletek le nem zajlottak. Ez fejlesztői szempontból egyszerű, ám a felhasználói élményt ronthatja. A webszolgáltatás meghívása és válasza között (akár a szerver terheltsége, akár az internetkapcsolat lassúsága miatt) ugyanis néha másodpercek is eltelhetnek, a program pedig közben lefagyottnak tűnhet, hiszen egy bizonyos programsornál (a webszolgáltatás meghívása) álldogál a válaszra várva.

Ezt a várakozást elkerülendő Windows 8 alkalmazásunkban nem használhatunk szinkron proxy osztályokat. Amikor szerveroldali webszolgáltatásunkból létrehozzuk a kliensoldali proxy osztályt („felvesszük a referenciát" a szolgáltatásra), kizárólag aszinkron kódot generáltathatunk. Az aszinkron kódnál a két művelet elkülönül: miután meghívtuk a webszolgáltatást, alkalmazásunk nem várakozik annál az egy kódsornál, hanem a válasz megérkezéséig más dolgokat is csinálhat (például reagálhat a felhasználó mozdulataira, az animációk tovább futhatnak stb.), azaz nem tűnik majd lefagyottnak. Amint a válasz megjön, alkalmazásunk erről értesítést kap, és folytathatja az elkezdett műveletsort.

A Windows 8 (és a .NET Framework 4.5) előtti időkben egy ilyen aszinkron hívás ugyan barátságosabb volt, de a fejlesztő számára kényelmetlen volt használni, mert a fenti elválasztás miatt kétfelé kellett vágnia a kódját – egy eseménykezelőt kellett írnia a szervertől érkező válasz kezelésére, ami a programkódban egy külön metódus, így logikailag egybetartozó kódrészleteket önkényesen szét kellett vagdosni. A C# nyelv új verziójába azonban bekerültek az **`async`** és **`await`** kulcsszavak, amelyekkel megmarad az aszinkron hívások előnye, de a szinkron hívásokhoz hasonlóan a kódot sem kell szétszabdalni. A fejezet példakódjaiban így ezzel a kényelmes új lehetőséggel írunk majd webszolgáltatásokat; a kulcsszavakról bővebben a könyv 4. fejezetében olvashatsz.

Webszolgáltatás egy mintaalkalmazásban

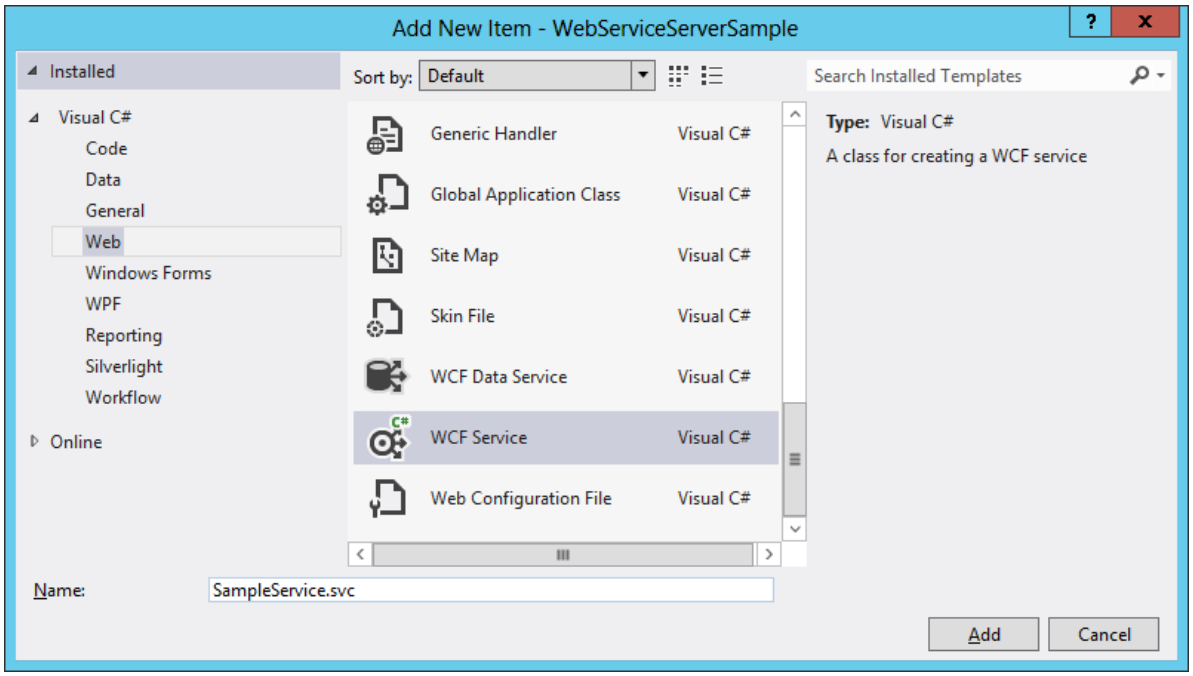
Írjunk egy Windows 8 alkalmazást, ami egy webszolgáltatást hív meg! Ez egyszerű fordító szolgáltatás lesz: a Windows 8 alkalmazás felküld egy szót magyarul, a webszolgáltatás pedig visszaküldi azt angolul. Bemutató alkalmazásról lévén szó, a „szótár" mindössze két szót tartalmaz majd. A mintaalkalmazásnak egy szerveroldali és egy kliensoldali komponense lesz.

Ha a Visual Studio 2012 Express for Windows 8 van a gépeden a Windows 8 fejlesztéshez, akkor szükséged lesz az ingyenes Visual Studio 2012 Express for Web változatra is. Ha viszont a Visual Studio valamelyik fizetős verzióját használod, akkor az alkalmas lesz mindkét fejlesztés elvégzésére.

Gyakorlat: *Webszolgáltatást használó Windows 8 alkalmazás fejlesztése*

Az alkalmazás létrehozásához kövesd az alábbi lépéseket:

1. Indítsd el a Visual Studiót, és hozz létre egy ASP.NET Empty Web Application típusú új projektet, amit nevezz el **`WebServiceServerSample`**-nek! Ez a projekt lesz az alkalmazás szerveroldali komponense.
2. A projekt létrejöttekor üres. Hozz létre benne egy új webszolgáltatást! Ehhez vegyél fel a projektbe egy WCF Service típusú elemet **`SampleService`** néven!A listában (lásd 11-2 ábra) találhatsz egy egyszerűen Web Service-nek nevezett elemet is. Ez egy ASMX kiterjesztésű fájlt hozna létre, és egy előző generációs technológiát képvisel. A mintaalkalmazásban a WCF technológiával készül a webszolgáltatás, ami az aktuális webszolgáltatás-fejlesztő eszköz a .NET Frameworkön belül.



11-2 ábra: A webszolgáltatás hozzáadása

A szolgáltatás hozzáadása után 3 elem is létrejön projektedben: 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 megadnod a kiajánlani kívánt műveleteket. 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. A **`SampleService.svc.cs`**-be pedig az **`ISampleService.cs`**-ben deklarált metódusok tényleges kódja kerül. (Az **`ISampleService.cs`** nem létfontosságú, nélküle is működhet a szolgáltatás, de itt nem tárgyalt WCF konfigurációs okok miatt célszerű használni.)

A fordításhoz szükséges metódus egy **`string`** változót vár bemeneti paraméterként (ez a magyar szó), és egy **`string`** változót is ad vissza (ez az angol megfelelő). Ahogy az előző bekezdésben olvashattad, ennek megírásához először deklarálnod kell a metódust az **`ISampleService.cs`** interfészben.

3. A fájlt megnyitva láthatod majd, hogy egy **`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). Nevezd át ezt a metódust **`Translate`**-re, és add hozzá a kimeneti, illetve bemeneti paramétereket!

```
[OperationContract]
string Translate(string input);
```

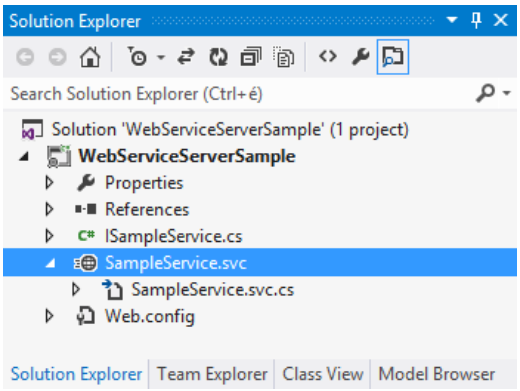
4. Most írd meg a metódushoz tartozó kódot! Ehhez nyisd meg a **`SampleService.svc.cs`** fájlt! Itt szintén látod majd a **`DoWork()`** metódust (ezúttal már attribútum nélkül). Töröld ki, és helyettesítsd az alábbi kóddal!

```
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 készültél.

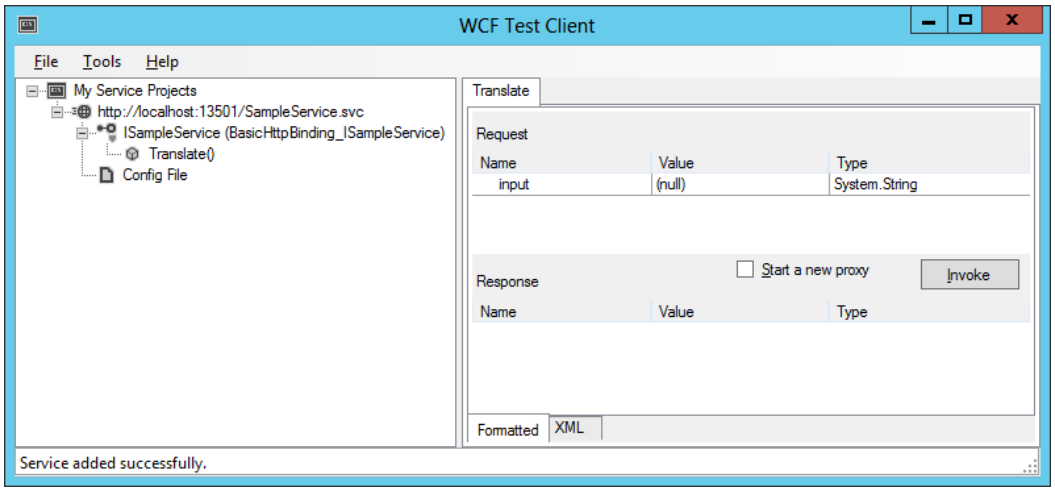
5. Teszteld a működését! Jelöld ki a Visual Studio Solution Explorer ablakában a **`SampleService.svc`** kódelemet (lásd 11-3 ábra), és nyomd meg az F5 gombot!



11-3 ábra: A Solution Explorerben kijelölt `SampleService.svc`

A Visual Studio érzékeli, hogy egy webszolgáltatást jelöltél ki futtatásra, és megnyitja a webszolgáltatások tesztelésére használható WCF Test Client eszközt (lásd 11-4 ábra). Ez univerzális eszköz, mellyel egyszerű felületen keresztül kéréseket küldhetsz bármilyen webszolgáltatásnak, és megnézheted a szolgáltatás válaszát – így anélkül tudod tesztelni a webszolgáltatást, hogy ténylegesen bajlódnod kellene egy kliens megírásával.

> *Ha nem a WCF Test Client nyílt meg, akkor győződj meg róla, hogy a **`SampleService.svc`**-t kijelölve nyomtad meg az F5 gombot!*



11-4 ábra: A WCF Test Client

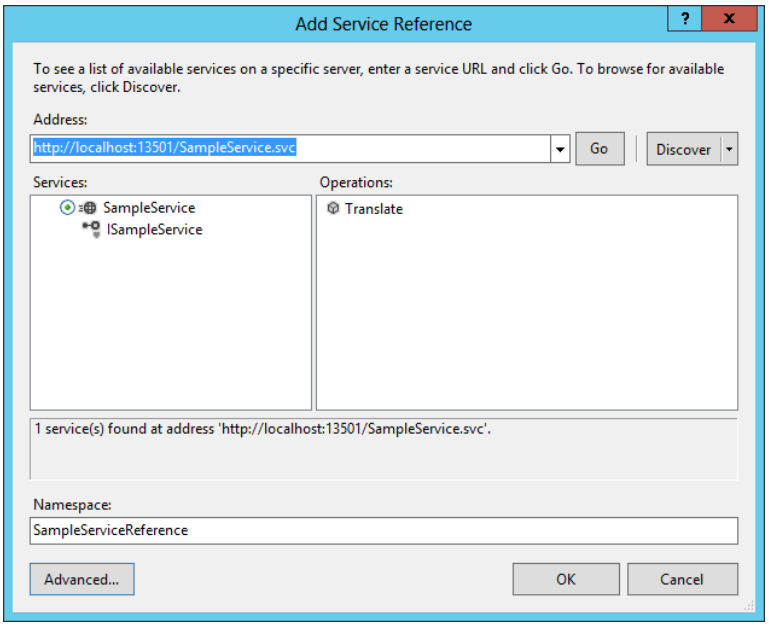
6. A szolgáltatás teszteléséhez kattints duplán a bal oldali fában látható **`Translate()`** metódusra, majd a jobb oldalon lévő szerkesztőfelület „input” sorának „Value” oszlopába írd be a „szilva” értéket, és kattints az Invoke gombra! A WCF Test Client elküldi a kérést, és lent, a Response ablakban kiírja a választ – „plum”, ha minden jól ment.

Miután ezzel megvagy, kattints jobb gombbal a szolgáltatás címére a bal oldali fában (ez a fenti ábrán http://localhost:13501/SampleService.svc, de a portszám, azaz a „13501” érték mindenhol más és más lesz), és a Copy Address menüpont segítségével mentsd el ezt a címet! Később még szükség lesz rá!

> *A fent leírtak gyakorlatilag változtatás nélkül működnek akkor is, ha a webszolgáltatásodat Windows Azure-ból futtatod.*

Most térjünk át a kliensoldalra, és írjuk meg a szervert meghívó Windows 8 alkalmazást!

7. Hozz létre egy új Visual Studio projektet (az eddig futó Visual Studio példányt *ne* zárd be, mert akkor leáll a webszolgáltatás szervere is, erre pedig rövidesen szükség lesz – lehetőség szerint nyiss egy új Visual Studio példányt)! A projekt sablonja legyen egy Blank App típusú Windows 8 stílusú alkalmazás, neve pedig **`WebServiceClientSample`**!
8. Ahhoz, hogy használhasd a webszolgáltatást, fel kell rá venned egy referenciát. Ezen művelet során olvassa majd be a Visual Studio a szerver által publikált leíró állományt, és generálja ki a bevezetőben említett proxy osztályt. Referencia felvételéhez kattints jobb gombbal a projekt nevére (**`WebServiceClientSample`**) a Solution Explorer ablakban, és válaszd az Add Service Reference menüpontot!
9. A megjelenő párbeszédpanel tetejére írd be a korábban kimásolt URL-t, majd kattints a Go gombra! (Fontos, hogy a webes projektnek még futnia kell!) A Visual Studio letölti és feldolgozza a megcélzott webszolgáltatás adatait, majd a párbeszédpanel közepén található területen megjeleníti az elérhető műveleteket (lásd 11-5 ábra). Jelenleg egy műveletet tartalmaz a webszolgáltatás: az előbb definiált **`Translate()`** metódust. Adj nevet a referenciának (**`SampleServiceReference`**), majd kattints az OK gombra!



11-5 ábra: Az Add Service Reference dialógus

A Visual Studio legenerálja a proxy osztályt, és létrehoz neki egy új bejegyzést a Solution Explorer ablakban, a Service References mappában.

Ha később módosítani szeretnéd a webszolgáltatás címét (például az alkalmazás publikálásakor), akkor jobb gombbal kattints erre az elemre, és a Configure Service Reference menüpont segítségével módosíthatod a szolgáltatás címét. Ha pedig webszolgáltatásod maga változik (például új metódusokat adsz hozzá), akkor a proxy osztályt kell frissítened. Ehhez kattints jobb gombbal a létrejött referenciára, és válaszd az Update Service Reference menüpontot!

10. Következő lépésként hozd létre a program felhasználói felületét! Ehhez nyisd meg a **`MainPage.xaml`** fájlt, és helyezz ki egy szövegdobozt (**`TextBox`**) és egy gombot (**`Button`**) a

tervezőfelületre! A szövegdoboz neve legyen **MessageTextBox**, a gomb neve pedig **TranslateButton**!

11. Az igazán elegáns megoldás érdekében ezután helyezz el egy **ProgressBar**t is (ez az a vezérlőelem, ami „csíkot húzva" mutatja, ha a számítógép dolgozik valamin)! Ennek a neve legyen **MainProgressBar**, állítsd az **IsIndeterminate** tulajdonságát **true** értékre, a **Visibility** tulajdonsága pedig legyen **Collapsed**! Ezzel létrehoztál egy olyan **ProgressBar**t, ami alaphelyzetben nem látszik, és határozatlan ideig jelzi egy folyamat előrehaladását – hiszen egy webszolgáltatás meghívásánál nem tudjuk megmondani, hogy a folyamat épp hány százalékánál jár.
12. Most kattints duplán a **TranslateButton** gombra, hogy belépj a gomb **Click** eseménykezelőjébe! Az eseménykezelőnek a következő feladatai vannak: be kell kapcsolnia a **ProgressBar**t, hogy a felhasználó lássa, valami folyamatban van; meg kell hívnia a webszolgáltatást; a válasz megérkezése után pedig ki kell kapcsolnia a **ProgressBar**t (hiszen véget ért, amire a felhasználó várt), és meg kell jelenítenie a választ.

Ez remek lehetőség a C# új **async** és **await** kulcsszavainak alkalmazásához! Ahogy a fejezetben már volt róla szó, ezek segítségével a webszolgáltatás aszinkron módon hívható, de mégsem kell eseményekre feliratkozni vagy szétvágni a kódot két részre. Mindössze el kell helyezni az eseménykezelő metódus elejére az **async** kulcsszót, a webszolgáltatás-hívás elé az **await** kulcsszót, és a többi már a C# fordító dolga.

A kódnak ezért így kell kinéznie:

```
async private void TranslateButton_Click(object sender, RoutedEventArgs e)
{
    MainProgressBar.Visibility = Visibility.Visible;

    SampleServiceReference.SampleServiceClient client = new
SampleServiceReference.SampleServiceClient();
    string result = await client.TranslateAsync(MessageTextBox.Text);

    MainProgressBar.Visibility = Visibility.Collapsed;

    Windows.UI.Popups.MessageDialog dialog = new Windows.UI.Popups.MessageDialog(result);
    dialog.ShowAsync();
}
```

Ezzel elkészült a mintaalkalmazás.

13. Teszteld az alkalmazást! Győződj meg róla, hogy a webszolgáltatás még mindig fut-e, majd indítsd be a Windows 8 alkalmazást! Írj be egy magyar szót (legyen „alma" vagy „szilva"), majd nyomd meg az Elküldés gombot! Látni fogod, hogy az alkalmazás elindítja a kérést, megjeleníti a **ProgressBar**t, majd kisvártatva megkapja és megjeleníti a választ (lásd 11-6 ábra).

11-6 ábra: A végrehajtott webszolgáltatás-hívás eredménye

Hogyan működik?

Az elsőként megírt webalkalmazás kiajánl egy webszolgáltatást, amit ha kívülről meghívnak, akkor a szerver elvégzi a kapcsolódó műveletet és visszajuttatja a klienshez az eredményt. A másodikként megírt Windows 8 alkalmazás erre a webszolgáltatásra tartalmaz egy referenciát, ami a fejlesztő számára lehetővé teszi, hogy C# kódból egyszerűen használja a szolgáltatást. Majd az alkalmazás néhány sor kódból ténylegesen meg is hívja azt, aszinkron módon.

Ezzel megismerted egy új webszolgáltatás létrehozásának módját. A webszolgáltatások segítségével lényegében tetszőleges kommunikációt bonyolíthatsz le egy Windows 8 gép és egy szerver között. Nem csak a primitív adattípusok, hanem komplex változók is használhatók a kommunikáció során. Ilyen például egy generikus lista (**`List<string>`**), egy tömb (**`int[]`**), egy saját magad által létrehozott adattípus (**`Employee`**) vagy szinte bármi más. Lényeg, hogy az átküldeni kívánt objektumnak sorosíthatónak (serializable) kell lennie, és a legtöbb .NET objektum, adattípus ilyen.

Itt ennek a technológiának csak a felszínét karcolgattuk. A leírtak már elegendőek egy egyszerű kliens-szerver kommunikáció létrehozásához. Ha szeretnél ennél jobban is elmélyedni a webszolgáltatások használatában, akkor gazdag Windows Communication Foundation irodalom áll a rendelkezésedre; jó kezdőpont lehet az MSDN alábbi oldala: http://msdn.microsoft.com/en-us/netframework/first-steps-with-wcf.aspx.

Bevezetés a Live SDK használatába

A Live SDK

A *Live* a Microsoft számos online szolgáltatásának gyűjtőneve. Ide tartozik a Hotmail platform (amely e-mail fiókot, naptárat, kontaktnyilvántartást stb. biztosít), a Live ID nevű azonosítási szolgáltatás (akinek van Live ID-je, az ezt az egyetlen fiókot tudja használni az összes Microsoft-szolgáltatás és számos külső szolgáltatás eléréséhez), a Live Messenger azonnali üzenetküldő szoftver, a Live Essentials nevű alkalmazásgyűjtemény és még számos egyéb komponens.

A *Live SDK* arra szolgál, hogy a Live szolgáltatásokat saját fejlesztésű alkalmazásokban is fel lehessen használni. Live SDK készült Windows Phone-hoz, „hagyományos" Windows alkalmazásokhoz, Windows 8 stílusú alkalmazásokhoz, de még iOS és Android eszközökhöz is. Segítségével (és természetesen a felhasználó beleegyezésével) hozzáférhetünk a felhasználó nevéhez és Live-ba feltöltött profilképéhez, beleírhatunk a naptárába, használhatjuk a Live Messenger infrastruktúráját chateléshez és így tovább.

Ebben a szakaszban a Live SDK használatának alapjait ismerheted meg. Egy mintaalkalmazást készítünk egy bortúrákat szervező képzeletbeli cég nevében. Az alkalmazás segítségével Live ID-s bejelentkezés után túrákra lehet majd feliratkozni. A túra kiválasztása után bejegyezzük annak időpontját a felhasználó naptárába. Ez a mintaalkalmazás megmutatja a Live SDK óriási előnyét: mivel a naptárbejegyzést a felhasználó „igazi" naptárába helyezzük el (nem pedig valamilyen alkalmazáson belüli, elkülönített adatbázisba), a bejegyzés szinkronizálódik majd a telefonjára, emlékeztetőt lát róla stb., vagyis alkalmazásunk túllép a saját keretein, és közvetlenül be tud épülni a felhasználó által napi szinten használt szolgáltatásokba.

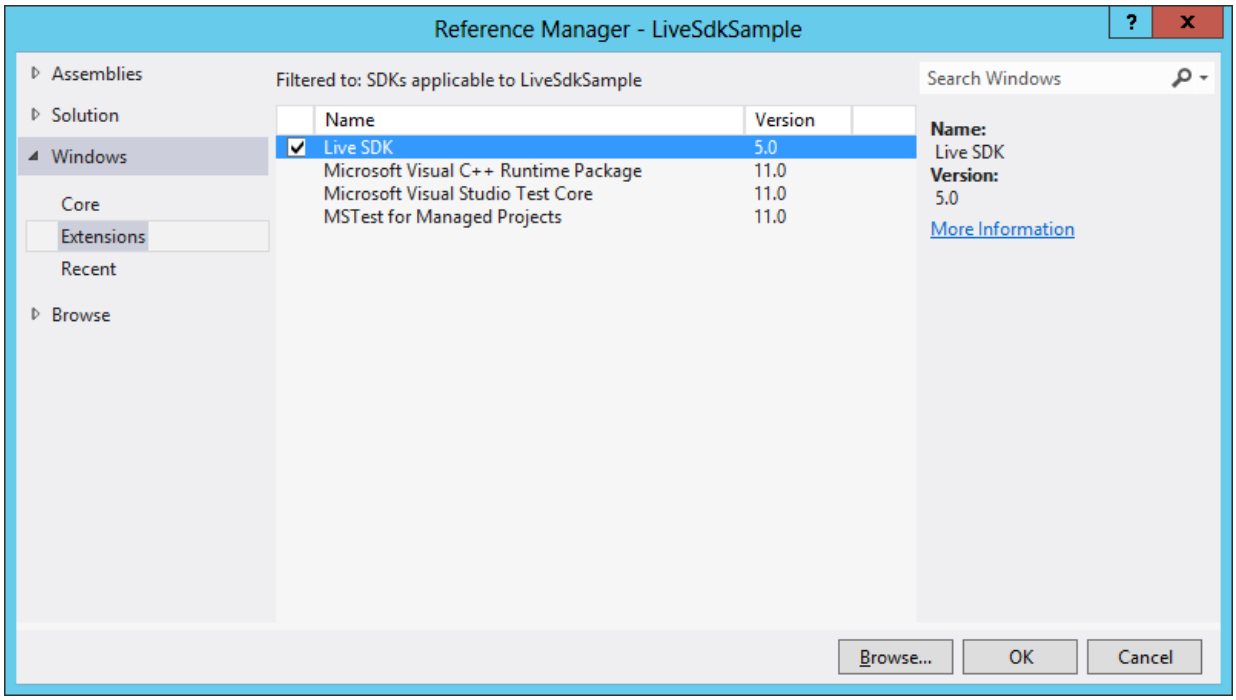
A Live SDK használata egy mintaalkalmazásban

A Live SDK nincs beleintegrálva a Visual Studióba, ezért külön le kell tölteni. Látogass el a http://msdn.microsoft.com/en-us/live/ff621310 linkre és töltsd le a Windows-hoz készült Live SDK verziót!

***Gyakorlat**: Live SDK-t alkalmazó Windows 8 program készítése*

A mintapélda elkészítéséhez kövesd az alábbi lépéseket:

1. Nyisd meg a Visual Studio 2012-t, és hozz létre egy új Blank App típusú Windows 8 stílusú alkalmazást **`LiveSdkSample`** néven!
2. Vegyél fel egy referenciát a Live SDK-ra! Ehhez kattints jobb gombbal a Solution Explorer ablak **`References`** mappájára, majd kattints az Add Reference menüpontra, és a megjelenő párbeszédpanel bal oldali fájában válaszd ki a Windows kategória Extensions alkategóriáját (lásd 11-7 ábra), itt találod a Live SDK-t! Jelöld ezt be, majd az OK gombbal zárd be a párbeszédpanelt!



11-7 ábra: Referencia felvétele a Live SDK-ra

A Live SDK használatához be kell regisztrálni az alkalmazást a Microsoftnál. Ez nagyon egyszerű folyamat, és arra szolgál, hogy az alkalmazásunktól érkező kéréseket azonosítani tudja a Live szolgáltatás.

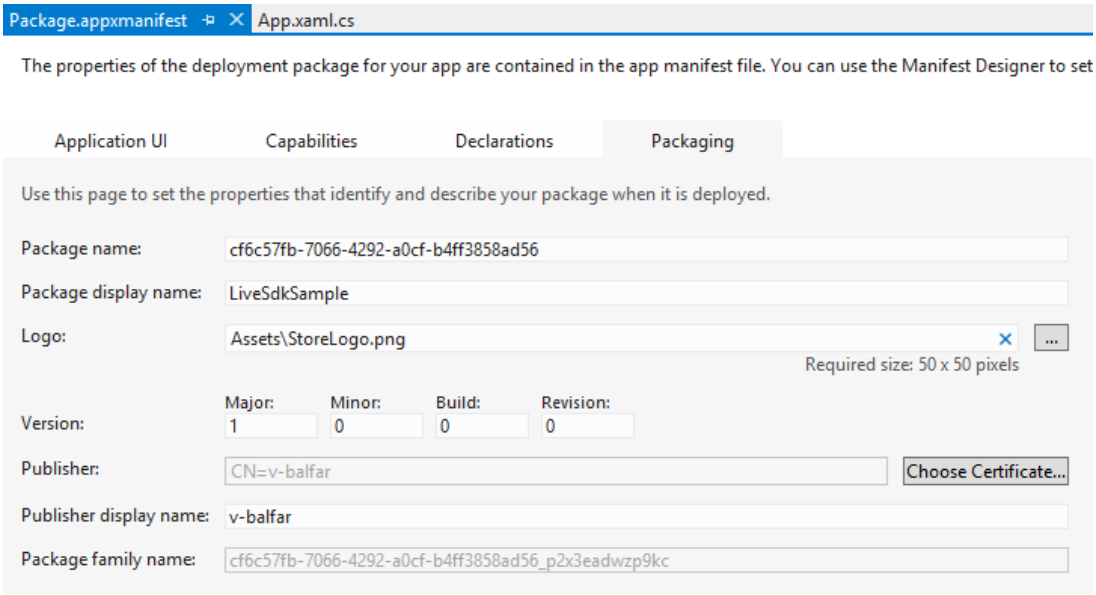
3. A regisztrációhoz először nyisd meg a Solution Explorerben látható **`Package.appxmanifest`** fájlt, és válts ennek a Packaging fülére (lásd 11-8 ábra)!
4. Kattints a https://manage.dev.live.com/build linkre! Az itt megjelenő űrlapra (a 2. lépésbe) másold be a **`Package.appxmanifest`** fájlból a „Package display name" és a „Publisher" értékeket, majd kattints az Accept gombra! A weboldal regisztrálja az alkalmazást, és egy „BUILD.84bba466-3c6b-4151-aaae-6d9d57bcac2f" formátumú nevet generál neki. Ezt a nevet írd vissza a **`Package.appxmanifest`** fájlban lévő régi „Package name" érték helyére, és mentsd el a fájlt!

Ezzel megtörtént az alkalmazás regisztrációja. A következő lépés a felhasználó bejelentkeztetése lesz. Ehhez a Live SDK-ban megtalálható bejelentkeztető gombot célszerű használni.

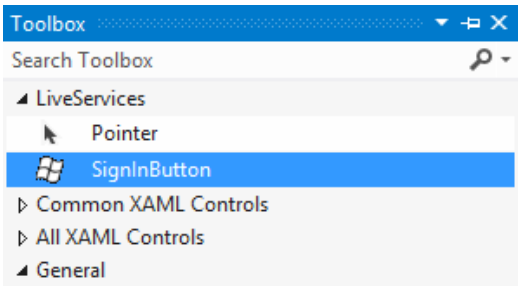
5. Nyisd meg az alkalmazás felületét (**`MainPage.xaml`**), és helyezz el rá egy **`SignInButton`** vezérlőelemet! Ezt a Toolboxon találod meg a LiveServices kategóriában (lásd 11-9 ábra).

6. A Windows 8-ba a hagyományos felhasználónév/jelszó párossal is be lehet jelentkezni, de az operációs rendszer már arra is lehetőséget nyújt, hogy a felhasználó a Live ID-ját használja azonosításra. Az imént elhelyezett gomb mindkét esetet képes kezelni: ha a felhasználó hagyományos felhasználónévvel jelentkezett be a Windowsba, akkor a gombra kattintva megjelenik majd számára a Live ID bejelentkeztető dialógusa. Ha viszont már eleve Live ID-vel azonosította magát a felhasználó az operációs rendszer felé, akkor a gombra kattintva nem kell még egyszer beírnia az adatait! Sőt, ha egyszer már bejelentkezett így az alkalmazásba, és

elfogadta a vonatkozó biztonsági felszólítást, akkor a Windows ezt megjegyzi, és a bejelentkeztetés a jövőben automatikusan megtörténik. Ez igen barátságos és kényelmes megoldás, hiszen a felhasználónak nem kell lépten-nyomon beírnia nevét és jelszavát – elég ezt egyszer megtennie a Windows-ba való bejelentkezéskor.



11-8 ábra: A **`Package.appxmanifest`** fájl



11-9: **`SignInButton`** az Eszköztáron

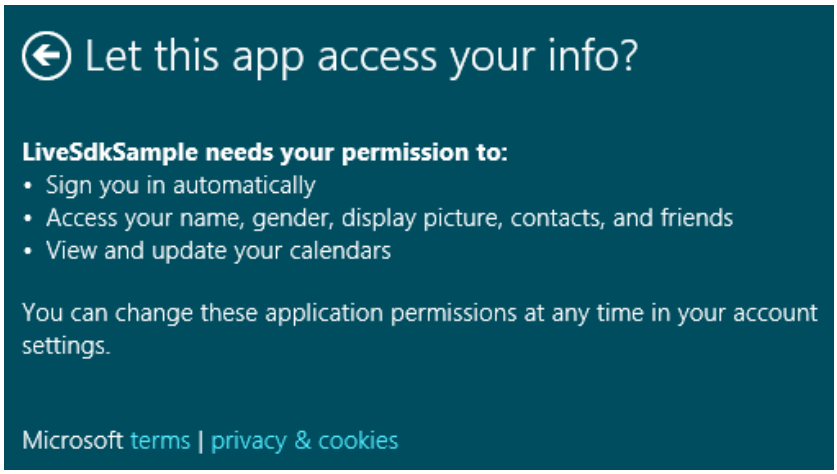
A fejezetrész elején írtaknak megfelelően azonban alkalmazásunk nem egyszerűen csak bejelentkeztetni szeretné a felhasználót, hanem szeretné lekérni annak profiladatait (például név), illetve a naptárjába is el akar helyezni egy bejegyzést. Ezek természetesen nem történhetnek meg a felhasználó beleegyezése nélkül. Alkalmazásunknak erre külön engedélyt kell kérnie, ami az úgynevezett Scope-ok használatával történik: meg kell adnunk, hogy milyen Live ID szolgáltatásokra van szükségünk, és a Windows megkérdezi a felhasználót, hogy engedélyezi-e az alkalmazás számára ezeknek a szolgáltatásoknak a használatát.

A Scope-ok (azaz használni kívánt szolgáltatások) megadásához jelöld ki a **`SignInButton`**-t, és Scopes tulajdonságába írd a következőket:

```
wl.signin wl.basic wl.calendars_update
```

Ez három különböző scope, mellyel a felhasználó automatikus bejelentkeztetésére, alapvető profiladatainak elérésére és naptárának írására kértünk jogot. Számos egyéb Scope létezik még, ezek teljes listája megtalálható a szakasz végén hivatkozott Live SDK fejlesztői oldalon.

7. Az F5 gombbal futtasd az alkalmazást, majd kattints a Sign In gombra! Az általad használt bejelentkeztetési módtól függően a Windows vagy bekéri a Live ID adataidat, vagy ezeket már ismeri. Továbbá rákérdez, hogy adsz-e az alkalmazásnak engedélyt a profiladataid elérésére és a naptárad írására-olvasására (lásd 11-10 ábra).



11-10 ábra: Biztonsági kérdés a Live-tól

8. Add meg az engedélyt, majd lépj ki az alkalmazásból, és térj vissza a Visual Studióba! A sikeres bejelentkeztetés után lássuk, hogy hogyan lehet lekérni a felhasználó profiladatait!
9. Helyezz el egy **`TextBlock`**-ot az alkalmazás felületén, neve legyen **`WelcomeTextBlock`**! Ebbe írod majd be az üdvözlő üzenetet.
10. Kell egy esemény, amivel érzékelhetjük, hogy bejelentkezett a felhasználó. Jelöld ki a **`SignInButton`**-t, majd a Properties ablakban kattints a „villám” ikonra a gomb lehetséges eseményeinek megtekintéséhez, végül kattints duplán a **`SessionChanged`** eseményre! A Visual Studio átvált kódnézetbe. A **`SessionChanged`** esemény akkor következik be, amikor a felhasználó bejelentkezett vagy kijelentkezett a Live ID szolgáltatásból. Helyezd el a kódfájl legtetejére a következő sort:

```
using Microsoft.Live;
```

11. Írd a **`public MainPage()`** sor fölé a következő változódeklarációt:

```
LiveConnectSession session = null;
```

Ebbe a változóba kerül a Live szolgáltatáshoz való kapcsolódásért felelős objektum, ami bejelentkezéskor jön létre, és több helyről is használjuk majd.

12. A gomb eseménykezelője pedig legyen a következő:

```
private async void SignInButton_SessionChanged_1(object sender,
Microsoft.Live.Controls.LiveConnectSessionChangedEventArgs e)
{
    if (e.Session != null && e.Status == LiveConnectSessionStatus.Connected)
    {
        session = e.Session;
        LiveConnectClient liveClient = new LiveConnectClient(session);
        var result = await liveClient.GetAsync("me");
        WelcomeTextBlock.Text = "Hello " + result.Result["name"].ToString();
    }
}
```

Az **`async`** és **`await`** kulcsszavak itt is hasznosnak bizonyulnak.

Ez a kód megnézi, hogy a felhasználó sikeresen bejelentkezett-e; ha igen, akkor elmenti a **`Session`** objektumot, majd lekéri a felhasználó adatait és kitölti az üdvözlő szöveget. A Live SDK-n keresztül Get és Post parancsok segítségével lehet adatokat lekérni és feltölteni, minden parancshoz megadva egy „elérési utat”; a „me” elérési út a felhasználó profiladatait kéri le. Scope-hoz hasonlóan ilyen elérési útból is igen sok áll a rendelkezésünkre, teljes listájuk a szakasz végén belinkelt honlapon található meg.

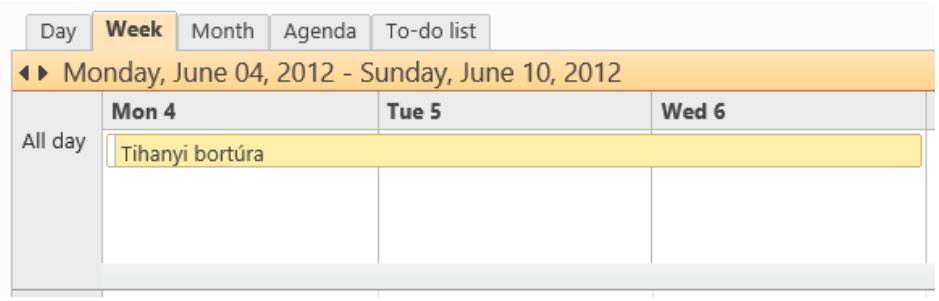
13. Teszteld az alkalmazást! Miután elindult, a **`SignInButton`** kisvártatva átvált „bejelentkezett” állapotba (azaz a „Sign Out” szöveg jelenik meg rajta), ezután rövid idővel pedig a **`TextBlock`** tartalma is megváltozik, és név szerint üdvözöl.

Az alkalmazás így már bejelentkezteti a felhasználót és beolvassa profiladatait. Valósítsuk meg az utolsó kívánalmat is – írjunk a felhasználó naptárába!

14. Hozz létre egy újabb gombot a felhasználói felületen! Nevezd el **`SubscribeButton`**nak, címkéje pedig legyen „Feliratkozás a következő eseményre”! Kattints rá duplán az eseménykezelő megjelenítéséhez! Az eseménykezelő kód létrehoz egy **`Event`** objektumot (mely a Live SDK által ismert sokféle objektum egyike), és ezt elküldi a felhasználó naptárába, azaz a „me/events” elérési útra. (A dátumokat egy meglehetősen speciális formátumban várja a Live, ezért megfelelően formázva kell őket elküldeni.)

```
private async void SubscribeButton_Click(object sender, RoutedEventArgs e)
{
    if (session != null)
    {
        LiveConnectClient liveClient = new LiveConnectClient(session);

        var evt = new Dictionary<string, object>();
        evt.Add("name", "Tihanyi bortúra");
        evt.Add("location", "Tihany");
        evt.Add("description", "A Példa Bortúra Kft. következő rendezvényét " +
            "Tihanyban tartjuk. Indulás a reggel 9-es komppal.");
        evt.Add("start_time", DateTime.Now.AddDays(3)
            .ToString("yyyy'-'MM'-'dd'T'HH':'mm':'ssK"));
        evt.Add("end_time", DateTime.Now.AddDays(6)
            .ToString("yyyy'-'MM'-'dd'T'HH':'mm':'ssK"));

        await liveClient.PostAsync("me/events", evt);
        var dialog = new Windows.UI.Popups.MessageDialog("Esemény sikeresen létrehozva");
        dialog.ShowAsync();
    }
}
```

15. Futtasd az alkalmazást! A Windows ismét bejelentkeztet, majd megjeleníti az üdvözlő szöveget. Kattints rá az újonnan létrehozott gombra; kisvártatva megjelenik a sikerről tájékoztató üzenetablak! Látogass el Hotmail naptáradba, és látni fogod az újonnan létrejött eseményt! (11-11 ábra)



11-11 ábra: A naptárban létrejött esemény

Hogyan működik?

Az alkalmazás a Live SDK segítségével fejlesztőbarát módon hozzáférést biztosít a felhasználó különféle adataihoz. A Live SDK felel azért, hogy a fejlesztő által kiadott egyszerű C# utasításokat ténylegesen és a megfelelő formátumban eljuttassa a Microsoft Live szervereinek, majd a választ visszafordítsa C# kódból értelmezhető formára. Természetesen a Live SDK ehhez először a felhasználó beleegyezését kéri.

Az itt látott profil- és naptáradatokon kívül (megfelelő engedély birtokában) számos egyéb Live szolgáltatáshoz is hozzáférhetsz felhasználód nevében:

- SkyDrive (fényképek, dokumentumok)
- Hotmail (naptárak, kontaktok, kontaktok naptárjai stb.)
- Messenger (azonnali üzenetküldés)

Ezen funkciók használatával egy sor érdekességet meg tudsz valósítani alkalmazásodban. Például:

- Felhasználónk kontaktlistájának ismeretében megkeresheted, hogy mely ismerősei használják még az alkalmazásodat (és remek közösségi szolgáltatásokat ajánlhatsz a figyelmébe),
- alkalmazásod felhasználói között chatszolgáltatást építhetsz a Messenger infrastruktúrát felhasználva,
- felhasználód adatait, alkalmazásodban készített fájljait elmentheted SkyDrive-ba stb.

Az ezekhez szükséges API-k logikájukban hasonlítanak az itt megismertekre, de pontos használatukhoz elengedhetetlen egy teljes körű leírás. A Live SDK dokumentációját a http://msdn.microsoft.com/en-us/live/ linken találhatod, amelyben a fent hivatkozott valamennyi szolgáltatást részletesen ismertetik.

Valós idejű kommunikáció

Az eddig megismert technológiák remekül működnek akkor, amikor alkalmazásunk épp fut (azaz előtérben van). A Windows 8 stílusú alkalmazások életciklus-modellje azonban a rendszer erőforrásainak kímélése érdekében megállítja alkalmazásunkat, amikor háttérbe kerül, így a fenti módszerekkel megírt kódunk ilyenkor nem fut. Számos olyan alkalmazási terület létezik viszont, amikor alkalmazásunknak a háttérből is kommunikálnia kell – képzeljünk el például egy chatklienst vagy akár egy tőzsdei kereskedőrendszert, aminek figyelmeztetnie kell a felhasználót bizonyos események bekövetkeztéről!

Ebben a szakaszban a valós idejű kommunikációval foglalkozunk, vagyis azokkal a fejlesztői eszközökkel ismerkedünk meg, amelyek segítségével egy háttérben lévő Windows 8 stílusú alkalmazás folyamatos kapcsolatot tud fenntartani egy szolgáltatással.

Alapfogalmak

A valós idejű kommunikáció lehetőségeinek megértéséhez fontos, hogy ismerd a háttérben futtatott Windows 8 stílusú alkalmazásokhoz kapcsolódó alapfogalmakat. A lenti fogalmak ismertetése után rátérünk arra, hogyan írhatsz ezek segítségével valós idejű kommunikációt használó alkalmazást.

A kérdéses fogalmak:

- **Nyitóképernyő (Lock Screen) alkalmazások:** A Start képernyőre a felhasználó tetszőleges számú alkalmazást elhelyezhet, melyek lapkájukkal jelennek meg ott. Ehhez hasonlóan a nyitóképernyőre is elhelyezhetők alkalmazások (ez az a képernyő, amely a gép bekapcsolása után, a felhasználó bejelentkezése előtt jelenik meg), ám egyszerre maximum 7 alkalmazás kerülhet fel ide. Az ide kirakott alkalmazások kapnak a kezdőképernyőn egy helyet, ahová értesítéseket (pl. beérkezett e-mailek darabszáma) helyezhetnek el; ezek naprakészen tartásához a Windows más alkalmazások számára nem megengedett lehetőségeket is biztosít számukra.
- **Windows Push Notification Services (WNS), azaz értesítések:** a Microsoft által biztosított szolgáltatás. Windows 8 alkalmazásod beregisztrálhatja magát a *Push Notification* szolgáltatásba (WNS-be), felhőben futó (vagy internetes) alkalmazásod pedig üzenhet a WNS-nek, amit ő továbbít Windows 8 alkalmazásod felé.
  A szolgáltatás lényege, hogy a mechanizmus segítségével „push" üzeneteket küldhetsz, azaz nem az alkalmazásodnak kell kérdésekkel bombáznia a szervert („Történt-e valami?"), hanem a WNS infrastruktúrán keresztül a szerver üzenhet az alkalmazásnak.

A küldhető értesítéseknek több fajtája van; fajtától függően a Windows egy értesítés hatására feldobhat egy üzenetablakot (*toast*ot), frissítheti alkalmazásod egyik lapkáját stb.

- **Background Tasks, azaz háttérben futó feladatok:** Alkalmazásod részeként írhatsz kisebb, különálló kódrészleteket, amelyeket a Windows 8 különféle feltételek teljesülése esetén meghív. Ezek a *Background Task*ok. Futási idejük, az általuk felhasználható erőforrások korlátozottak; időnként végrehajtandó karbantartási, értesítési stb. feladatok ellátására valók.
- **Triggerek:** Események, melyekre alkalmazásod felirathat Background Task-okat. Ha valamelyik trigger bekövetkezik (tüzel), akkor a hozzá regisztrált Background Task lefut. A triggerek néhány fajtája:
  - *System Trigger*: A Windowsban bekövetkező rendszereseményekhez (például a felhasználó bejelentkezéséhez, az internetkapcsolat felépüléséhez vagy elvesztéséhez stb.) kötődő trigger.
  - *Time Trigger*: Általad definiált időközönként eseményt jelző trigger.
  - *Push Notification Trigger*: A WNS-en keresztül érkező egyik típusú értesítés (raw notification) hatására jelez eseményt.
  - *Network Trigger*: Ha a WNS valamiért nem megfelelő céljaidnak, akkor Network Triggerek és Background Taskok alkalmas beállításával fenntartható egy tetszőleges TCP kapcsolat egy saját szerver felé; ha ezen keresztül csomag érkezik, akkor az alkalmazás értesítést kap és Background Taskot futtathat.

Valós idejű kapcsolat fenntartásának lehetőségei

Lássuk, hogy a fentiek segítségével hogyan készíthető háttérből is működő, folyamatos (valós idejű) kapcsolat egy szolgáltatás felé! Erre több módszer is létezik:

1. Alkalmazásunk regisztrálhatja magát a Push Notification (WNS) szolgáltatásba. A Windows az ilyen módon kapott csatornát folyamatosan nyitva tartja, a szervertől érkező üzenetek hatására pedig egy triggert indít, amely meghívhat egy Background Taskot. Ezt a lehetőséget csak nyitóképernyőre (lock screen-re) kikerült alkalmazások használhatják.
2. Alkalmazásunk Network Triggerek fenntartásával felépíthet és fenntarthat egy TCP csatornát egy saját szerver felé (a WNS szolgáltatás itt nem jön a képbe). Amikor ezen a TCP csatornán keresztül csomagok érkeznek, akkor a Windows elindít egy triggert, amely meghívhat egy Background Taskot. Ezt a megoldást is csak nyitóképernyőre kikerült alkalmazások használhatják.
3. Alkalmazásunk Time Triggerek segítségével néhány percenként lefuttat egy kódrészletet, amely információkat kérdez le a szerverről. A fentiekhez hasonlóan ezt is csak nyitóképernyőre kikerült alkalmazások használhatják.

Az első megoldás előnyei:

- A WNS infrastruktúrát a Microsoft biztosítja; saját szervereden nem kell folyamatosan nyitva tartott TCP kapcsolatokkal bajlódni (hiszen a WNS-en keresztül való üzenetküldéshez a szervered a WNS-nek üzen, aztán rögtön be is zárhatja a kapcsolatot).
- A WNS használata egyszerűbb, mert a Windows 8 app és a WNS közti kapcsolatot a Windows tartja ébren, ennek implementációjával nem kell külön foglalkoznod.

A második megoldás előnyei:

- Tetszőleges protokoll használható a szerver és a kliens közötti kapcsolattartásra.
- Mivel az alkalmazás és a szerver közvetlenül kommunikálhatnak, ezért a fejlesztő megoldhatja az üzenetek garantált kézbesítését, ha erre igény van. A WNS használatával ilyen garancia nem érhető el (azaz a WNS üzenetek kézbesítése „best effort” jelleggel történik, a Microsoft mindent megtesz, de nem vállal garanciát).

A harmadik megoldás előnyei:

- Ezt a legegyszerűbb implementálni, mert alkalmazásunknak nem kell semmilyen csatornát nyitnia a szerver felé – egyszerűen néha meghívja azt, és megvárja a választ.

- A megoldás igazából nem valós idejű (hiszen nem a szerver küld értesítést, hanem az alkalmazás kérdezget), de látszólag az lesz (ha a felhasználó elmegy egy órára a gépétől, akkor visszatértekor a nyitóképernyőn relatíve friss adatokat talál majd, amelyek „maguktól" frissültek úgy is, hogy az alkalmazás nem futott).

Mindhárom megoldásra igaz:

- Csak a nyitóképernyőn (lock screen-en) kinnlévő alkalmazások használhatják őket.
- Az általuk meghívott Background Taskoknak C#-ban, VB.NET-ben vagy C++-ban kell készülniük; a JavaScript nem támogatott.

A három megoldás közül az elsővel (Push Notification-ök használata) a könyv 9. fejezete foglalkozik. A második megoldás (Network Trigger, saját TCP csatorna a szerver felé) implementációjának bonyolultsága meghaladja könyvünk kereteit, ezért itt a 3. megoldásra (Time Trigger, időnként lefuttatott kódrészlet) térünk ki.

> *A kimaradó Network Triggerek használatáról a Windows 8 MSDN dokumentációjában található részletes leírás. Ezeket akkor érdemes használni, ha valamilyen okból a Push Notification-öket nem kívánjuk alkalmazni, vagy saját kommunikációs protokollt szeretnénk fejleszteni.*

Valós idejű kommunikáció megvalósítása Time Trigger segítségével

A Time Trigger működésének lényege, hogy alkalmazásunk részeként írunk egy kis kódrészletet (Background Task), melyet a Windows 8 előre definiált időközönként lefuttat. Ebből a kódrészletből pedig tetszőleges kérést (pl. egy webszolgáltatás-hívást, lásd korábban) elküldhetünk a szerver felé, mellyel frissíthetjük alkalmazásunkat.

A Time Trigger használatának előfeltétele, hogy alkalmazásunk kint legyen a felhasználó nyitóképernyőjén (Lock Screen). A Windows 8-ban egyszerre maximum hét alkalmazás lehet a nyitóképernyőn, és a kihelyezés lépését a felhasználónak külön meg kell tennie, ezért távolról sem biztos, hogy alkalmazásunknak futási időben lehetősége lesz Time Triggerek definiálására. Ennek megfelelően úgy kell megírnunk alkalmazásunkat, hogy a Time Triggerek kiegészítő szolgáltatásként jelenjenek meg, ne pedig a működés elengedhetetlen elemeként!

Tartsuk azt is észben, hogy a Time Triggerek segítségével megvalósított viselkedés nem igazi valós idejű kommunikáció lesz, hiszen nem a szerver fog üzenetet küldeni alkalmazásunknak (push modell), hanem alkalmazásunk fog kérdezgetni a szervertől (pull modell)! Ettől függetlenül alkalmazásunk önállóan tud majd frissülni akkor is, amikor a felhasználó nem futtatja. Ennek a modellnek a megvalósítása pedig jóval egyszerűbb, mint a két másik megoldásé (Push Notification és Network Trigger).

***Gyakorlat**: Önmagát háttérből is frissíteni képes alkalmazás készítése Time Triggerrel*

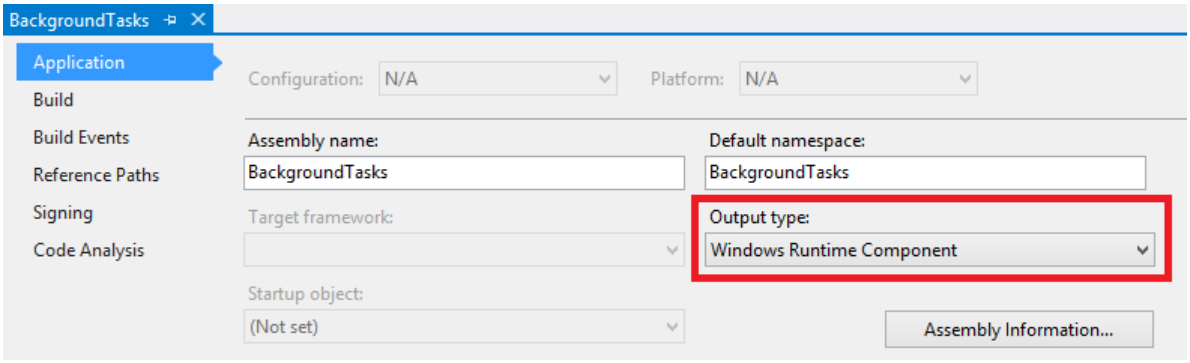
Lássunk is neki! Elkészítendő alkalmazásunk kérni fogja saját kirakását a nyitóképernyőre, majd pedig egy háttérfeladatot futtat, ami 15 percenként frissíti a nyitóképernyőn lévő ikonját. Az alkalmazás elkészítéséhez kövesd az alábbi lépéseket:

1. Készíts egy új, üres (Blank App) Windows 8 Metro style alkalmazást **`BackgroundCommunicationSample`** néven!

Az első feladatod a Background Task megírása lesz. Ezt a háttérfeladatot fogja majd 15 percenként hívogatni a Windows, és ez kommunikál szerverünkkel az alkalmazás állapotának frissítéséhez. Ügyelned kell arra, hogy a gép erőforrásainak kímélése érdekében a háttérfeladatok erőforrás-felhasználására komoly megkötések vannak, ezekről az MSDN oldalain tájékozódhatsz.

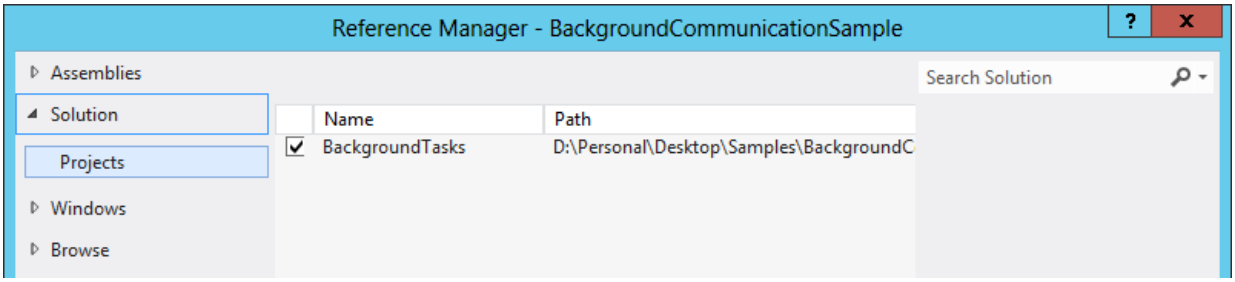
2. A háttérfeladat létrehozásához adj hozzá egy új projektet a Visual Studio megoldáshoz! Ehhez kattints jobb gombbal a megoldás nevére (*Solution 'BackgroundCommunicationSample'* néven a legfelső elem a Solution Explorer ablakban), a helyi menüből válaszd az Add menü New Project lehetőségét, majd adj hozzá egy Class Library (Windows Store apps) projektet **`BackgroundTasks`** néven!

3. Lépj be a projekt tulajdonságlapjára, és az Application fülön a projekt Output type tulajdonságát állítsd át Windows Runtime Component értékre (11-12 ábra)!



11-12 ábra: A `BackgroundTasks` projekt kimeneti típusának beállítása

4. Töröld le a projektből a **`Class1.cs`** fájlt; erre nem lesz szükség.
5. Vegyél fel egy referenciát a **`BackgroundCommunicationSample`** projektből a **`BackgroundTasks`** projektre! Ehhez kattints jobb gombbal a **`BackgroundCommunicationSample`** projekt **`References`** mappájára, válaszd az Add Reference menüpontot, és bal oldalt a Solution fül Projects menüpontja alatt jelöld ki a **`BackgroundTasks`** elemet (11-13 ábra)!



11-13 ábra: Referencia felvétele a `BackgroundTasks` projektre

6. Vegyél fel egy új osztályt a **`BackgroundTasks`** projektbe, annak neve legyen **`BadgeUpdaterBackgroundTask`**! Ebben az osztályban szerepelnie kell egy **`Run`** metódusnak; a Windows 8 ezt hívja majd meg, amikor a háttérfeladat aktivizálódik. A **`Run`** metódus feladata az, hogy frissítse alkalmazásunk nyitóképernyőn lévő ikonját.

Az ikon frissítése némileg csavaros módon történik: a kód egy speciális formátumú XML üzenetet küld a Windows felé. A lenti kód beszerzi ennek az XML üzenetnek a formátumát, beleírja a megfelelő tartalmat, és elküldi az operációs rendszernek, mire az ikon frissülni fog.

7. Helyezd a kódfájl tetejére az alábbi két **`using`**-ot:

```
using Windows.ApplicationModel.Background;
using Windows.UI.Notifications;
```

8. Helyezd el frissen létrehozott osztályodba az alábbi kódot (a **`sealed`** kulcsszó nagyon fontos):

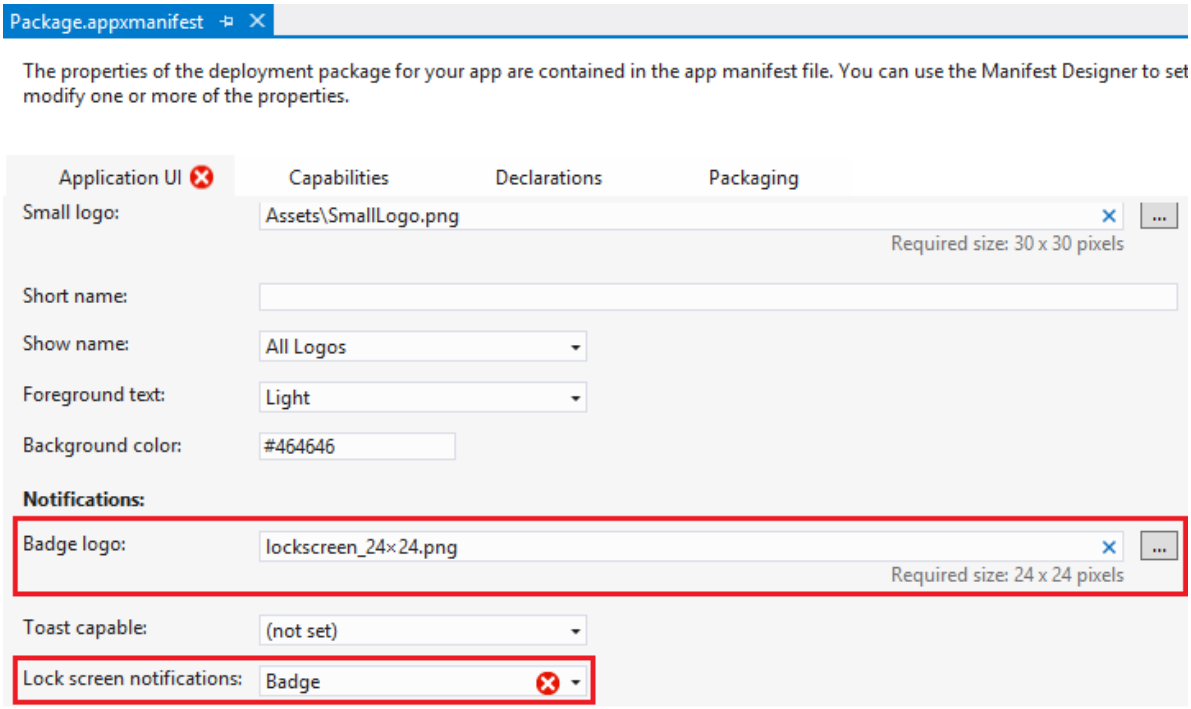
```
public sealed class BadgeUpdaterBackgroundTask : IBackgroundTask
{
    void IBackgroundTask.Run(IBackgroundTaskInstance taskInstance)
    {
        SendUpdate();
    }

    public void SendUpdate()
    {
        var xml = BadgeUpdateManager.GetTemplateContent(BadgeTemplateType.BadgeNumber);
```

```
            xml.GetElementsByTagName("badge").First().Attributes[0].InnerText =
                (new Random((int)DateTime.Now.Ticks)).Next(0, 100).ToString();

            BadgeUpdateManager.CreateBadgeUpdaterForApplication()
                .Update(new BadgeNotification(xml));
        }
    }
```

Ezzel a Background Task kódja elkészült, de persze még nincs semmi, ami meghívja azt.

A Windows 8 kötelezően előírja, hogy amennyiben egy program Background Task-okat kíván használni, vagy ki akar kerülni a nyitóképernyőre, ezt külön deklarálnia kell egy leírófájlban. Következő feladatunk ennek a deklarációnak az elvégzése.

9. A **`BackgroundCommunicationSample`** projektben kattints duplán a **`Package.appxmanifest`** fájlra, majd válaszd ki az Application UI fület! A Lock screen notifications menüpontot állítsd Badge-re! Megjelenik a Badge logo mező mellett egy piros ikon; itt azt kell beállítani, hogy mi legyen az alkalmazás ikonja a nyitóképernyőn. Ide egy 24×24 pixeles képet kell megadni (ld. 11-14 ábra). Magad is gyárthatsz egy ilyet, illetve a fejezethez tartozó példakódok között is találsz megfelelő képet.

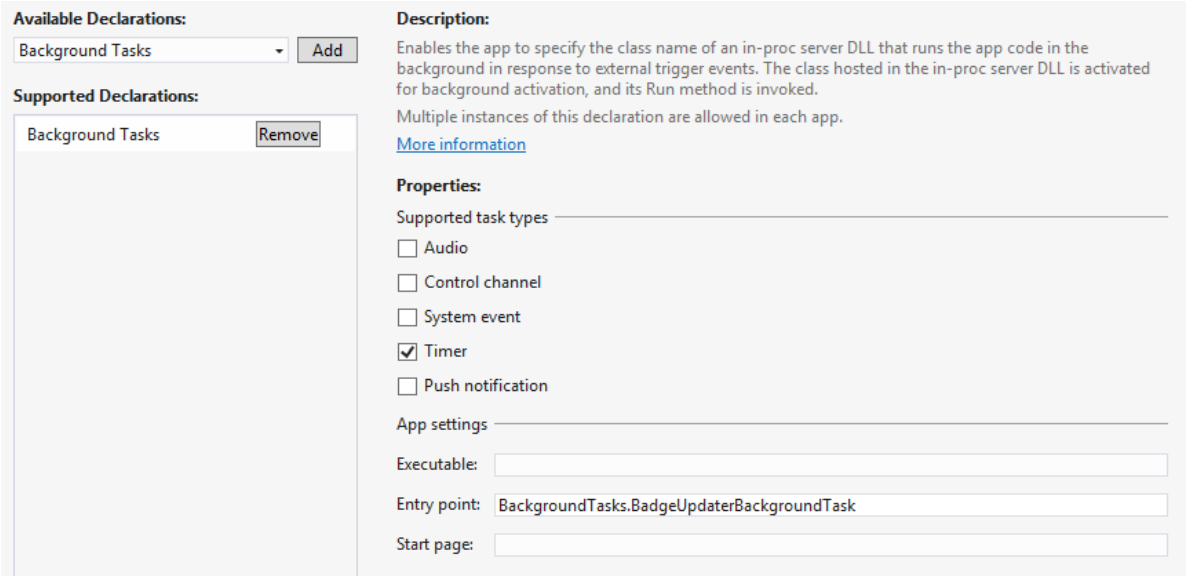


11-14 ábra: A Badge beállításai

Egy alkalmazás kizárólag akkor tud a háttérből kommunikálni (akár Push Notification-ök, akár Network Triggerek, akár Time Triggerek segítségével), ha a felhasználó felvette azt a nyitóképernyőre (Lock Screen)! Fontos tudni, hogy ez visszafelé is igaz: egy alkalmazás csak akkor kerülhet ki a nyitóképernyőre, ha utána használni fog ilyen jellegű szolgáltatásokat.

Mivel az előbb megadtad, hogy az alkalmazás kikerülhet a Lock Screen-re, ezért most azt is meg kell adnod, hogy milyen Background Task fogja ezt frissíteni. (Ezt jelzi a Lock screen notifications mellett egy vörös ikon is.)

10. Válts át a Declarations fülre, és adj hozzá egy Background Tasks deklarációt! Ezt konfiguráld is be (ld. 11-15 ábra): pipáld be, hogy Timer típusú lesz a háttérfeladat, illetve az Entry point mezőben add meg a háttérfeladat osztályának teljes nevét (esetünkben **`BackgroundTasks.BadgeUpdaterBackgroundTask`**)!



11-15 ábra: A Declarations fül konfigurációja

A deklaráció ezzel elkészült: a Windows 8 immár tisztában van azzal, hogy alkalmazásunk ki akar majd kerülni a nyitóképernyőre, és ismeri az ehhez használandó háttérfeladat paramétereit is. Mivel a háttérfeladat kódját is megírtuk, ezért már csak egy feladatunk van: ténylegesen kérnünk kell, hogy az alkalmazás kikerüljön a nyitóképernyőre, valamint be kell állítanunk a háttérfeladat 15 percenkénti futtatását.

11. Nyisd meg az alkalmazás **`MainPage.xaml`** fájlját, és vegyél fel rá egy új gombot! A gomb neve legyen **`RegisterButton`**, felirata pedig Regisztráció! Miután megvan, kattints rá duplán, hogy belépj a kódszerkesztőbe!

Alkalmazásunknak először is kérnie kell, hogy kikerülhessen a nyitóképernyőre. A Windows 8 erről természetesen megkérdezi a felhasználót. Ha a felhasználó nemet mond, akkor hiába fut le a kódunk többször is, a Windows 8 nem fog ismét rákérdezni, ezért ebben az esetben emlékeztetjük a felhasználót, hogy kézzel tudja elhelyezni alkalmazásunkat a nyitóképernyőjén. Ezután felvesszük a 15 percenkénti frissítést végző Time Triggert.

12. Helyezd a fenti két sort a kódfájl tetejére:

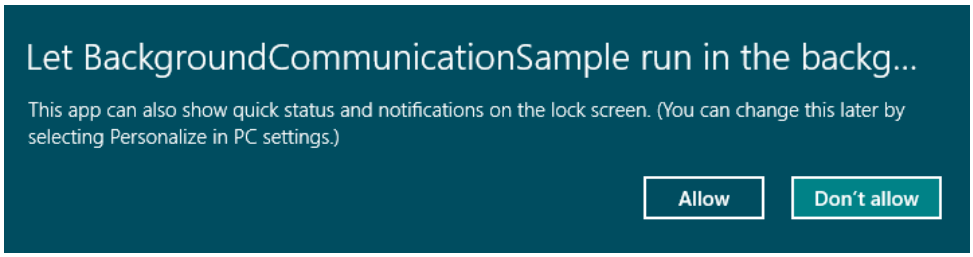
```
using Windows.ApplicationModel.Background;
using Windows.UI.Popups;
```

13. Írd a következő kódot a kattintás eseménykezelőjébe (ne hagyd figyelmen kívül a metódus első sorában lévő **`async`** kulcsszót sem):

```
private async void RegisterButton_Click(object sender, RoutedEventArgs e)
{
    BackgroundAccessStatus status = await BackgroundExecutionManager.RequestAccessAsync();
    if (status == BackgroundAccessStatus.Denied || status == BackgroundAccessStatus.Unspecified)
    {
        new MessageDialog("Kérlek add hozzá az appot kézzel a nyitóképernyőhöz!").ShowAsync();
    }

    BackgroundTaskBuilder builder = new BackgroundTaskBuilder();
    builder.Name = "Véletlenszerű frissítő";
    builder.TaskEntryPoint = "BackgroundTasks.BadgeUpdaterBackgroundTask";
    builder.SetTrigger(new TimeTrigger(15, false));
    builder.Register();
}
```

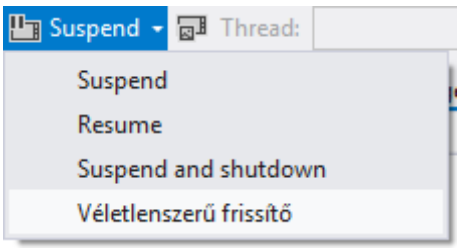
Miután ezzel elkészültél, itt az idő az alkalmazás tesztelésére!

14. Indítsd el az alkalmazást, és kattints a Regisztráció gombra! A Windows 8 megkérdezi (11-16 ábra), hogy az alkalmazás futhat-e a háttérben, és kikerülhet-e a nyitóképernyőre. Nyomd meg az Allow gombot!



11-16 ábra: A háttérben futtatást engedélyező ablak

Ezzel az alkalmazás kikerül a nyitóképernyőre, és lefut a háttérfeladat időzítését végző kód. Ha nem akarsz várni 15 percet, hogy lefusson az első frissítés (és ezzel megjelenjen az alkalmazás a nyitóképernyőn), akkor gördítsd le a Visual Studio debuggolási eszköztárán a „Suspend" gombot, válaszd ki rajta a Background Task-ot (Véletlenszerű frissítő), és nyomd meg a gombot (11-17 ábra) – ezzel kézzel is le bírod futtatni a Background Task-ot. Ha ezután zárolod a Windowsodat (Win+L), akkor rögtön látni is fogod az alkalmazás által elhelyezett ikont, és ha néhány percet vársz, akkor az ikon frissülni is fog.



11-17 ábra: A Background Task kézi lefuttatása

Innentől kezdve a beállított háttérfolyamat 15 percenként lefut majd, akkor is, ha az alkalmazás maga nincs megnyitva, és akkor is, ha közben a gép újraindul.

Hogyan működik?

A Windows 8 azzal (is) takarékoskodik a számítógép erőforrásaival, hogy a nem előtérben lévő alkalmazásokat felfüggeszti. Indokolt esetben, a felhasználó beleegyezésével azonban ez a korlátozás megkerülhető; ebben a gyakorlatban az alkalmazás kikerül a Lock Screen-re, majd egy Time Trigger segítségével rendszeresen elindít egy rövid idő alatt lefutó kódrészletet, amivel frissíti a Lock Screen-en lévő ikonját és egyéb belső adatait.

Ezzel megismerted a háttérfolyamatok, a Lock Screen alkalmazások és a Time Trigger használatát. Ennek segítségével elkészítheted saját, háttérből való kommunikációra képes alkalmazásodat, és láttad a valós idejű kommunikációra szolgáló két további módszert is.

A felhő és a Windows 8 alkalmazások

Számítási felhő alatt óriási, központilag menedzselt adatközpontokat értünk, amelyek „kívülről" gyakorlatilag végtelenül nagynak tűnnek. Ezekből informatikai kapacitást bérelhetünk, például szervergépeket, tárhelyet, adatbázisokat és más hasonló erőforrásokat. A felhőben mindig csak az aktuális felhasználás alapján kell fizetni, és az ár jellemzően kedvezőbb, mint ha mi magunk építenénk ki az adott kapacitást. Ezenkívül számos helyben jelentkező problémaforrást a felhőben megoldanak helyettünk – nincsenek például hardveres (mi van, ha elromlik a vinyó?) és skálázási (mi van, ha holnap hirtelen még ötször ennyi szerverre van szükségem) aggodalmaink.

Ennek megfelelően, ha alkalmazásunknak szüksége van egy szerverre, tárolnia kell valahol egy high-score listát, esetleg fel kell valahová töltenie a felhasználók profilképeit, akkor érdemes megfontolni a felhő bevetését.

A Microsoft felhőplatformját Windows Azure-nak hívják. A Windows 8 alkalmazások közvetlenül képesek igénybe venni az Azure számos szolgáltatását. Ehhez készült az alkalmazásokba beépíthető eszközkészlet Windows Azure Toolkit for Windows 8 néven. Mivel jelen sorok írásakor (a Windows 8 kiadás előtti fázisában) az eszközkészlet még nem támogatja az aktuális Windows 8 és Visual Studio verziót, ezért ennek használatát könyvünk egy későbbi frissítésében mutatjuk be.

A Windows Azure Toolkit honlapja http://watwindows8.codeplex.com/, az Azure platformról (és annak Windows 8 támogatásáról) pedig a http://www.azure.com címen olvashatsz bővebben. Könyvünk frissítéseit a http://devportal.hu portálon találod majd!

Adatelérés az OData protokollon keresztül

Az interneten és a vállalatoknál számos olyan szoftver található, amelyektől 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 stb. tölthetjük le.

Legyen szó bármilyen adatforrásról, az adatokon alapvetően 4 műveletet tudunk végrehajtani: létrehozás (Create), olvasás (Read), frissítés (Update), törlés (Delete). Ezeket hívják CRUD műveleteknek.

Hiába rögzített 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, az elvégezhető műveletek nem változnak. 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 egy teljesen általános fogalom; a **`Categories`** gyűjtemény fizikailag *bármi* lehet. Állhat mögötte egy SQL tábla, egy XML fájl, vagy akár egy szöveges fájl. Ez nekünk fogyasztóknak teljesen lényegtelen is, 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>
```

A fentihez hasonló, az URL részeként kiadható parancsok minden CRUD művelethez definiáltak. Az OData szolgáltatóknak 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 a **`HttpWebRequest`** osztállyal kiadhatunk nyers HTTP kéréseket, melyekre az OData adatforrás válaszolni fog.

> *Ez adja egyben az OData platformfüggetlenségét – HTTP kéréseket gyakorlatilag bármilyen platformról indíthatunk, legyen az Windows, Linux, Mac vagy akármilyen egyéb eszköz.*

Ez azonban így kényelmetlen. Így számos platformra, többek között Windows 8-hoz is készült OData kliensoldali osztálykönyvtár. Az osztálykönyvtár segítségével nem kell a nyers HTTP kérésekkel bajlódnunk; 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 igénybevételével beolvassa az eBay.com aktuális akcióit.

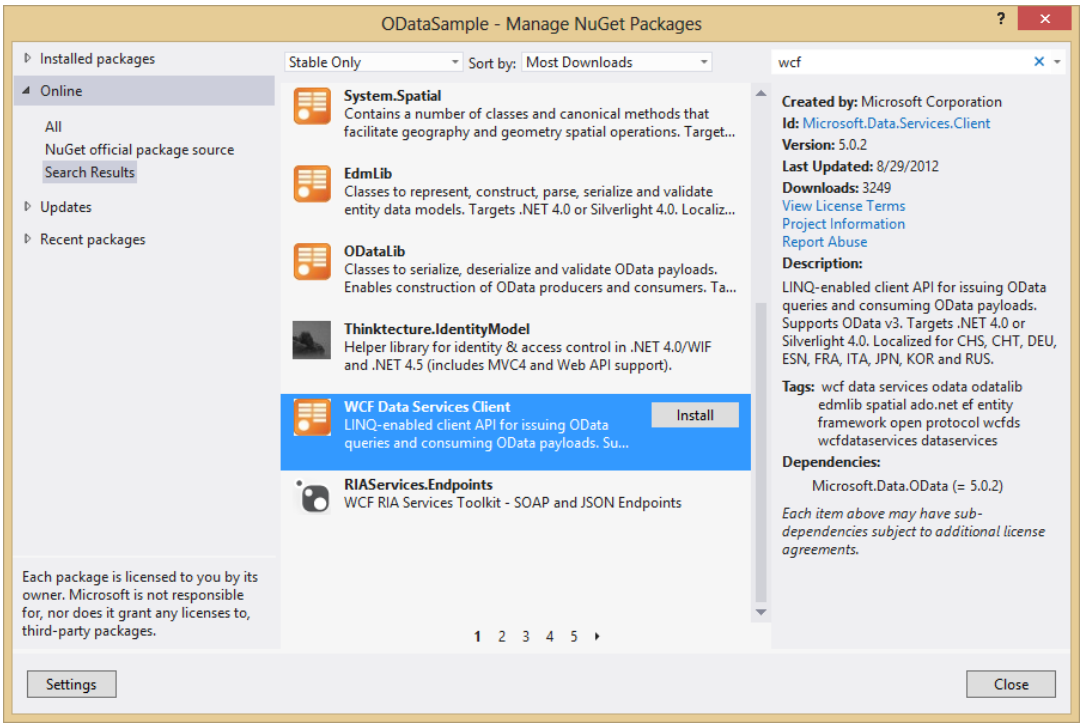
***Gyakorlat**: OData végpont felhasználására képes alkalmazás fejlesztése*

Az alkalmazás elkészítéséhez kövesd az alábbi lépéseket:

1. Hozz létre egy új üres (Blank Application típusú) projektet, nevezd el azt **`ODataSample`**-nek!

Az OData használatához a WCF Data Services nevű .NET Framework-komponensre van szükségünk. A .NET Framework 4.5 végleges verziója óta ez nem része az alap .NET Frameworknek, ezért külön le kell töltenünk. Erre célszerű a NuGet nevű csomagkezelő rendszert használnunk, amely bele van építve a Visual Studióba.

2. Kattints jobb gombbal az **`ODataSample`** projektre, és válaszd a Manage NuGet Packages menüpontot. A megjelenő felületen a bal oldalon válaszd ki az Online kategóriát, majd a jobb felső sarokban lévő keresőmező segítségével keress rá a „wcf" kifejezésre! A találatok közül keresd ki és telepítsd fel a WCF Data Services Client elemet (11-18 ábra)!

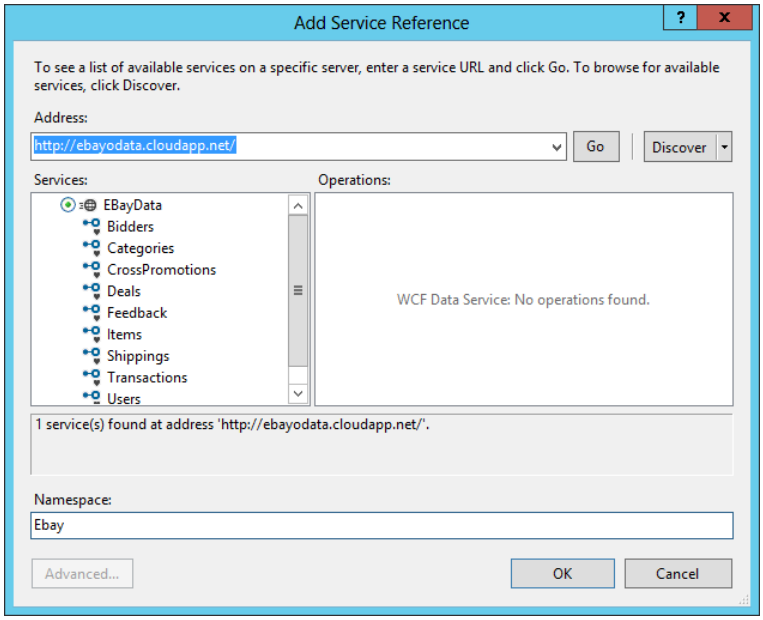


11-18 ábra: a WCF Data Services telepítése NuGet-tel

Jelen sorok írásakor a legújabb WCF Data Services Client verzió még nem támogatta a .NET Framework 4.5-öt. A lenti leírás ezért a .NET Framework 4.5 kiadás előtti verziójával készült, így a végleges állapothoz képest apróbb eltérések előfordulhatnak.

3. Vegyél fel egy referenciát az OData végpontra! Ehhez (a korábban már látott webszolgáltatásokhoz hasonlóan) kattints a projekt nevére a Visual Studióban, és válaszd az Add Service Reference lehetőséget! A cím mezőbe írd be az eBay OData végpontjának címét: **`http://ebayodata.cloudapp.net/`**, a Namespace mezőben pedig nevezd el a referenciát **`Ebay`**-nek!

Amint a 11-19 ábrán is látható, a Visual Studio felismeri, hogy ez nem hagyományos webszolgáltatás, hanem adatszolgáltatás (lásd a panel jobb oldala), hiszen nem tartalmaz műveleteket.



11-19 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 metódusokkal, amikkel kódunkból megszólíthatjuk az OData szolgáltatást. (Az ismétlés 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.)

4. Készítsd el a szolgáltatás megszólításához szükséges kódot! Ehhez két névtér kell nekünk, ezeket helyezd el a **MainPage.xaml.cs** kódfájl legtetején!

```
using System.Data.Services.Client;
using Windows.UI.Popups;
```

A feladat elvégzéséhez két fontos változót kell deklarálnunk. Az **EBayData** típusú változó biztosítja majd 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 szerveroldalról lekérdezett rekordokkal.

5. Helyezd a két változó deklarációját közvetlenül a **public sealed partial class MainPage : Page** sor (és az azt követő nyitó kapcsos zárójel) után, hogy azt több metódusból is használhasd!

```
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 az azt végző kódnak közvetlenül a **public MainPage()** konstruktorba (a már ott lévő sor alá) kell kerülnie.

6. Írd be a kijelölt kódot a MainPage osztály konstruktorába:

```
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 végpont megszólításához elsőként létrehozunk egy példányt az **EBayData** nevű osztályból. Ennek a példánynak létrejöttekor átadjuk a megcímzendő szolgáltatás URL-jét. Majd kreálunk 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 adatokat vennie.

Ezután létrehozunk egy LINQ lekérdezést, amellyel megadjuk, hogy a **deals** gyűjteménybe mit szeretnénk betölteni a szerveroldalról. 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 feliratkozunk a lekérdezés befejeztét jelző eseményre, majd kiadjuk az utasítást a **deals** gyűjtemény feltöltésére az általunk megadott LINQ lekérdezés szerint. (Jelen sorok írásakor az OData könyvtár még nem támogatja az **async** kulcsszót, így a „hagyományos", eseményre feliratkozó aszinkron kódolási mintát követjük.)

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 tovább is.

A **deals_LoadCompleted** eseménykezelő meghívásakor lekérdezésünk lefutott, és a visszaadott objektumok betöltődtek a **deals** gyűjteménybe. A fent említett mechanizmus miatt azonban előfordulhat, hogy még vannak betöltetlen rekordok az adatforrásban. Mi valamennyi rekordot szeretnénk lekérdezni, ezért ha kódunk azt érzékeli, hogy a lekérdezés még folytatható, akkor a **LoadNextPartialSetAsync()** hívással a következő adag objektumot is letölti. Ehhez nem kell új eseménykezelőt felvennünk, mert ez a metódus is a **deals_LoadCompleted** eseménykezelőt hívja meg befejeztekor, így lekérdezésünk addig fut, amíg minden objektum meg nem érkezik a szerverről.

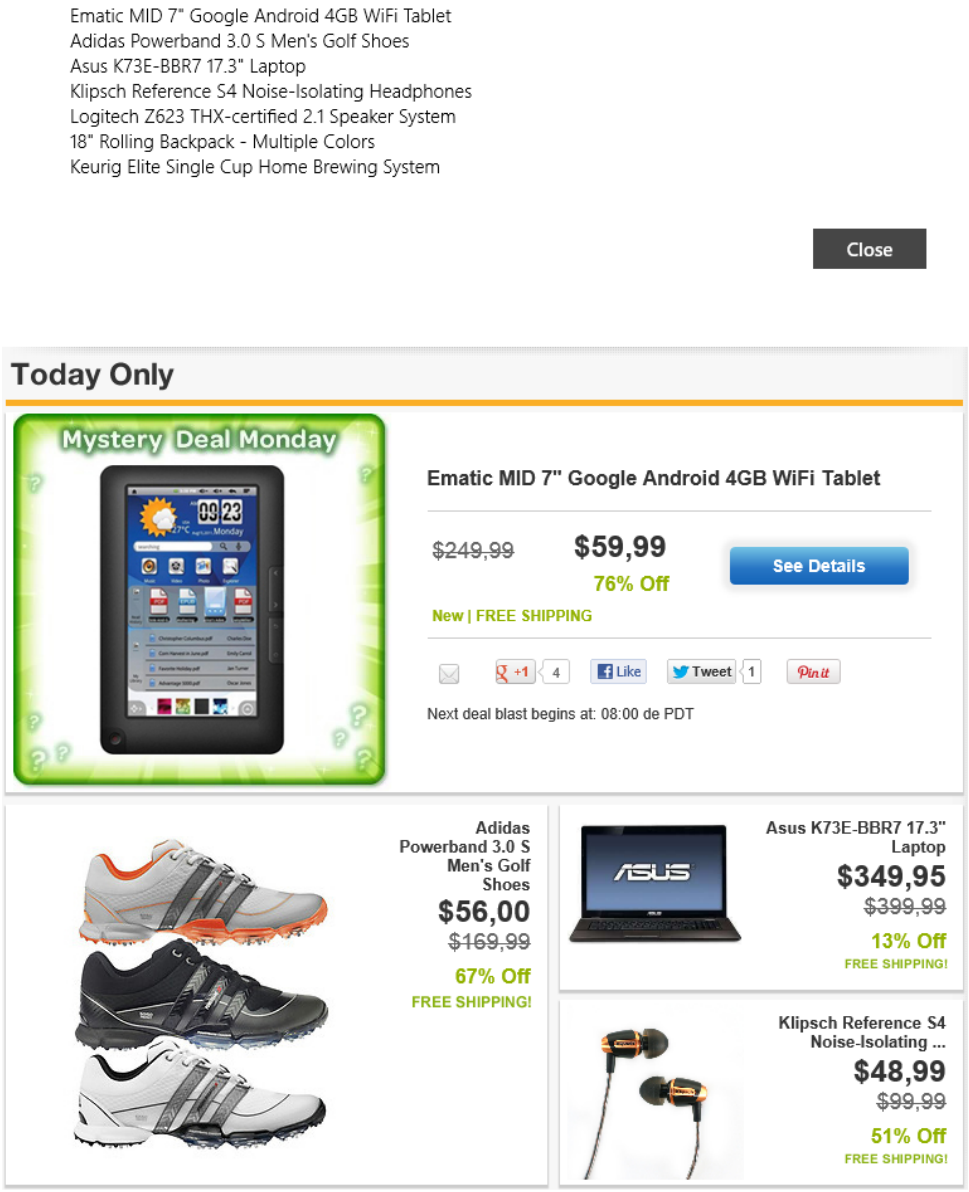
Ha úgy találjuk, hogy minden objektum megérkezett, akkor egy üzenetdobozban kiíratjuk az aktuális eBay akciók címeit. Az esetleges hibát pedig természetesen kezeljük.

7. Írd be az alábbi eseménykezelő kódot a **public MainPage()** konstruktor alá:

```
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;
            (new MessageDialog(message)).ShowAsync();
        }
    }
    else
    {
        (new MessageDialog(e.Error.ToString())).ShowAsync();
    }
}
```

Ezzel elkészültünk. Teszteljük alkalmazásunkat!

8. Indítsd el az F5-tel, és kisvártatva megjelennek az aktuális eBay akciók. Az eBay honlapján ellenőrizheted is, hogy jókat töltött-e le – a http://deals.ebay.com/ honlapot megnyitva láthatod, hogy valóban az épp elérhető akciókat kérdezted-e le (11-20 ábra).



11-20 ábra: Az OData-n keresztül letöltött objektumok és az eBay honlapja

Hogyan működik?

Az alkalmazás a futtatókörnyezettel letölteti az OData végpont adatait, melyek „nyers" formájukban XML adatok. Ezeket azonban a C# környezetnek köszönhetően a fejlesztő már C# objektumokként látja, így ugyan az alkalmazás a teljesen platformfüggetlen OData protokollon keresztül kommunikál, fejlesztői szempontból mégis C# objektumokról van szó.

Ezzel megismerted az OData használatának alapjait. Az itt megtanultak alapján már el tudsz készíteni egy OData-fogyasztásra képes alkalmazást. A webszolgáltatásokhoz hasonlóan az OData is rendkívül széles témakör. Ha szeretnéd jobban megismerni az OData protokollt, látogass 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 fórum. Érdemes megismerkedni a protokollal, mert rengeteg Microsoft-termék és számos külső gyártó már most támogatja: ilyen például az Entity Framework, a SharePoint, az eBay, a Netflix és még sokan mások.

Összegzés

Egy Windows 8 alkalmazásfejlesztőnek feltétlenül ismernie kell azokat a módszereket, amelyekkel alkalmazása kommunikálhat a külvilággal, az internettel. Így tud ugyanis igazán naprakész, interaktív programokat készíteni. A fejezetben a legfontosabb ilyen módszereket ismerhetted meg.

Ha arra van szükséged, hogy üzenetet válts egy szerverrel (például letölts híreket, feltölts felhasználói választásokat stb.), kiváló és egyszerűen használható választást jelentenek a webszolgáltatások.

A Windows 8 világban nagy hangsúlyt kap a Live ID (Microsoft Account). Ehhez kötődően a felhasználók megadhatják (és sokszor meg is adják) nevüket, fényképüket; ezen keresztül levelezhetnek, naptárt és feladatokat kezelhetnek; a Live ID-n keresztül elérhető SkyDrive tárhelyre fájlokat tölthetnek fel és így tovább. Ha ehhez az igen gazdag információtárhoz szeretnél hozzáférni, akkor használd a Live SDK-t!

A Windows 8 erőforráskímélő alkalmazásmodellje leállítja az alkalmazásokat, ha azok a háttérbe kerülnek. Az élet azonban ekkor sem áll meg: megfelelő privilégiumok birtokában alkalmazásod folyamatosan naprakészen tarthatja magát a valós idejű kommunikáció és a háttérben futó feladatok segítségével.

Ha szerveroldali erőforrásokra van szükséged, akkor a legjobb választás a felhő: költséghatékonyan vásárolhatsz mindenféle kapacitást (webszervert, tárhelyet, SQL adatbázist), ráadásul nem kell küzdened a szerverek adminisztrációjával sem, ha pedig alkalmazásod „nagyot durran”, és óriási népszerűségre tesz szert, a felhő képes lesz kiszolgálni a megugró igényeket is. Windows 8 alkalmazásokból könnyen együttműködhetsz a Windows Azure felhőplatformmal.

Végül, de nem utolsósorban az OData technológia segítségével rengeteg gyártó és tartalomszolgáltató online erőforrásaihoz férhetsz hozzá – a Windows 8 alkalmazások fejlesztőkörnyezete ezt is támogatja.

12. A C++ programozási nyelv és a Windows 8 alkalmazások

Ebben a fejezetben az alábbi témákat ismerheted meg:

- Melyek azok a helyzetek, amikor a C++ programozási nyelv a legjobb választás?
- A C++ nyelv újdonságai
- Melyek azok az új nyelvi képességek, amelyek a Windows 8 stílusú alkalmazások fejlesztését biztosítják?

Bár ennek a fejezetnek a címe azt sugallja, hogy ez csak C++ fejlesztőknek szól, ez nincs így! Ha C#, Visual Basic vagy éppen JavaScript fejlesztő vagy, ez a fejezet segít megérteni, hogy milyen helyzetekben a lehetséges legjobb választás a C++ programozási nyelv Windows 8 stílusú alkalmazások készítéshez. Ha már volt C++-os tapasztalatod a múltban – még ha akkoriban frusztráló is volt –, ez a fejezet demonstrálni fogja, hogy a C++ ma már egy modern, letisztult és biztonságos programozási nyelv, első osztályú választás egy Windows 8 stílusú alkalmazás fejlesztéséhez.

A fejezet elején megtanulhatod, hogy milyen dolgok állnak a C++ nyelv reneszánsza mögött, majd áttekintést kapsz a nyelv legutóbbi fejlődéséről. A fejezet legnagyobb részében azokat az új képességeket ismerheted meg, amelyeket a Microsoft azért adott a Visual Studio 2012-es C++ implementációjához, hogy támogassa a Windows 8 stílusú alkalmazások fejlesztését és a Windows Runtime-mal való integrációt. Miután megismerted ezeket, néhány egyszerű mintapéldán keresztül ellenőrizheted le, hogyan is működik mindez a gyakorlatban.

> *Ennek a fejezetnek távolról sem célja, hogy megtanítsa neked a C++ programozási nyelvet.*

A Microsoft és a C++ programozási nyelv

Bár a körülmények azt a látszatot kelthetik, hogy a .NET nyelvek (a C# és a Visual Basic) a Microsoft által használt elsődleges programozási nyelvek – és így a cég ezeket használja leginkább a termékeiben –, ez még sincs így. A Microsofton belül a legtöbb termékfejlesztési csapat még mindig a C++ nyelvvel dolgozik, és ezek a projektek még hosszú ideig ezt a nyelvet fogják használni.

Miért is van ez így? A Microsoft talán nem hisz a saját felügyelt kódú futásidejű környezetében (a .NET keretrendszerben) és ezért nem használja? Esetleg nem tudják vagy nem akarják kihasználni a .NET produktivitását? Nem erről van szó! A C++ két fontos tulajdonsággal bír, amelyek a felügyelt programozási nyelvek számára komoly kihívást jelentenek. Ezek a teljesítmény és a közvetlen kontroll a rendszer erőforrásai felett.

Amikor a .NET keretrendszer felügyelt nyelveivel dolgozol, a fordítóprogram olyan végrehajtható állományt hoz létre, amely MSIL (Microsoft Intermediate Language) kódot tartalmaz, vagyis egy közbenső nyelv utasításait. Amikor ez a program fut, az MSIL utasítások röptében CPU-specifikus utasításokra fordulnak a JIT (Just-In-Time) fordítóval, és ez a CPU-specifikus kód kerül végrehajtásra. Bár a felügyelt nyelvek és az MSIL olyan konstrukciókat tartalmaznak, amelyek a szoftverfejlesztést hatékonyabbá teszik, igen körülményes az olyan alacsony szintű utasítások, mint például a bitszintű műveletek leírása, amelyeket egyébként nagyon egyszerű CPU-specifikus utasításként megadni. Ott, ahol egy algoritmus teljesítménye jelentősen javítható alacsony szintű konstrukciókkal, a felügyelt nyelvek kihívásokkal szembesülnek.

Természetesen a natív nyelvek, így a C++ használatának is vannak hátulütői. Ahhoz, hogy minden szinten kezedben tarthasd a teljesítményt, gyakran apróságokkal is foglalkozni kell. Például, ha a memóriahasználat

felett szeretnél kontrollt gyakorolni, neked kell explicit módon lefoglalni és felszabadítani a memóriát. Ez a tevékenység rendkívül sok odafigyelést kíván, hogy elkerülhesd a memória „szivárgását", illetve a már felszabadított erőforrásokra való hivatkozást.

Korábban már megtanulhattad, hogy a Windows korszak kezdete a C és C++ programozásról szólt. Abban az időben a fejlesztők termelékenysége igen alacsony volt a 2002-ben a .NET-tel felbukkanó felügyelt nyelvekkel szemben. A felügyelt nyelvek mindig kicsit alacsonyabb teljesítményt jelentettek cserébe a produktivitásért. Erős hardver mellett – különösen a szerveroldalon – ez hosszú ideig elfogadható kompromisszum volt, mert kb. 20%-kal drágább hardver megvásárlása még mindig olcsóbb volt, mint az alkalmazás teljesítményhangolását végző programozókat néhány hónapig tovább fizetni.

Az okos telefonok, az újgenerációs táblagépek és az ultramobil eszközök azonban teljesen megváltoztatták a játékszabályokat. Az ezekben az eszközökben található CPU-k sokkal kisebb teljesítményűek, mint az asztali számítógépekben találhatók, és így ugyanazoknak az algoritmusoknak a futása hosszabb ideig tarthat. Ezek az eszközök meglehetősen korlátos kapacitású akkumulátorral működnek, azért, hogy súlyuk alacsonyan tartható legyen. Minél nagyobb teljesítményre van szükséged, annál rövidebb ideig tart ki az akkumulátor egyetlen feltöltéssel. Ráadásul mindezek mellett a fogyasztók kiemelkedő felhasználói élményt, villámgyorsan reagáló felhasználói felületet szeretnének látni, nem pedig akadozva futó alkalmazásokat.

Ezeken a mobil eszközökön az ilyen „jól viselkedő" alkalmazások írásának kulcsa a hatékonyság. Az alkalmazásoknak a lehető legrövidebb idő alatt a lehető legtöbb tevékenységet kell elvégezniük olyan kevés CPU-használat mellett, ami csak lehetséges. Takarékosan kell bánniuk a memóriával és az egyéb hardveres erőforrásokkal. Ez az a pont, ahol a natív kód eredményesen alkalmazható, mert ezek azok az elvárások, ahol a natív programozási nyelvek – közöttük a C++ – nyújtják a legjobb teljesítményt.

Így talán nem meglepő, hogy a Microsoft a C++ nyelvet a Windows 8 stílusú alkalmazások támogatásának egyik elsődleges programozási nyelveként akarta támogatni – és ezen a területen elképesztő munkát végeztek! Nemcsak egyszerűen megoldották a Windows 8 stílusú alkalmazások fejlesztését C++-ban, de egyúttal modernizálták is a nyelvet: tisztává és biztonságossá tették!

Tiszta és biztonságos

A legtöbb programozó vagy nagyon szereti, vagy nagyon utálja a C++ nyelvet. Nincsenek olyanok, akik azt mondanák: „egy kicsit szeretem" vagy „rendben van, de nem szeretem azt a képességet, hogy…". A C++ hívei azért szeretik, mert ez a nyelv teljes kontrollt biztosít az alkalmazás végrehajtása felett. Egyfajta izgalmat jelent majdnem mindig előre tudni, hogy milyen utasítások fognak a CPU-n végrehajtásra kerülni egy adott C++ programozási konstrukció eredményeképpen. Azok, akik nem szeretik a C++-t, elsősorban az alacsony szintű konstrukciókat és az abból eredő gyakori buktatókat utálják.

A C++ egyik legújabb implementációjában – amelyet a Visual Studio 2012-ben is megtalálhatsz – a Microsoft rengeteg energiát fektetett abba, hogy a C++-t a termelékenység javításával modernebb nyelvvé alakítsa, miközben megtartja az alkalmazás végrehajtása felett a teljes kontroll lehetőségét. Számtalan új C++11 képességet valósítottak meg, sőt, a Microsoft kiterjesztette a nyelvet a Windows 8 stílusú alkalmazások és a Windows Runtime integráció támogatására.

> ***A C++11***
>
> *A C++ programozási nyelvet Bjarne Stroustrup fejlesztette ki 1979-ben a Bell Laboratóriumban. Abban az időben ezt az új nyelvet a „C – osztályokkal" néven említették. A nyelv az első szabványosítási folyamatán 1998-ban ment keresztül (C++98), majd később 2003-ban (C++03). A nyelv szabványának utolsó alapvető átvizsgálása a C++11 volt, amelyet az ISO/IEC alig egy évvel ezelőtt, 2011. augusztus 12-én fogadott el.*

Hogy megértsd, milyen képességek tették produktívabbá a nyelvet a korábbi változatainál, nézzünk meg egy példát, amely a C++03 (vagyis az előző C++ szabvány) használatával készült, amint azt a 12-1 kódlista mutatja be.

12-1 kódlista: Egyszerű c++ program – a régi stílusban megírva

```
// --- Create a vector of 10 rectangles
vector<rectangle *> shapes;
for (int i = 1; i <= 10; i++)
{
    shapes.push_back(new rectangle(i * 100, i * 200));
}

// --- This is the rectangle we are looking for
rectangle* searchFor = new rectangle(300, 600);

// --- Iterate through all rectanges
for (vector<rectangle*>::iterator i = shapes.begin(); i != shapes.end(); i++)
{
    (*i)->draw();
    if (*i && **i == *searchFor)
    {
        cout << "*** Rectangle found." << endl;
    }
}

// --- Dispose resources held by this program
for (vector<rectangle*>::iterator i = shapes.begin(); i != shapes.end(); i++)
{
    delete *i;
}
delete searchFor;
```

Ez a program tíz **rectangle** típusú objektum vektorát hozza létre (ezek a **shapes** változóban tárolódnak), majd egy **for**-ciklusban kirajzolja őket, és leellenőrzi, hogy bármelyikük egyezik-e azzal, amely a **searchFor** változóban van letárolva. Néhány dolog zavaró komplexitást ad ehhez az egyszerű feladathoz:

- A **shapes** változó olyan vektor, amely **rectangle** objektumokra hivatkozó mutatókból épül fel. Az első ciklus egytől tízig haladva **rectangle** objektumokat hoz létre, és a **shapes** vektorhoz adja őket. A programozónak fejben kell tartania, hogy ezeket az objektumokat, majd a program egy adott pontján fel kell szabadítania. Az utolsó **for**-ciklus a **delete i** törzzsel éppen ezt a takarítást végzi el.
- Hasonlóképp, a **searchFor** változó is egy **rectangle** objektumra hivatkozó mutatót tárol. Ez az objektum az utolsó sorban lévő **delete searchFor** utasítással kerül felszabadításra.
- Két **for**-ciklus is iterátort használ, hogy végighaladjon a vektor elemein. Bár ezeknek a ciklusoknak a szándéka világos, az ezt kifejező kód mégis feleslegesen hosszúnak néz ki.
- Az ***i && **i == *searchFor** feltételes kifejezés ugyan egyszerű, de mégis nehéz azt kibogozni. Ebben a kifejezésben az **i** egy iterátor, és a ***i** azt a **rectangle** objektumot jelölő mutatót reprezentálja, amelyre az iterátor éppen hivatkozik. A ****i** kifejezés az a **rectangle**, amelyet az iterátor éppen megszemélyesít, a ***searchFor** pedig az a **rectangle** objektum, amelyre a ciklus keres. Ez a feltétel azt mondja, hogy ha az iterátor éppen egy **rectangle** típusú objektumra mutat, és az megegyezik azzal, amit keresünk, egyezésről van szó. Bár a kifejezés jelentése a tapasztalt C++ fejlesztők számára egyértelmű, mégis nehezen olvasható.
- Egy ilyen programban potenciális buktató lehet, hogy a programozónak kell a teljes felügyeletet biztosítania a memória felett, ahol az objektumok tárolódnak. Nem mindig teljesen egyértelmű, hogy kinek a felelőssége lehet a **shapes** vektorban tárolt **rectangle** objektumok „eltakarítása”. A 12-1 kódlistában ez éppen nyilvánvalónak tűnik abból, hogy a kód explicit módon felszabadítja az objektumokat, de ha csak az allokációt végző **for**-ciklust látnád, ez nem lenne ilyen egyértelmű.

Az új – és a Visual Studio 2012-ben már elérhető – C++ nyelvben ez a program sokkal elegánsabban írható le, amint azt a 12-2 kódlista is mutatja.

12-2 kódlista: Egyszerű C++ program – az új stílus használatával

```
// --- Create a vector of 10 rectangles
vector<shared_ptr<rectangle>> shapes;
for (int i = 1; i <= 10; i++)
{
    shapes.push_back(shared_ptr<rectangle>(
        new rectangle(i * 100, i * 200)));
}

// --- This is the rectangle we are looking for
auto searchFor = make_shared<rectangle>(300, 600);

// --- Iterate through all rectanges
for_each(begin(shapes), end(shapes), [&](shared_ptr<rectangle>& rect)
{
    rect->draw();
    if (rect && *rect == *searchFor)
    {
        cout << "*** Rectangle found." << endl;
}
});
```

Ez a program néhány olyan elemet tartalmaz, amelyek megszüntetik a 12-1 kódlistában lévő problémákat:

- A **shapes** vektor referenciaszámlálással kezelt **rectangle** objektumokat tartalmaz (erre utal a **shared_ptr** deklaráció). Egy vagy több objektum hivatkozhat egy mutatóval ugyanazon **rectangle** objektumra. Amikor az utolsó hivatkozás is megszűnik, amely a **rectangle** objektumra mutat, az objektum törlésre kerül.
- Az **auto** kulcsszó a **searchFor** változó előtt automatikusan kikövetkezteti ennek a változónak a típusát az inicializáláshoz használt kifejezésből. Egyébként tudnod kellene, hogy a **make_shared<rectangle>** deklaráció egy **shared_ptr<rectangle>** típusú objektumot hoz létre, és ezt a típust kellene használnod az **auto** kulcsszó helyett.
- A **for_each** metódus nyilvánvalóvá teszi, hogy olyan ciklusról van szó, amely a **shapes** vektor összes elemén végighalad. A művelet első két argumentuma, a **begin(shapes)** és az **end(shapes)** világossá teszik, hogy a ciklus az elsőtől az utolsó elemig halad. A harmadik argumentum egy lambda kifejezés – a C++11 egy új képessége –, amely kirajzolja a **rectangle** objektumot, és elvégzi az egyezés ellenőrzését. A lambda kifejezés törzsét összehasonlítva a 12-1 kódlistában szereplő kifejezéssel felismerheted, hogy itt a **rect** referencia egy **rectangle** objektumra, és így a ***rect == *searchFor** kifejezést sokkal egyszerűbb megérteni, mint a ****i == *searchFor** kifejezést.
- A 12-2 kódlista egyetlen utasítást sem tartalmaz, amely a **shapes**-ben tárolt **rectangle** objektumokhoz kapcsolódóan explicit módon felszabadítaná a memóriát. A **sharep_ptr** típus automatikusan elvégzi ezt, így a fejlesztőnek nem kell ezzel explicit módon foglalkoznia.

Ez a rövid kódrészlet csak egyetlen példa volt a C++ nyelvet modernné és produktívvá tevő képességeknek a hasznosságára. A Microsoft számos ilyen képességet valósított meg a C++ fordítóprogramjában, megfelelve a C++ 11 szabványnak.

A C++ programozási nyelv legfontosabb változásai

Már hosszú idő eltelt a 2003-ban kibocsátott C++03 szabvány felülvizsgálata óta, és az új C++ 11 szabványt az azt kezelő testület 2011 augusztusában hagyta jóvá. Az eltelt idő alatt a szoftverfejlesztés világa sokat változott, és a C++ nyelvnek is igazodnia kellett azokhoz. Bár a Microsoft csak egy részét valósította meg a C++11 képességeknek, a Visual Studióban azokat használhatod, amelyek a legnagyobb fejlődést jelentik a produktivitás terén. Itt a legfontosabb új képességekkel ismerkedhetsz meg.

A C++ itt bemutatott változásai nem sokat mondanak neked, ha még sohasem használtad a C++ nyelvet. Ugyanakkor, ha gondolkozol ennek a régi-új programozási nyelvnek az elsajátításán, ezekből a változásokból felmérheted a nyelv fejlődését.

Deklarációk

Korábban az **auto** kulcsszó egy tárolási osztály specifikálására szolgált, de a C++11-ben új értelmezést nyert. Automatikus típuskövetkeztetésre használható, feltéve, hogy egy deklaráció inicializáló kifejezést tartalmaz. A fordítóprogram kikövetkezteti a változó típusát az inicializálásból:

```
auto myFavoriteNumber = 8;
auto myPtr = &myFavoriteNumber;
double sum(double a; duble b);
auto myFunction = sum;
```

Ebben a kódrészletben a **myFavoriteNumber** változó **integer** értékként kerül deklarálásra. Mivel a **myPtr** a **myFavoriteNumber** címeként inicializálódik, ezért típusa **int *** (mutató egy **integer**re) lesz. A **myFunction** változó olyan függvény lesz, amely két **double** típusú argumentumot vár és egy **double** értéket ad vissza, mert ez a **sum** típusa.

A C++11 bevezeti a **decltype** kulcsszót, amely egy változó típusát egy megadott kifejezés típusaként hozza létre. Ezzel az alábbihoz hasonló deklarációkat írhatsz le:

```
int a;
double b;
decltype(a*b) c;
```

Mivel egy **integer** érték és egy **double** érték szorzata **double**-t eredményez, **c** egy **double** típusú változó lesz. A **decltype** kifejezett előnye akkor jön elő, ha azt egy **template** definíciójában használjuk:

```
template<typename T1, typename T2>
void MyFunction(T1 ta, T2 t2)
{
    decltype(T1*T2) n;
    ...
}
```

Ebben az esetben az n típusát mindaddig nem tudjuk meghatározni, amíg egy konkrét példányosítás nem történik (például egy olyan, ahol **T1** egy **integer**, **T2** pedig egy **double**). A **decltype(T1*T1) n** kifejezés azt jelenti, hogy **n** típusa a **T1*T2** kifejezés típusával egyezik meg, bármilyen típus is származik ebből a műveletből – feltéve, hogy a ***** operátor értelmezett **T1** és **T2** között.

A C++11 új szintakszist vezet be a függvények visszatérési értékének deklarálására. Ez az ún. *követő visszatérési típus*, mivel ez a jelölés ezt a visszatérési típust a függvény neve és paraméterlistája után helyezi el:

```
double myFunction(int arg1, double arg2);         // Traditional definition
auto myFunction(int arg1, double arg2) -> double; // Trailing return type
```

Első pillantásra nem világos, hogy miért is van szükség erre az új szintakszisra, ez inkább visszalépésnek tűnik. Azonban e nélkül a jelölés nélkül nem lenne lehetőség a **decltype** konstrukció használatára **template** függvények visszatérési értékének megadására:

```
template<typename T1, typename T2>
auto MyFunction(T1 ta, T2 t2) -> decltype(T1*T2)
{
    ...
}
```

Új konténerek a Standard Template Library-ban

A Standard Template Library (STL) C++ osztályok újrafelhasználható gyűjteménye. Négy összetevőt tartalmaz, ezek az algoritmusok, konténerek, funkcionális elemek és iterátorok. A C++11 specifikáció új elemeket ad az STL-hez. A 12-1 táblázat az STL új konténereit foglalja össze.

12-1 táblázat: Az STL új konténerei

Konténer	Leírás
forward_list	Ez a lista egy olyan konténer, amely lehetővé teszi az elemek gyors beszúrását és gyors törlését a konténeren belül bárhova. Ugyanakkor nem támogatja a gyors, véletlenszerű hozzáférést, bár a konténer elemein – az elsőtől az utolsóig – gyorsan végig lehet haladni.
unordered_map	Ez a konténer asszociatív tár, amely kulcs és érték párokat tartalmaz egyedi kulcsokkal.
unordered_multimap	Ez a konténer az **unordered_map**-hez hasonló, de lehetővé teszi több érték tárolását ugyanahhoz a kulcshoz. Egy olyan iterátort is tartalmaz, amellyel végig lehet haladni egy adott kulcshoz tartozó értékeken.
unordered_set	Ez a konténer adott típusú egyedi értékek asszociatív halmaza. Támogatja a gyors keresést, illetve a beszúrás és törlés műveleteket átlagosan konstans idejű komplexitással.
unordered_multiset	Ez a konténer adott típusú, potenciálisan nem egyedi értékek asszociatív halmaza. Támogatja a gyors keresést, illetve a beszúrás és törlés műveleteket átlagosan konstans idejű komplexitással.

Ezek a konténerek a teljesítményre optimalizáltak.

A legtöbb STL konténer (beleértve a régieket, és nemcsak a 12-1 táblázatban szereplő újakat) támogatja a **begin()**, **end()**, **rbegin()** és **rend()** műveleteket. Amint azt a 12-1 és 12-2 kódlistákban már láthattad, a **begin()** és **end()** műveletekkel a konténer elemein haladhatsz végig, mégpedig az elsőtől az utolsó elemig. Az **rbegin()** és **rend()** műveletekkel az elemeken fordított sorrendben, vagyis az utolsótól az elsőig haladhatsz végig. Az STL új műveleteket is kapott, ezek a **cbegin()**, **cend()**, **crbegin()** és a **crend()**. Ezeknek a műveleteknek a szemantikája megegyezik a hasonló nevű, de **c** prefix nélküli műveletekével. A **c** a műveletek nevében arra utal, hogy ezek az iterátorok konstans elemeket használnak, vagyis lekérdezheted, de nem módosíthatod azokat.

„Okos" mutatók

A C++ programok a **new** kulcsszó segítségével foglalhatnak dinamikusan memóriát. A programozó felelőssége, hogy a **delete** utasítással felszabadítsa a memóriát, amikor már nincsen rá szüksége. Ennek a folyamatnak az automatizálására a C++ bevezette az **auto_ptr** típusú mutatót, azonban ez a megoldás nem volt elég kifinomult. A C++11 az **auto_ptr** használata helyett három új „okos" mutatót vezetett be, amelyeket a 12-2 táblázat mutat be.

12-2 táblázat: A C++11 új „okos” mutatói

Mutató	Leírás
unique_ptr	Ez a mutató egy objektum egyedi birtoklását reprezentálja. Egy **unique_ptr** nem másolható, nem létezik két példány, amely ugyanarra az objektumra mutat. Az objektum tulajdonjoga azonban átadható.
shared_ptr	Ez a mutató egy objektum osztott birtoklását reprezentálja. Egy vagy több **sharep_ptr** példány is mutathat ugyanarra az objektumra. Ha az utolsó mutató is megszüntetésre kerül, amely az objektumra mutat, akkor az objektum is törlésre, az általa foglalt memória felszabadításra kerül.
weak_ptr	Ez a mutató egy gyenge (tulajdonlással nem járó) hivatkozást kínál egy **shared_ptr** által kezelt objektumra. Ideiglenes birtoklást modellez, amikor egy objektum elérésére csak akkor van szükség, ha az objektum ténylegesen létezik, és az bármikor, bárki által törölhető.

Rvalue hivatkozások

A C++ hagyományosan ún. *lvalue* (left-hand side value) hivatkozásokat használ egy azonosító értékhez kötésére. Egy lvalue egy kifejezés, pl. egy változó neve, amely a program által elérhető memóriacímmel rendelkező adatot reprezentál. Például a következő kódrészletben az **addrn** változó az **n** változó memóriacímére van beállítva:

```
int n = 11;
int & addrn = n;
```

A C++11 az && jelölést használja ún. *rvalue* (right-hand side value) hivatkozások azonosítókhoz kötéséhez. A következő kódrészlet ezt mutatja be:

```
int n = 11;
int & addrn = n;
int && addrnp5 = n + 5;

n = 12;
cout << "Value of n: " << addrn << endl;
cout << "Value of m: " << addrnp5 << endl;
```

Mivel az **addrn** egy lvalue hivatkozás, az **addrn** tartalmának a kiírása 12-t eredményez, hiszen az **n** erre az értékre van beállítva közvetlenül a kiírás előtt. Azonban az **addrnp5** már egy rvalue hivatkozás, és így az az **n+5** értékre mutat, amely azonnal, már a deklarálás időpontjában kiértékelésre kerül, is érintetlen marad azután is, hogy az **n** értéke 12-re lesz állítva. Így az **addrnp5** kiírása 16-ot (11+5) eredményez.

Az rvalue hivatkozások önmagukban nem túl hasznosak, de nagyon hatékonyak a C++11 új, ún. mozgatási szemantikájával.

Mozgatási szemantika

A C++ az értékeket lemásolja, amikor egy függvénynek argumentumot ad át, vagy egy függvény értékével visszatér. Tegyük fel, hogy van egy **customer** objektumokat tartalmazó vektorod, amelyben minden objektum mérete 1000 byte, és egy művelet (**LoadCustomers**) 100 000 **customer** adatot olvas fel a diszkről:

```
vector<customer> customerVector;
...
vector<customer> LoadCustomer();
```

Tegyük fel, hogy a **LoadCustomer** művelet dinamikusan foglalja a memóriát, és a metódus törzsén belül felolvas minden egyes **customer** példányt, és azokat a **new** operátorral példányosítja. Amikor az eredményt a **customerVector**-ban letárolja, minden – a művelet által felolvasott – **customer** adatot a **vector<customer>** osztály másolást végző (copy) konstruktorával lemásolja:

```
vector<customer>:
vector<customer> = LoadCustomer();
```

Összességében ez 100 millió bájt felesleges lemásolását jelenti. Miután a **LoadCustomer** visszatér, a metódus már nem fogja a továbbiakban az általa létrehozott eredményt használni, elegendő volna, ha az eredmény birtoklásának joga egyszerűen átszállna a hívóra az adatok másolása nélkül.

A C++11 bevezeti az ún. mozgatási szemantikát, hogy javítsa az ehhez hasonló szituációkban az alkalmazások teljesítményét. Egy új típusú konstruktor, a mozgató (move) konstruktor használatával megvalósíthatod ezt a mozgatási szemantikát. Ez a konstruktor egy rvalue értéket fogad argumentumként, például a **vector<customer>** típus esetében a prototípusa az alábbi módon néz ki:

```
vector<customer>(vector<customer> && otherVector);
```

A mozgató konstruktornak egyszerűen át kell vennie az argumentumként megadott rvalue tulajdonjogát. Például **vector** esetében a mutatót, amely az elemek tárolási területére mutat, egyszerűen hozzá kell rendelni az új objektumhoz. Az **otherVector** objektum hasonló szerepkörű mutatóját **null** értékre kell állítani, jelezve, hogy a továbbiakban már nem az a tulajdonosa a kérdéses memóriaterületnek.

> *További részleteket is megtudhatsz a rvalue hivatkozásokról és a mozgatási szemantikáról az alábbi MSDN lapokon: http://msdn.microsoft.com/en-us/library/dd293665.aspx, http://msdn.microsoft.com/en-us/library/dd293668.aspx*

Lambda kifejezések

A C++ ún. *functor* objektumokat biztosít, amelyek úgy használhatók, mintha függvények lennének, bár valójában olyan objektumok, amelyek állapotot tárolnak. Például a következő functor segít azoknak az egész számoknak az összeszámlálásában, amelyek 7-tel oszthatók:

```
class Div7
{
private:
    int& _count;

public:
    explicit Div7(int& count)
    void operator()(int number)
    {
        if (n % 7 == 0)
        {
            _count++;
        }
    }

};
```

A **Div7** functor segítségével egyszerű programot írhatsz, amely végrehajtja a számlálási folyamatot:

```
int main()
{
    vector<int> numbers;
    for (int i = 0; i < 1000; i++) numbers.push_back(i);
    int count = 0;
    for_each(number.begin(), numbers.end(), Div7(count));
    cout << "There are " << count << "number(s) that can be divided by 7.";
}
```

A Div7 functornak két feladata is van. Először is megkapja a **count** változót, amelyet a **main()** függvénybe adtunk át neki, hogy tárolja a számlálót. Másodszor megvalósítja az **operator()** metódust, amely az ellenőrzést végzi el.

A C++11 lambda kifejezések segítenek a functorok olyan névtelen függvényekké való átalakításában, amelyek képesek az állapotuk kezelésére és a hatókörükben lévő változók elérésére. Egy lambda kifejezést használva a Div7 functor eliminálható, amint azt az alábbi kódrészlet is bemutatja:

```
int main()
{
    vector<int> numbers;
    for (int i = 0; i < 1000; i++) numbers.push_back(i);
    int count = 0;
    for_each(number.begin(), numbers.end(), [&count](int number)
    {
        if (number % 7 == 0) count++;
    });
    cout << "There are " << count << "number(s) that can be divided by 7.";
}
```

A lambda kifejezés használata egyszerű. Az **(int number)** rész definiálja a függvény input argumentumát. A kapcsos zárójelek közötti rész ellenőrzi le, hogy a bemenő érték osztható-e 7-tel. A **[&count]** deklaráció azt írja le, hogy a **main()** metódus **count** változóját a lambda kifejezés törzse is eléri, és így a **count++** művelet ennek a változónak az értékét növeli.

Új C++ típusok

Amikor a C és C++ nyelveket megalkották, a 64-bites operációs rendszerek egyáltalán nem voltak még elterjedtek. A C++ bevezeti a **long long** és az **unsigned long long** típusokat, amelyek az előjeles és előjel nélküli 64-bites (vagy akár szélesebb) egész értékek kezelését támogatják.

A C++11 két új karakteres típussal is támogatja a Unicode szabványt. Ezek a **char16_t** és a **char32_t** típusok, amelyek 16-bites, illetve 32-bites karaktereket reprezentálnak.

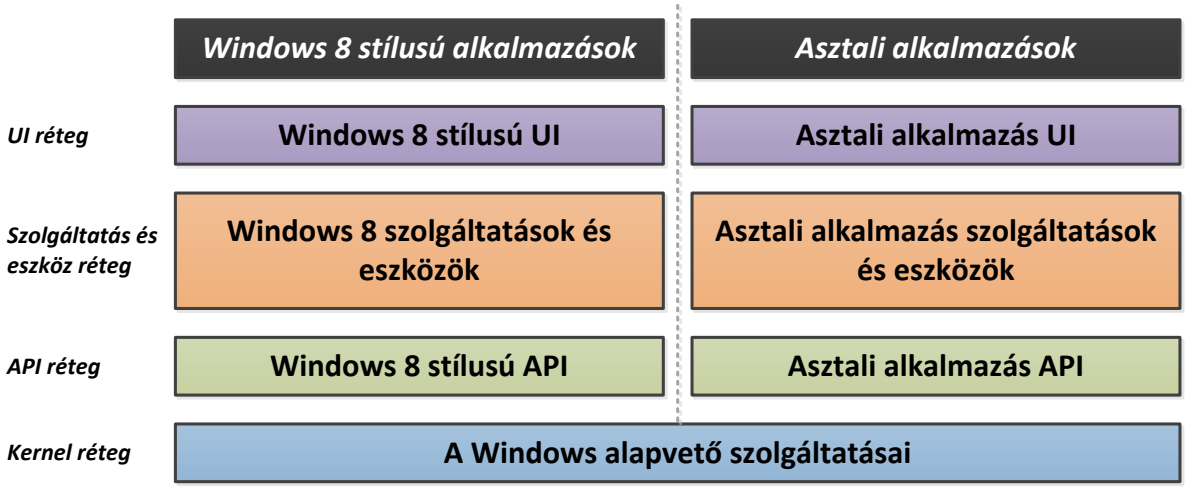
> *Ez a fejezetrész nem sorolt fel minden a Visual Studio 2012-ben megvalósított C++11 képességet. További részletes információt találhatsz az újdonságokról a Visual C++ csapat MSDN blogján: http://blogs.msdn.com/b/vcblog/archive/2011/09/12/10209291.aspx*

A C++11 néhány újdonságának áttekintése után itt az ideje, hogy megismerkedj azokkal az új nyelvi képességekkel, amelyek lehetővé teszik a Windows 8 stílusú alkalmazások fejlesztését a C++ programozási nyelvvel.

Windows 8 stílusú alkalmazások készítése C++ programozási nyelven

A 3. fejezetben már megtanultad, hogy a Windows 8 stílusú alkalmazások számára készített, független komponenskészlet létrehozásával a Microsoft új koncepciót vezetett be, amely újraértelmezi a Windows API-k és a nyelvi futtatókörnyezetek fogalmát. Ezek a komponensek – amint azt a 12-1 ábra is ábrázolja –

lehetővé teszik, hogy a C++, C#, Visual Basic és HTML5/CSS3/JavaScript technológiák mind felhasználhatók legyenek a Windows 8 stílusú alkalmazások készítésében.



12-1 ábra: A Windows 8 stílusú alkalmazások komponenskészlete

Nézzünk az ábra mögé, és ismerjük meg, hogy milyen Windows 8 stílusú alkalmazások valósíthatók meg a C++ nyelv közvetlen vagy közvetett felhasználásával!

A C++ programozási nyelv privilégiumai a Windows 8 stílusú alkalmazásokban

A 12-1 ábra úgy mutatja, mintha a négy programozás nyelv egyenrangú lenne, de valójában a C++ „egyenlőbb" a többieknél. A C++ nyelv rendelkezik néhány olyan privilégiummal (például a Windows 8 rendszerszintű erőforrásainak kezelése), amelyek a többi programozási nyelvben elérhetetlenek.

Létezik a Windows 32 API szolgáltatásainak egy keskeny halmaza, amely a C++-ból még mindig használható. Ezek nem a Windows Runtime rétegen keresztül hívhatók, hanem a rendszer dinamikus könyvtáraiban (DLL-ekben) találhatók.

A DirectX technológiák (Direct2D, Direct3D, DirectWrite, XAudio2 és így tovább) könnyedén felhasználhatók a C++-ból. Ezek közvetlenül a GPU-t és a hangeszközöket használják – lenyűgöző teljesítménnyel.

Egy teljesen új technológia, a C++ Accelerated Massive Parallelism (C++ AMP) lehetővé teszi, hogy a mai párhuzamosított architektúrára épülő hardver eszközöket – például a grafikus processzorokat és a hangfeldolgozó egységeket – képességeiknek megfelelően használd ki. A C++ AMP segítségével programjaid egyes részeit a C++ program közvetlenül a számítógépedben található grafikus processzoron futtatja!

Amint arra a 12-1 ábra rámutat, a C++ programozási nyelvet alkalmazásod egészének elkészítésére használhatod a XAML technológiával. Sőt, ezenkívül még van néhány olyan eset, amikor a C++ programozási nyelv használata szintén jó választást jelenthet:

- Lehetőséged van olyan Windows 8 stílusú játékprogramok készítésére, amelyek a DirectX-et használják a C++-szal. Mivel a legtöbb játéknak az utolsó cseppeket is ki kell facsarnia a CPU és GPU teljesítményéből, a C++ tűnik a játékfejlesztés tökéletes eszközének.
- Hibrid alkalmazásokat is készíthetsz, amelyek C++ komponenseket használnak, de akár a C#, akár a Visual Basic nyelvet használhatják – természetesen a .NET keretrendszerrel és a XAML-lel a háttérben –, sőt, akár a JavaScriptet a HTML5/CSS3 felhasználói felülettel. Ebben az esetben a C++ komponensek azokat a feladatokat vállalhatják fel, amelyek komoly teljesítményt, illetve a rendszer erőforrásainak közvetlen elérését kívánják.

Először ismerkedj meg azzal, hogyan is működik együtt a Windows Runtime és a C++!

A Windows Runtime és a C++

Ebben a könyvben alig szerepel olyan fejezet, amelyben a Windows Runtime-ot ne említenénk, és ahol el tudnád kerülni ennek a komponensnek az objektumait és műveleteit. Amint azt korábban már megtanultad, a Windows Runtime komponenseit natív módon (C++-ban) valósították meg. Azt azonban még csak érintőlegesen tárgyaltuk, hogy ezek az objektumok a jó öreg COM (Component Object Model) technológia egy új változatát használják.

A COM már közel 20 éves múltra tekint vissza, egy bináris interfészszabvány, amelyet a Microsoft 1993-ban dolgozott ki. Eredeti célja az volt, hogy széles körben lehetővé tegye különböző programozási nyelvek számára saját dinamikus objektumaik, szolgáltatásaik létrehozását és megosztását – akár különböző nyelveken megírt processzekben is. Maga a COM megnevezés gyűjtőfogalommá vált, amely több kapcsolódó technológiát egyesített, mint például a DCOM, OLE, OLE Automation, COM+ és ActiveX.

Bár ez a bináris szabvány műszaki értelemben mindig nagyszerű volt, nagyon nehéz volt felhasználni, különösen C++-ból. Először is minden objektumot és azok interfészeit regisztrálni kellett – a Windows rendszer adatbázisában (Windows registry) –, és ez a telepítési eljárások visszatérő problémájává vált. Másodszor, a COM objektumok életciklusa referenciaszámláláson alapult. Ha egy objektum egy új változóhoz lett rendelve, a referenciaszámlálót kézzel kellett növelni. Ha az objektumot lefűzték erről a változóról, vagy a változót megszüntették, a számlálót csökkenteni kellett. Amikor a számláló elérte a nullát, a COM objektum automatikusan törlésre került, és az általa foglalt erőforrások felszabadultak. Elképzelheted, milyen problémákhoz vezetett, amikor ezt a számlálót valaki rosszul használta! Harmadszor, a kivételeket ez a modell a műveletek által visszaadott **HRESULT** értékekkel (32-bites egész számok) kezelte. A COM műveletek hívóinak mindig ellenőrizniük kellett, hogy vajon a művelet sikeres volt-e.

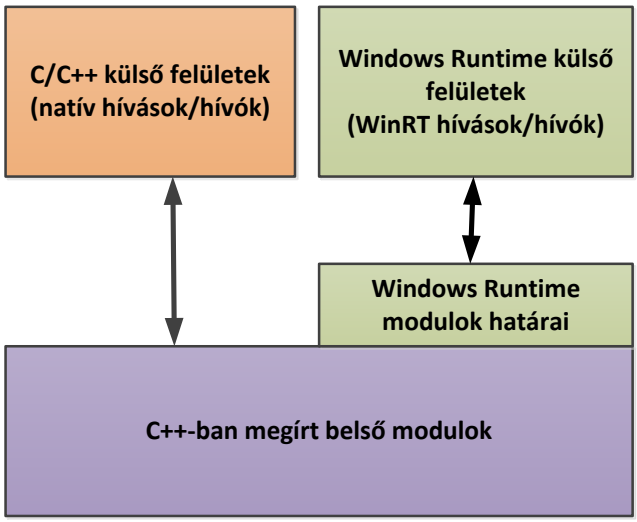
Ma a Windows Runtime egy új COM változatot használ, amely nem követeli meg az objektumok és interfészek előzetes regisztrációját, és átlátható módon végzi az objektumreferenciák számlálását – amint azt a következő fejezetrészben megismerheted.

A Windows Runtime objektumok kezelése a C++-ban

A Visual Studio 2012-ben a Microsoft új képességeket adott a C++ programozási nyelvekhez, amelyek a Windows Runtime objektumok kezelését ugyanazzal az egyszerűséggel biztosítják, mint ahogyan a .NET kezeli objektumait a C# és Visual Basic nyelvekben. Ezeknek az új nyelvi elemeknek a halmazát a Windows Runtime C++ komponens kiterjesztésnek (C++ Component Extensions, C++/CX) nevezzük.

A Windows Runtime objektumai erősen típusosak, és automatikus referenciaszámlálást valósítanak meg. Az **HRESULT** értékek helyett strukturált kivételkezeléssel bírnak. A Windows Runtime objektumok integráltak a Standard Template Library-vel (STL), és így minden STL-ben megvalósított függvény, konténer és algoritmus teljesen együttműködik a Windows Runtime típusaival. Létezik egy határvonal a Windows Runtime és az azt körülvevő világ között; a COM új változata jól definiált bináris interfész e között a két tartomány között. Felmerül néhány nyilvánvaló kérdés:

Mikor és hogyan kell egy C++ programozónak ezeket a nyelvi kiterjesztéseket használnia? Mindig ezeket a típusokat kell használni a Windows 8 stílusú alkalmazások készítésénél? Hogyan lehet a régi stílusú C++ típusokat és az STL-t használni? A válasz igazán egyszerű! A Windows Runtime típusokat akkor kell használnod, amikor ezt a határvonalat át szeretnéd lépni, azaz vagy Windows Runtime műveletet szeretnél végrehajtani, vagy más programozási nyelvek számára szeretnél szolgáltatásokat nyújtani. A követendő programozási stílust a 12-2 ábra mutatja be.



12-2 ábra: A C++ használata Windows Runtime objektumokkal

Szóval. pontosan úgy írhatsz C++ modulokat, mint eddig (feltéve, hogy C++ programozó vagy), és ezek a modulok közvetlenül használhatók más natív C vagy C++ modulokból. Ugyanezek a modulok fogyaszthatnak Windows Runtime objektumokat, vagy felkínálhatnak szolgáltatásokat más Windows Runtime objektumoknak vagy programozási nyelveknek egy vékony illesztőrétegen keresztül, amely oda-vissza adja át az adatokat Windows Runtime kompatibilis objektumok formájában.

Ennek az illesztésnek a megvalósítására a C++ kiterjesztése (C++/CX) kínálja a megoldást, amely lehetővé teszi, hogy a Windows Runtime objektumok más típusrendszerekkel is kompatibilisek legyenek, mint például a .NET vagy a JavaScript motor. A 12-3 táblázat a kiterjesztésekhez tartozó új típusokat foglalja össze.

12-3 táblázat: A C++ Component Extension adattípusai

Típus	Leírás
Érték típusok	Az érték típusok példányai mindig a hívási veremben vagy az alkalmazás statikus adatai között tárolódnak. Az érték struktúrák (a **`value struct`** kulcsszavakkal deklarálva) csak publikus adatmezőket tartalmaznak. Az érték osztályok (a **`value class`** kulcsszavakkal deklarálva) – ezek ritkábban használtak – csak publikus adatmezőket és műveleteket tartalmaznak.
Referencia típusok	A referencia típusok példányai mindig az alkalmazás dinamikusan kezelt memóriaterületén tárolódnak (ott kerülnek lefoglalásra és felszabadításra). Egy referencia osztály, amely a **`ref class`** kulcsszavakkal van deklarálva, publikus, védett és privát tagokat is, sőt, beágyazott típusokat is tartalmazhat. Egy referencia struktúrát a **`ref struct`** kulcsszavak vezetnek be. Ez ugyanazt a szemantikát követi, mit egy **`ref class`**, kivéve, hogy egy referencia osztály alapértelmezett elérési módja a publikus.
Interfész osztályok	Az interfész osztályok (ezeket az **`interface class`** kulcsszavak vezetik be) nagyon hasonlóak a natív C++ interfészekhez, de van néhány speciális jellemzőjük: • Minden tag implicit módon publikus. • Statikus mezők és egyéb statikus tagok nem használhatók. • A tagok lehetnek tulajdonságok, műveletek és események. • A műveletek paramétereiben és visszatérési értékeiben használt típusok csak Windows Runtime kompatibilis típusok lehetnek.
Generikus típusok	Már generikus interfész típusok is használhatók a C++-ban, ugyanazt a szemantikát követve, amelyet a .NET keretrendszer is használ. Ezeket

Típus	Leírás
	generic<typename T1, …, typename Tn> interface class konstrukció deklarálja. A generikus interfészek a C++ **template** definícióval is megvalósíthatók, de csak **typename** paramétereket tartalmazhatnak. Nem típus jellegű paraméterek nem használhatók.
Tulajdonságok	A szabványos C++ nem támogatja a tulajdonságokat. A C++/CX a **property** kulcsszavával már definiálhatsz tulajdonságokat – a .NET nyelvek **get/set** szemantikájának megfelelően.
Delegáltak	A delegáltak típusos függvénymutatók, és így deklarációjuk egy függvény definíciójára hasonlít, kivéve, hogy egy delegált az típus és nem függvény. A deklarációjukat a **delegate** kulcsszó vezeti be.
Események	Az események speciális típusok (delegáltak tulajdonságként használva), amelyek a .NET események **add/remove** szemantikájával működnek, és egy **raise** műveletet is tartalmaznak, amely leírja, hogyan kell az adott eseményt jelezni.

Amint már tudod, a Windows Runtime típusok COM típusok, és ennek megfelelően referenciaszámlálással kezelődnek. A Windows Runtime típusok speciális jelölést használnak annak érdekében, hogy a számlálók manuális kezelése elkerülhető legyen. A „^” („kalap”) karaktert használják annak jelzésére, hogy ők referenciaszámlálóval rendelkeznek, és allokációjuk során a **ref new** kulcsszót kell használni, amint azt az alábbi példa is mutatja:

```
Map<String^, int>^ MainPage::CreateMap()
{
    map<String^, int> myMap;
    myMap.insert(pair<String^, int>("One", 1));
    myMap.insert(pair<String^, int>("Two", 2));
    myMap.insert(pair<String^, int>("Three", 3));
    return ref new Map<String^, int>(move(myMap));
}
```

Itt a **CreateMap()** művelet olyan szótárat hoz létre, amelyben az elemek értékei **int**, a kulcsai **string** típusúak. A **Map** és a **String** egyaránt Windows Runtime típusok. A művelet egy referenciaszámlált mutatót ad vissza egy **Map** példányra, így az a **ref new** operátorral jön létre, és a visszatérési érték a „^” jelölést használja. A **Map** kulcsai szintén referenciával számlált mutatókat használnak, és így **String^** típusként vannak deklarálva.

*A „^” jelölés és a **ref new** kulcsszavak a fordítóprogram egyszerű „trükkjei”. Ezek nélkül a programozónak a COM referenciaszámlálókat manuálisan kellene növelnie. C++/CX kiegészítésnek köszönhetően ezeket a feladatokat a fordítóprogram teljes egészében átveszi, és a megfelelő kódot generálja.*

A Windows Runtime objektumokkal a „” (mutató) jelölés nem használható. Ha így teszel, a fordítóprogram hibaüzenetet ad.*

Futásidejű osztályok létrehozása

Egy C++-ban írt Windows 8 stílusú alkalmazás létrehozásakor a Windows Runtime objektumok fogyasztása mellett gyakran ilyen objektumokat létre is hozol. Például egy alkalmazást és annak a képernyőn megjelenő oldalait is Windows Runtime osztályok reprezentálják. A megvalósítás szemszögéből ezeknek referencia osztályoknak kell lenniük néhány megszorítással, amint azt a következő mintapélda is mutatja:

```
public ref class Customer sealed
{
public:
    Customer(String^ firstName, String^ LastName);
    Customer(String^ lastName);
    property String^ FullName
    {
        String^ get();
    }
    void MarkAsKeyAccount();
    ~Customer();
private:
      std::wstring m_firstName;
      std::wstring m_LastName;
};
```

A futásidejű osztályokat a **public** módosítóval kell megjelölni, hogy elérhetők legyenek külső alkalmazásokból és más programozási nyelvekből. Bár a műveletek paraméterei és visszatérési értékei tetszőleges típusúak lehetnek, a publikus tagok csak Windows Runtime típusokat használhatnak. Szintén használni kell a **sealed** kulcsszót, hogy megakadályozd más típusok leszármaztatását.

> *A **public** kulcsszó egyúttal jelzi a fordítóprogramnak, hogy a publikus tagok metaadatait a fordításkor a külvilág számára létrehozott **.winmd** fájlba is bele kell tenni.*
>
> *A **sealed** módosítót elhagyhatod, ebben az esetben a fordítóprogram olyan figyelmeztetést ad ki, amelyben jelzi, hogy az adott típust nem fogod tudni JavaScriptből használni, mivel az nincs **sealed** típusként megjelölve.*

A saját futásidejű osztályaidat a **ref new** operátorral példányosíthatod, éppen úgy, mint akármelyik Windows Runtime osztályt:

```
Customer^ myCustomer = ref new Customer("John", "Doe");
```

Nemcsak a **ref new** példányosítás, de az osztály egyszerű lokális változóként való deklarálása is biztosítja az életciklus automatikus kezelését:

```
{
    Customer myOtherCustomer("Jane", "Doe");
    myOtherCustomer.MarkAsKeyAccount();
    // ...
    // Az adatok feldolgozása, lekérdezése, módosítása
    // ...
}   // ~Customer() automatikusan meghívódik
```

Ebben a kódrészletben a **myOtherCustomer** lokális változó. A fordítóprogram kezeli annak életciklusát, és automatikusan felszabadítja, amikor a változó elhagyja a hatókörét. Ebben az esetben a **~Customer()** destruktor akkor hívódik, amikor az alkalmazás elhagyja a kódrészletben szereplő utasításblokkot.

Kivételek

A COM kivételek kezelése elég fáradságos munka volt, mert a COM műveletek **HRESULT** hibakódokat adtak vissza. Például, ha rossz paramétert adtál át egy műveletnek, egy E_**INVALIDARG** értékű **HRESULT** kódot adott az vissza. Ha ezt a visszatérési értéket nem ellenőrizted le, és a programot engedted továbbfutni, annak előbb vagy utóbb az alkalmazás összeomlása lehetett a vége. A Windows Runtime még mindig COM-ot használ, de most már lehetővé teszi a strukturált kivételkezelést. Ahelyett, hogy közvetlenül **HRESULT** kódokkal kellene foglalkozni, elkaphatod azokat a kivételeket, amelyeket az **HRESULT** kódok reprezentálnak. A 12-4 táblázat ezeket a COM-specifikus kivételtípusokat mutatja be azzal az HRESULT kóddal együtt, amit reprezentálnak.

12-4 táblázat: A Windows Runtime kivételtípusai a hozzájuk tartozó HRESULT kódokkal

Kivételtípus	HRESULT
InvalidArgumentException	**E_INVALIDARG**
NotImplementedException	**E_NOTIMPL**
AccessDeniedException	**E_ACCESSDENIED**
NullReferenceException	**E_POINTER**
InvalidCastException	**E_NOINTERFACE**
FailureException	**E_FAIL**
OutOfBoundsException	**E_BOUNDS**
ChangedStateException	**E_CHANGED_STATE**
ClassNotRegisteredException	**REGBD_E_CLASSNOTREG**
DisconnectedException	**E_DISCONNECTED**
OperationCanceledException	**E_ABORT**
OutOfMemoryException	**E_OUTOFMEMORY**

*Az összes kivételtípus a **Platform** névtérben található.*

Nem kell azzal törődnöd, hogyan lesznek az HRESULT kódokból kivételek, ezt a futásidejű környezet elintézi. Ha a Windows Runtime objektumok műveleteivel kapcsolatos potenciális problémákat szeretnéd kezelni, az alábbi mintát kövesd:

```
void MyWIndowsRTOperation(String^ parameter1, int parameter2)
{
    OtherRuntimeObject^ otherObj = ref new OtherObj();
    try
    {
        otherObj->SimpleOperation(parameter1);
        // ...
        otherObj->AnotherOperation(parameter2);
        // ...
        otherObj->CompountOperation(parameter1, parameter2)
        // ...
    }
    catch (NotImpelentedException^ ex)
    {
        // Reagálj arra a helyzetre, amikor a művelet nincs definiálva!
    }
    catch (InvalidArgumentException^ ex)
    {
        // Naplózd le a problémát!
       throw ex;
    }
    catch (COMException^ ex)
    {
        // Vizsgáld meg a HResult tulajdonságot, és döntsd el, mit is kell csinálni!
    }
    catch (...)
    {
        // Döntsd el, mi legyen az összes egyéb kivétellel!
    }
}
```

Az első két **catch** blokk konkrét COM kivételeket kezel. A harmadik **catch** blokk egy **COMException** példány mutatóját fogadja. Mivel a 12-4 táblázatban lévő minden kivétel a **COMException** típusból származik le, ez a blokk az összes ilyet kezeli, kivéve a **NotImplementedException** és az

InvalidArgumentException típusokat, mert azok feldolgozását már a korábbi **catch** ágak elvégzik. Ebben az ágban megvizsgálhatod a kivételpéldány **HResult** tulajdonságának értékét, és az alapján dönthetsz a továbbiakról. A negyedik **catch** blokk minden más kivételt elkap. Amint azt a fenti mintából felismerheted, az mindig a legspecifikusabb kivételt kapja el először és a legkevésbé specifikust legutoljára.

Abból a tényből kiindulva, hogy minden kivétel a **COMException** típusból származik, könnyen abba a kísértésbe eshetsz, hogy saját COM-specifikus kivételt hozz létre a saját hibakódoddal egy új típus örökítésével a **COMException**ből. Hát, ez belső implementációs részleteknek köszönhetően bizony nem fog működni! Ebben az esetben egy olyan **COMException** kivételt kell emelned, amelynek a konstruktorában adod át a kérdéses hiba kódját, amint azt a következő kódrészlet is mutatja:

```
void MyOperation(int param)
{
    if (param == 0)
    {
        throw ref new COMException(MY_ERROR_CODE);
    }
    // ...
}
// ...

try
{
    // ...
    MyOperation(0);
    // ...
}
catch (COMException^ ex)
{
    if (ex->HResult == MY_ERROR_CODE)
    {
        // Handle your own error code
    }
    // ...
}
```

A **COMException** példányt csak a **catch (ComException^ ex)** ágban lehet elkapni. Még ha egy ismert hibakódhoz tartozó kivételpéldányt is emelsz a **COMException** példányon keresztül, amely szerepel a 12-4 táblázatban, nem tudod azt a táblázatban szereplő kivételtípusként elkapni, csak **COMException**ként:

```
try
{
    // ...
    throw ref new COMException(E_FAIL)
    // ...
}
catch (FailureException^ ex)
{
    // The COMException above cannot be caught here
}
catch (COMException^ ex)
{
    if (ex->HResult == E_FAIL)
    {
        // This is where to handle the COMException above
    }
}
```

Már elegendő információval rendelkezel a C++-ról a Windows 8 stílusú alkalmazások kapcsán. Itt az ideje, hogy kipróbáld ezeket az új nyelvi képességeket a Visual Studióval!

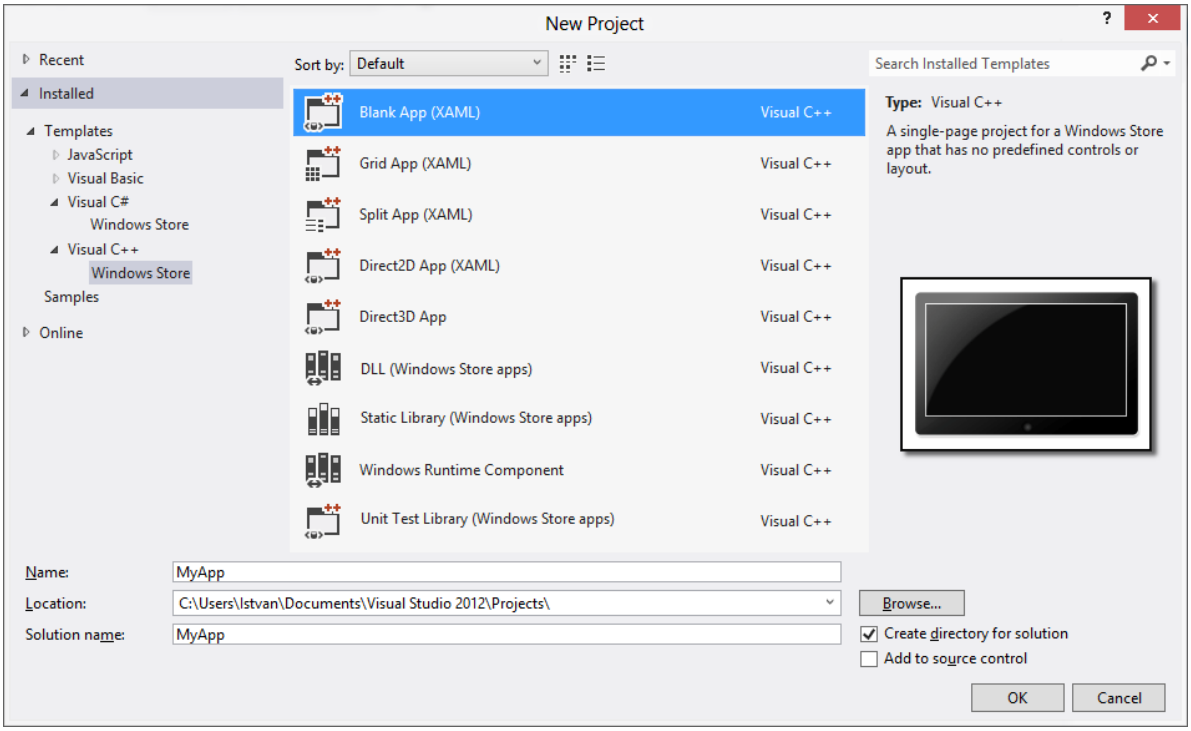
A C++ képességeinek a felfedezése a Visual Studióval

A Visual Studio kiemelkedő támogatást nyújt a C++ alkalmazások készítéséhez. Minden fontos produktivitást javító eszközt megtalálhatsz a környezetében, éppen úgy, mint a többi programozási nyelvhez. Ebben a fejezetrészben megtanulsz néhány dolgot a C++ alkalmazások létrehozásáról és szerkesztéséről, mialatt néhány új C++ képességgel is megismerkedsz.

Az előző fejezetekben már megtanultad a Windows 8 stílusú felhasználói felületek létrehozásával kapcsolatos összes fontos dolgot. Ebben a részben egy előre elkészített mintapéldából kiindulva néhány gyakorlaton keresztül tekinted át az újdonságokat. Mielőtt elkezdenél a mintapéldával játszani, nézd meg, hogyan is kell egy C++ projektet létrehozni!

C++ projektek létrehozása

A Visual Studióban C++ programokat a File | New Project paranccsal (Ctrl+Shift+N) tudsz létrehozni, amint korábban azt már megtanultad. Természetesen a Visual C++ kategória alatt lévő sablonokból kell választanod, ezeket a 12-3 ábra mutatja be. Az ábra középső részén láthatod azokat a C++ Windows 8 stílusú sablonokat, amelyeket a Visual Studio 2012 Express for Windows 8 támogat.



12-3 ábra: A C++ programozási nyelv által támogatott Windows 8 stílusú sablonok

A Visual Studio 2012 egyéb (nem ingyenes) változatai további C++ projektsablonokat is biztosítanak asztali alkalmazások és komponensek fejlesztéséhez.

A C++ ugyanazokat a szabványos Windows 8 stílusú alkalmazássablonokat támogatja, mint a többi programozási nyelv (Blank Application, Grid Application és Split Application), és még néhány egyebet is:

- **Blank Dynamic Link Library**: Olyan könyvtár létrehozására használd ezt a sablont, amelyet `.dll` fájllá kell lefordítani! Bár ez a könyvtár ugyanazt a kiterjesztést használja, mint a .NET komponenskönyvtárak szerelvényei, a fordítás eredménye natív kód lesz, így azokra nem tudsz C# vagy Visual Basic projektekből hivatkozni.
- **Blank Static Library**: Ezzel a sablonnal `.lib` kiterjesztésű könyvtárakat tudsz fordítani. Ezeket a fordítóprogram statikusan hozzáfűzi más projektjeid végrehajtható állományaihoz.

- **Unit Test Library**: Amint a sablon neve is sugallja, ezzel a típussal egységteszt projekteket tudsz készíteni C++ alkalmazásaidhoz és könyvtáraidhoz.
- **Windows Runtime Component**: Ez a projekttípus natív Windows Runtime komponens létrehozását teszi lehetővé, amelyet bármelyik más nyelvhez kapcsolódó projektben fel tudsz használni. A komponenskönyvtár mellett ez a projekttípus egy **`.winmd`** fájlt is létrehoz, amely a publikus típusokhoz kapcsolódó metaadat információt tárolja.
- **Direct2D Application** és **Direct3D Application**: ezek a sablonok értelemszerűen a Direct2D és Direct3D technológiákkal működő alkalmazások készítésére használhatók.

A projektjeid jellemzőinek megadására a New Project dialógus ugyanazokat az opciókat biztosítja – például az alkalmazás neve, helye a fájlrendszerben, létrehozási mód stb. –, mint a többi programozási nyelv.

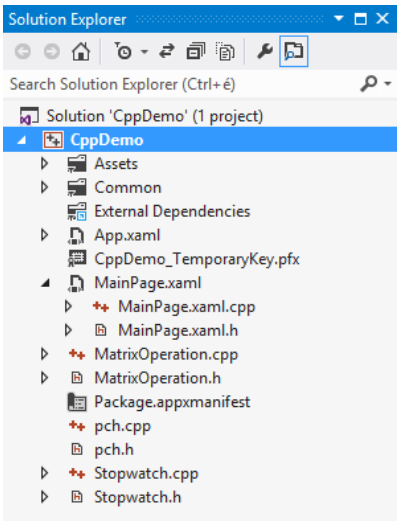
Egy C++ projekt elemei

A következő gyakorlatban a **`CppDemo - Start`** mappában található *solution* fájlt használhatod egy projekt elemeinek megtekintésére. A C++ projektek szerkezete nagyon hasonló azokhoz, amelyeket más programozási nyelvekkel hozhatsz létre, bár van néhány jellegzetes különbség, amelyeket itt megismerhetsz.

***Gyakorlat**: Egy C++ projekt elemeinek megvizsgálása*

Kövesd az alábbi lépéseket egy C++ projekt szerkezetének megismeréséhez:

1. A File | Open Project paranccsal (Ctrl+Shift+O) nyisd meg a **`CppDemo - Start`** mappában található **`CppDemo.sln`** fájlt! Néhány másodperc alatt az betöltődik a Visual Studióba, és a projekt szerkezete megjelenik a Solution Explorer ablakban, amint az a 12-4 ábrán is látható.

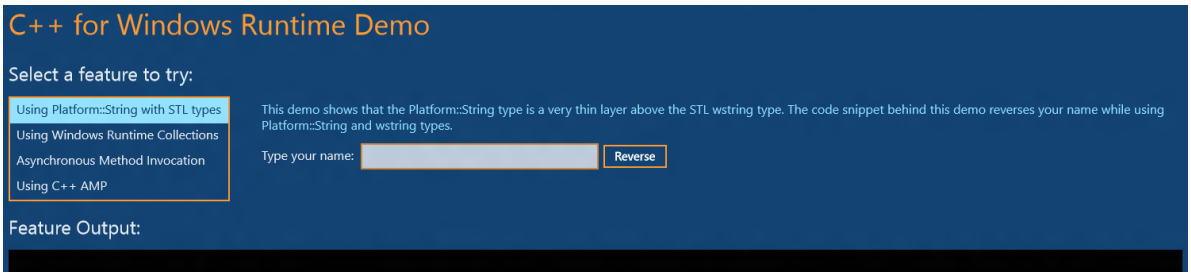


12-4 ábra: A CppDemo projekt a Solution Explorerben

2. A Solution 'CppDemo' alatt találhatod a **`CppDemo`** projektet. Kattints az **`Assets`** mappától balra elhelyezkedő kicsi háromszögre, hogy megvizsgálhasd a mappa tartalmát! Négy képfájlt láthatsz ott, amelyek az alkalmazás logóit és indítóképernyőjét tartalmazzák különböző méretekben.
3. Nyisd ki a **`Common`** mappát! Az különböző C++ és header fájlokat tartalmaz, amelyek mind a Windows 8 stílusú alkalmazás vázának részei. A **`StandardStyles.xaml`** fájl tartalmazza az alkalmazásban használt stílusok és egyéb erőforrások definícióit.
4. Nyisd ki az **`External Dependencies`** mappát! Ez több száz bejegyzést tartalmaz, amelyek a közvetlenül vagy közvetve használt header fájlokat és egyéb függőségeket, mint például a projektben hivatkozott **`.winmd`** állományokat mutatja. Csukd be a mappát!
5. Nyisd le a **`MainPage.xaml`** fájlt! Akárcsak a többi programozási nyelv esetében, ez a XAML fájl is beágyaz egy kódfájlt, a **`MainPage.xaml.cpp`**-t, ahol a **`.cpp`** fájlkiterjesztés már sugallja, hogy egy

C++ fájlról van szó. Találsz itt még egy beágyazott fájlt, a **MainPage.xaml.h** header fájlt, amely a **MainPage.xaml.cpp** fájlban szereplő típusok definícióját tartalmazza.

6. Dupla kattintással nyisd meg a **MainPage.xaml.cpp** fájlt! A kódszerkesztőben lapozz a fájl aljáig, ott néhány üres törzsű metódust találsz „Feature” megjegyzésekkel. Ezek azok a műveletek, amelyekbe a következő gyakorlatok során a kódrészleteket be fogod írni.
7. Nyisd meg a **MainPage.xaml.h** fájlt! A kódszerkesztőben kattints az **#include "MainPage.g.h"** sorra, majd nyomd meg a Ctrl+Shift+G billentyűt! A **MainPage.g.h** fájl nyílik meg, és megmutatja a **MainPage.xaml** fájlban leírt objektumok C++ nyelvű definíciót.
8. A Debug | Start Without Debugging paranccsal (Ctrl+F5) indítsd el az alkalmazást! A fordítóprogram pár másodperc alatt felépíti a program futtatható állományát, majd elindítja azt. Az alkalmazás a 12-5 ábrán látható képernyővel indul el.



12-5 ábra: A CppDemo alkalmazás az indulása után

9. Kiválaszthatsz egy elemet a képernyő bal oldalán megjelenő listából, hogy megadhasd az adott képességet demonstráló programrész adatait a jobb oldalon található mezőkben. Ez az alkalmazás egy kezdeti változata, és még egyetlen funkció sem működik benne.
10. Az Alt+F4-gyel zárd le az alkalmazást!

Hogyan működik?

A **CppDemo** projekt struktúrájának vizsgálata közben megállapíthattad, hogy annak felépítése nagyon hasonló a többi programozási nyelven létrehozott projektekéhez, bár a mappáknak itt más neve van.

A C++-nak is szoros az integrációja a XAML-lel. Amint azt az 5. lépésben felfedezhetted, a **MainPage.xaml** fájl is tartalmaz beágyazott kódfájlokat. Amíg a **MainPagex.xaml.cpp** fájlban te magad írhatod le a működéshez szükséges kódot, addig a **MainPage.g.h** fájl minden alkalommal újragenerálódik, amikor csak módosítod és elmented a **MainPage.xaml** fájlt. A kulisszák mögött a fejlesztőkörnyezet egy **MainPage.xaml.hpp** fájlt is generál, amely a komponenseidet inicializálja a **MainPage** objektum létrehozásakor.

Amikor a **CppDemo** alkalmazást a 9. lépésben a fordítás után elindítottad, a Visual Studio egy telepítő csomagot is létrehozott – a **Package.appxmanifest** fájl használatával –, és telepítette is az alkalmazást csakúgy, mint azt már a korábbi fejezetek gyakorlataiban is megismerhetted.

Most már készen állsz arra, hogy ezt az egyszerű C++ alkalmazásod módosítsd a következő gyakorlatok során, és megismerkedj egy-egy új dologgal.

A Platform::String típus használata

Azért, hogy a Windows Runtime komponenseknek karakterláncokat át tudj adni, illetve szöveges adatokat tudj visszakapni egy Windows Runtime műveletből, a Microsoft létrehozott egy új **String** típust, amely a **Platform** névtérben található. Ez a típus egy vékony burkoló, amely néhány egyszerű művelettel rendelkező szöveges objektumok használatát teszi lehetővé. Az a nagyszerű ebben a típusban, hogy teljes egészében együttműködik az STL **wstring** típusával – amint azt a következő gyakorlatban is láthatod.

*Ha nem vagy még gyakorlott C++ programozó, tudnod kell, hogy a C++ a „**::**” operátort használja a névterek és a típusok elválasztására. Így a Windows Runtime **String** típusának a teljes neve **Platform::String**, mivel az a **Platform** névtérben van definiálva.*

Gyakorlat: A `Platform::String` típus használata

Kövesd az alábbi lépéseket, hogy megismerkedhess a **Platform::String** típus használatával:

1. Nyisd meg a **CppDemo – Start** mappában található **CppDemo.sln** fájlt, ha nem maradt nyitva az előző gyakorlat után a Visual Studióban!
2. A Solution Explorerben dupla kattintással nyisd meg a **MainPage.xaml.cpp** fájlt, és keresd meg a kódban az üres törzsű **ReverseStringButton_Tapped** műveletet!
3. Illeszd be az alább megjelölt kódot a művelet törzsébe!

```
void MainPage::ReverseStringButton_Tapped(Object^ sender, TappedRoutedEventArgs^ e)
{
    String^ yourName = YourName->Text;
    Feature1OutputText->Text += "Your name: " + yourName + "\n";
    wstring inputString(yourName->Data());
    wstring reverseString(inputString.rbegin(), inputString.rend());
    String^ yourReversedName = ref new String(reverseString.c_str());
    Feature1OutputText->Text += "Reversed: " + yourReversedName + "\n";
}
```

Ebben a kódban a **yourName** egy referenciaszámlálással kezelt mutató (amint arra a „**^**” jelölés is utal), és az a **YourName** szöveges mező tartalmát veszi át. Ezt a változót felhasználhatod arra, hogy szöveges konstansokat illessz össze a „**+**” operátorral – ezt a **String** típus deklarálja –, és beállítsd a **Feature1OutputText** blokk Text tulajdonságát.

4. Futtasd az alkalmazást! A menülistából válaszd ki az első elemet! Írd be a neved (vagy bármilyen más szöveget) a képernyőn látható szövegmezőbe, és kattints a Reverse gombra! A program kimenetén megjelenik a beírt szöveg fordítva (annak betűi jobbról balra).
5. Zárd be az alkalmazást!

Hogy működik?

A **String** típus közvetlenül használható más Windows Runtime típusokkal, beleértve a XAML vezérlőket. Ezen a módon jeleníti meg a fenti kódrészlet a kimenetét a **Feature1OutputText** változó **Text** tulajdonságának beállításával. Ez a **String** típus azonban csak nagyon kevés műveletet valósít meg! Ha ezeknél összetettebbekre van szükséged – ez pedig gyakran előfordul –, akkor az STL **wstring** típusát használhatod együtt a **String**gel. A kezdeti tartalom beállításához a fenti kódban az **inputString** változó a **String::Data()** műveletével szerez egy mutatót a C++ stílusú szövegre.

A következő sorban a **reverseString** változó azt a **wstring** konstruktort használja, amely karakterek egy tartományát használja saját tartalma létrehozásához. Az **rbegin()** és **rend()** műveleteknek köszönhetően ez az iterátor a szöveg utolsó karakterétől az első felé halad, vagyis az eredeti **string** fordítottját hozza létre. A megfordított szöveg kiírásához egy **String^** példányra van szükség. A **yourReversedName** változó egy új **String**-et példányosít (a **ref new** operátorral) és a **c_str()** műveletet használja a **reverseString** változó tartalmának megszerzéséhez.

Az itt bemutatott együttműködési minta igen gyakori a Windows Runtime és az STL típusok között. A Windows Runtime konténereit is hasonló módon használhatod együtt az STL konténereivel, amint azt a következő gyakorlatban megismered.

Futásidejű konténerek használata

A C++ STL több konténert is definiál, amelyeket a hozzájuk kapcsolódó műveletek tárháza egészít ki. Sőt, amint azt a 12-1 táblázatban is láthattad, a C++11 új konténertípusokat is ad az STL-hez. Programjaidban bárhol használhatsz STL konténereket, azonban azokat nem tudod közvetlenül átadni Windows Runtime objektumoknak. A megoldást a **`Platform::Collections`** névtérben található Windows Runtime konténerek használata jelenti. A következő gyakorlatban megtanulod a **`Vector`** és **`VectorView`** konténerek használatát.

Gyakorlat: A `Vector` és `VectorView` konténerek használata

Kövesd az itt leírt lépéseket, hogy megismerkedhess ezeknek a konténereknek a használatával:

1. Nyisd meg a **`CppDemo - Start`** mappában található **`CppDemo.sln`** fájlt, hacsak még mindig nincs nyitva a Visual Studióban!
2. A Solution Explorerben dupla kattintással nyisd meg a **`MainPage.xaml.cpp`** fájlt, és keresd meg a **`CreateRandomVector()`** műveletet, amelynek törzse jelenleg üres!
3. Illeszd be a következő kódrészletet a művelet törzsébe:

```
col::Vector<int>^ MainPage::CreateRandomVector()
{
    vector<int> myNumbers;
    srand((unsigned)time(0));
    for (int i = 0; i < 10; i++)
    {
        myNumbers.push_back(rand()%100 + 1);
    }
    return ref new col::Vector<int>(move(myNumbers));
}
```

A művelet egy **`for`**-ciklust használ a **`vector<int>`** típusú konténer – a **`vector`** az STL-ben van definiálva – véletlen számokkal való feltöltésére. Az utolsó sorban lévő **`return`** utasítás egy új **`Vector<int>`** konténert definiál, amely a **`move`** művelettel veszi át a véletlen számokat tartalmazó vektor tulajdonlását.

4. Keresd meg a **`SumAVectorButton_Tapped`** műveletet, és másold be a következő kód kiemelt részét a metódus törzsébe:

```
void MainPage::SumAVectorButton_Tapped(Object^ sender, TappedRoutedEventArgs^ e)
{
    wstringstream streamVal;
    WFC::IVectorView<int>^ vv = CreateRandomVector()->GetView();
    int sum = 0;
    int index = 0;
    for_each(begin(vv), end(vv), [&](int value)
    {
        streamVal << "Number " << ++index << " is " << value << endl;
        sum += value;
    });
    streamVal << "Sum of these values is " << sum;
    Feature2OutputText->Text = ref new String(streamVal.str().c_str());
}
```

Ez a művelet meghívja a **`CreateRandomVector()`** metódust, és a visszatérési érték **`GetView()`** műveletével egy pillanatképet készít a véletlen számokat tartalmazó vektorról. A ciklus az elemeket **`streamVal`** kimenetre írja, és összeadja azokat. Miután az eredményt kiszámolta, a kimenetet a **`FeatureOutput2Text`** vezérlőre írja.

5. A Ctrl+F5-tel futtasd az alkalmazást! A menülistából válaszd a „Using Windows Runtime Collections” parancsot, majd kattints a Sum Up a Vector nyomógombra, vagy érintsd meg azt!

6. A program a kimenetén megjeleníti a véletlen számokból álló vektort és az elemeinek az összegét. Zárd be az alkalmazást!

Hogyan működik?

A **CreateRandomVector()** művelet az STL **vector** osztályát használja az értékek előállítására, és ezt egy **Vector** objektumba csomagolja az értékek visszaadásánál. A **SumAVectorButton_Tapped** művelet ennek a vektornak az értékeit csak kiolvasni szeretné azok megváltoztatása nélkül, ezért egy **VectorView** objektumot használ (ez az **IVectorView** interfészt valósítja meg). A WFC azonosító rövidített név (alias) a **Windows::Foundation::Collection** névtérhez, és a **MainPage.xaml.cpp** fájl kezdetén van definiálva.

Az utolsó kódsorban a **String** osztály használatára láthatsz jó példát. A programrészlet minden kimeneti eleme a **streamVal** szöveges kimenetre íródott (ez egy **stream** objektum), és az utolsó programsor azt egy String példányra konvertálja, hogy a program kimenetén megjeleníthető legyen.

A Windows Runtime meglehetősen kevés konténert definiál, ezek mindegyike együttműködik az STL konténereivel. Ezeknek a Windows Runtime típusoknak a listáját az alábbi MSDN oldalon találod meg: http://msdn.microsoft.com/en-us/library/windows/apps/hh710418(v=vs.110).aspx

Aszinkron műveletek használata

A könyv korábbi fejezeteiben már számtalan példát láthattál az aszinkron műveletek használatára. A C#-ban az új **async** és **await** kulcsszavak a fordítóprogram segítségével varázslatos dolgokat végeznek, lehetővé teszik, hogy az aszinkron metódusokat ugyanazzal az egyszerűséggel használhasd, mint a szinkron metódusokat – miközben megőrzik a felhasználói felület válaszképességét. A JavaScriptben a Common JavaScript Promises mintát használhatod.

A C++ fordítóprogramja nem tartalmaz olyan trükköket, mint a C# vagy Visual Basic fordítóé, de van egy nagyszerű komponens, a Parallel Pattern Library (PPL), amely az aszinkron műveletek hívását viszonylag egyszerű módon engedi kezelni, amint azt a következő gyakorlat során megtapasztalhatod.

***Gyakorlat**: Egy fájl aszinkron írása*

Kövesd az alábbi lépéseket egy rövid szöveges üzenet aszinkron módon való fájlba írásához:

1. Nyisd meg a **CppDemo – Start** mappában található **CppDemo.sln** fájlt, hacsak még mindig nincs nyitva a Visual Studióban!
2. A Solution Explorerben dupla kattintással nyisd meg a **MainPage.xaml.cpp** fájlt, és keresd meg a **WriteFileButton_Tapped** műveletet, amelynek törzse jelenleg üres!
3. Illeszd be a kijelölt kódrészletet a művelet törzsébe:

```
void MainPage::WriteFileButton_Tapped(Object^ sender, TappedRoutedEventArgs^ e)
{
    Feature3OutputText->Text += "File creation has been started.\n";
    StorageFolder^ documentsFolder = KnownFolders::DocumentsLibrary;

    // --- Start file creation
    task<StorageFile^> createTask(documentsFolder->CreateFileAsync(
        "MySample.txt", CreationCollisionOption::ReplaceExisting));
    createTask.then([this](StorageFile^ file)
    {
        Feature3OutputText->Text += "File " + file->Name + " has been created.\n";
        // --- Start writing into the file
        task<void> writeTask(FileIO::WriteTextAsync(file, YourMessage->Text));
```

```
        writeTask.then([this](void)
        {
            Feature3OutputText->Text += "Text '" + YourMessage->Text +
                "' has been written to the file.\n";
        });
    });
}
```

Ez a kódrészlet két aszinkron műveletet láncol fel, a **createTask**-ot, amely egy **MySample.txt** nevű fájlt hoz létre, illetve a **writeTask**-ot, amely a felhasználó által megadott üzenetet írja a fájlba. Minden művelet a **then** metódust használja egy lambda kifejezéssel az adott taszk folytatásának leírásához.

4. A Ctrl+F5-tel futtasd az alkalmazást! A menülistából válaszd ki az „Asynchronous Method Invocation” parancsot! A szövegmezőbe gépelj be egy üzenetet, majd nyomd meg (érintsd meg) a Write Message to File nyomógombot!
5. A program kimenetén megjelenít néhány sort, amelyek jelzik a fájl létrehozásának és az üzenet kiírásának tényét.
6. Indítsd el a Windows Explorert, és válaszd ki a **Documents** mappát a **Libraries** alatt! Keresd meg a **MySample.txt** fájlt, és dupla kattintással nyisd meg! Leellenőrizheted, hogy az általad megadott szöveg tényleg kiíródott-e!
7. Kapcsolj vissza a futó **CppDemo** alkalmazásra, és zárd le!

Hogyan működik?

A gyakorlatban szereplő kód a **task<TResult>** sablont használja az aszinkron művelet létrehozásához, ahol a **TResult** a művelet által eredményként visszaadott érték típusa. A **task** argumentuma egy olyan objektum, amely megvalósítja az **IAsyncOperation<TResult>** interfészt.

A **createTask** a **CreateFileAsync** műveletet használja, amely egy **StorageFile^** értéket ad vissza. A **writeTask** a **WriteTextAsync** műveletet hívja meg, amely nem ad vissza értéket. Ezek a műveletek fel vannak láncolva: a **createTask** művelet folytatása (a **then** metódus) létrehozza a **writeTask**-ot a **createTask** eredményét felhasználva argumentumként.

Felismerheted, hogy a műveletek folytatásához tartozó lambda kifejezések megfogják a **this** mutatót, és így hozzáférnek a felhasználói felület elemeihez. Ezen a módon írnak a műveletek üzenetet a program képernyőjére.

> *Több aszinkron működési minta is van, amelyeket a C++-ban használhatsz. Látogasd meg az alábbi lapot további információkért: http://msdn.microsoft.com/en-us/library/windows/apps/hh780559.aspx!*

Az Accelerated Massive Parallelism használata

A legtöbb computer nemcsak CPU-t, hanem nagy teljesítményű grafikus processzort (GPU) és más processzorokat, például audió feldolgozót (APU) is tartalmaz. Rengeteg olyan művelet van, amelyeket sokkal gyorsabban lehet ezeken az eszközökön végrehajtani, mert az architektúrájuk miatt speciális műveletek esetében nagyfokú párhuzamosítást tesznek lehetővé. A Microsoft C++ AMP (Accelerated Massive Parallelism) egy nyílt specifikáció, amely közvetlenül a C++-ban valósítja meg adatfeldolgozások párhuzamosítását.

A Direct2D és Direct3D technológiák a GPU-t használják bonyolult grafika megjelenítésére. A GPU-knak olyan architektúrájuk van, amely a pixeleket és vektorokat nagyon gyorsan tudja feldolgozni, ráadásul nagyon sok műveletet tud párhuzamosan végrehajtani. Ez a teljesítmény nemcsak grafika létrehozására, hanem adatok feldolgozására is használható. A C++ AMP technológia saját könyvtárával terjeszti ki a C++-t, és a fordítóprogram olyan kódot generál, amely közvetlenül a GPU-n (és az egyéb gyorsítást támogató processzorokon) hajtható végre.

A könyvnek távolról sem célja a C++ AMP technológia részletes bemutatása. Ha további részletek érdekelnek, az MSDN-en sokkal több információt találsz. Indulj erről a lapról: http://msdn.microsoft.com/en-us/library/hh265137(v=vs.110)!

Amint azt a következő gyakorlatban is láthatod, a Windows 8 stílusú alkalmazásokban közvetlenül felhasználhatod a C++ AMP technológiát! A gyakorlatban 1000 sorból és 1000 oszlopból álló lebegőpontos számokat tartalmazó mátrixokat fogsz összeszorozni. Ez a mátrixok méretéből fakadóan komplex művelet. A szorzat mind az egymillió cellaértékének kiszámításához egyenként 1000 lebegőpontos szorzást, majd ezek eredményének összeadását kell elvégezned. Ez összességében kétmilliárd lebegőpontos műveletet jelent!

A következő gyakorlatban ezt a műveletet megvalósítod a hagyományos soros feldolgozási stílusban és a C++ AMP alkalmazásával egyaránt.

A mátrixok szorzásával kapcsolatos ismereteidet az alábbi lapon frissítheted fel: http://en.wikipedia.org/wiki/Matrix_multiplication

***Gyakorlat**: A C++ AMP használata Windows 8 alkalmazásokban*

A soros és a C++ AMP mátrixszorzás algoritmusának összehasonlításához kövesd az alábbi lépéseket:

1. Nyisd meg a **`CppDemo – Start`** mappában található **`CppDemo.sln`** fájlt, hacsak még mindig nincs nyitva a Visual Studióban!
2. A Solution Explorerben dupla kattintással nyisd meg a **`MainPage.xaml.cpp`** fájlt, és keresd meg a **`CalculateSerialButton_Tapped`** műveletet, és illeszd be a kijelölt kódrészletet a művelet törzsébe:

```
void MainPage::CalculateSerialButton_Tapped(Object^ sender,
    TappedRoutedEventArgs^ e)
{
    Feature4OutputText->Text += "Sequential matrix multiplication started...\n";
    CalculateSerialButton->IsEnabled = false;
    MatrixOperation matrixOp;
    task<unsigned long long> matrixTask (matrixOp.MultiplyMatrixSeriallyAsync());
    matrixTask.then([this](unsigned long long elapsed)
    {
        CalculateSerialButton->IsEnabled = true;
        wstringstream streamVal;
        streamVal << "Sequential operation executed in " << elapsed << "ms.\n";
        Feature4OutputText->Text += ref new String(streamVal.str().c_str());
    });
}
```

3. Illeszd be a következő kódot a **`CalculateWithAMPButton_Tapped`** műveletbe:

```
void MainPage::CalculateWithAMPButton_Tapped(Object^ sender, TappedRoutedEventArgs^ e)
{
    Feature4OutputText->Text += "AMP matrix multiplication started...\n";
    CalculateWithAmpButton->IsEnabled = false;
    MatrixOperation matrixOp;
    task<unsigned long long> matrixTask (matrixOp.MultiplyMatrixWithAMPAsync());
    matrixTask.then([this](unsigned long long elapsed)
    {
        CalculateWithAmpButton->IsEnabled = true;
        wstringstream streamVal;
        streamVal << "AMP operation executed in " << elapsed << "ms.\n";
        Feature4OutputText->Text += ref new String(streamVal.str().c_str());
    });
}
```

4. A Solution Explorerben duplán kattintva a **`MatrixOperation.cpp`** fájlra nyisd meg azt! Keresd meg a **`MultiplySequential`** műveletet, és írd be annak törzsébe az itt kijelölt kódot, amely a mátrixszorzást a hagyományos módon végzi el:

```
void MatrixOperation::MultiplySequential(vector<float>& vC,
    const vector<float>& vA, const vector<float>& vB,
    int M, int N, int W)
{
    for (int row = 0; row < M; row++)
    {
      for (int col = 0; col < N; col++)
      {
        float sum = 0.0f;
        for(int i = 0; i < W; i++)
          sum += vA[row * W + i] * vB[i * N + col];
        vC[row * N + col] = sum;
     }
   }
}
```

5. Másold be a kijelölt kódot a **`MultiplyAMP`** metódus törzsébe! Ez a művelet a C++ AMP technológiát használja a mátrixok összeszorzására:

```
void MatrixOperation::MultiplyAMP(vector<float>& vC,
    const vector<float>& vA, const vector<float>& vB,
    int M, int N, int W)
{
    concurrency::array_view<const float,2> a(M, W, vA);
    concurrency::array_view<const float,2> b(W, N, vB);
    concurrency::array_view<float,2> c(M, N, vC);
    c.discard_data();
    concurrency::parallel_for_each(c.extent,
    [=](concurrency::index<2> idx) restrict(amp) {
      int row = idx[0]; int col = idx[1];
      float sum = 0.0f;
      for(int i = 0; i < W; i++)
        sum += a(row, i) * b(i, col);
      c[idx] = sum;
    });
}
```

6. A Ctrl+F5 lenyomásával fordítsd le és indítsd el az alkalmazást! A menülistából válaszd a „Using C++ AMP” parancsot, és kattints a Multiply Matrices Sequentially nyomógombra! A számítógéped teljesítményétől függően a számítás 5 és 20 másodpercet fog igénybe venni, majd üzenetet kapsz a számítás sikeres elvégzéséről – a végrehajtási idővel együtt.
7. Most kattints a Multiply Matrices with C++ AMP nyomógombra! Ez a művelet kevesebb időt fog igénybe venni az előzőnél, amint azt a művelet végén megjelenő üzenetből te is láthatod. A számítógépedben lévő grafikus kártyától függően ez akár 10-szer vagy 25-ször gyorsabb is lehet, mint a soros végrehajtás.
8. Zárd be az alkalmazást!

*A gyakorlat teljes forráskódját a **`CppDemo – Complete`** mappában találhatod.*

Hogyan működik?

A 2. és 3. lépésben beillesztett kódok egyszerűen megérthetők – a könyv korábbi fejezeteiben tanult technikák segítségével. Ezek a kódrészletek a soros és az AMP-alapú mátrixszorzást hívják aszinkron módon. A munka oroszlánrészét a **`MultiplySequential`** és **`MultiplyAMP`** műveletek végzik, amelyeket a 4. és 5. lépésben illesztettél a kódba.

Mindkét művelet vektorokkal reprezentálja a mátrixokat. A **MultiplySequential** művelet teljes egészében a legegyszerűbb mátrixszorzási algoritmussal dolgozik, semmilyen további magyarázatot nem kíván.

A **MultiplyAMP** művelet már más, ez nem igazán tiszta az első látásra. A kód első három sora a **vA**, **vB** és **vC** vektorokat definiálja, amelyeket a GPU-n kétdimenziós lebegőpontos tömbként (mátrix) kell kezelni az **a**, **b** és **c** változókon keresztül:

```
concurrency::array_view<const float,2> a(M, W, vA);
concurrency::array_view<const float,2> b(W, N, vB);
concurrency::array_view<float,2> c(M, N, vC);
```

Az **array_view** típus is felelős az adatok mozgatásáért a CPU és a GPU között. A **c.discard_data()** művelet azt deklarálja, hogy a **c** mátrix GPU-n keletkező adatait nem kell visszamásolni a CPU-n a **vC** vektorba. A következő két sor a GPU-n végrehajtandó **parallel_for_each** konstrukciót deklarálja egy lambda kifejezéssel:

```
concurrency::parallel_for_each(c.extent,
[=](concurrency::index<2> idx) restrict(amp) {
```

Ez olyan párhuzamos végrehajtású ciklust definiál, amely végighalad a **c** mátrix elemein (**c.extent**) és a művelet törzsében az aktuális cellát az **idx** indexszel azonosítja, amelynek két dimenziója van (a mátrixcella sor- és oszlopindexe). A **restrict(amp)** a fordítóprogrammal két dolgot közöl. Először is azt, hogy a ciklus törzsét olyan kódra kell fordítani, amely az AMP-vel kompatibilis hardveren futhat (DirectX11 meghajtóval rendelkező GPU). Másodszor, csak olyan utasítások és konstrukciók engedélyezettek a ciklustörzsben, amelyeket az AMP kezelni tud. Például a törzsben nem tudsz fájlokat megnyitni, mert azok a műveletek nem érhetők el a GPU-n. A ciklus törzse egyszerűen kiszámítja az **idx** változóval hivatkozott cella értékét.

Ebben a párhuzamos konstrukcióban az a nagyszerű, hogy a DirectX11 meghajtóra és a GPU-ra bízza a párhuzamosítás szintjét. Ha például a GPU 128 független feldolgozó csatornát támogat, akkor párhuzamosan 128 cella értékét számíthatja ki.

Összegzés

A C++ programozási nyelv a többiekkel (C#, Visual Basic, JavaScript) egyenrangú a Windows 8 stílusú alkalmazások létrehozásában. Az új eszközöknek – mint például a táblagépek, okos telefonok, ultramobil számítógépek – köszönhetően a C++ egyfajta reneszánszát éli. Ezek az eszközök gyengébb számítási képességű CPU-kat és GPU-kat tartalmaznak, mint asztali társaik, és takarékosan kell bánniuk az akkumulátoraikkal, miközben a felhasználói elvárások – különösen a felhasználói élmény szempontjából – igen magasak velük szemben. Az alkalmazások teljesítménye a kulcs ezeknél az eszközöknél! A C++ alacsony szintű konstrukciói és a hardver képességek közvetlen kihasználásának a lehetősége ezt a programozási nyelvet teszik a legjobbá ezeken az eszközökön.

A múltban a C++ soha nem volt olyan jó produktivitást nyújtó nyelv, mint a többi – főleg a felügyelt – programozási nyelvek. Azonban a C++11 szabvány olyan új képességeket adott a nyelvhez, amelyek azt tisztábbá, gyorsabbá és robusztusabbá tették. Azért, hogy a C++ első osztályú nyelv lehessen a Windows 8 stílusú alkalmazások fejlesztéséhez, a Microsoft további bővítésekkel látta el a nyelvet. Ezek tökéletes kapcsolódást biztosítanak a Windows Runtime-mal, és az új konstrukciók – *generics*, érték- és referencia típusok stb. – a C++-t olyan erős nyelvvé alakítják, mint a felügyelt nyelvek.

A C++ néhány egyedi képességgel is bír a Windows8 stílusú alkalmazások fejlesztésében! Támogatja a DirectX technológiákat, és olyan Windows 8 stílusú alkalmazásokat lehet vele létrehozni, amelyek kihasználják a Direct2D, Direct3D, DirectWrite és XAudio2 lehetőségeit. A Microsoft egy új technológiát alkotott, a C++ AMP-t a hardveres gyorsítással rendelkező eszközök (pl. GPU-k) felhasználására, és ezt kizárólag a C++ nyelvvel lehet elérni.

13. A Windows Store és használata

Ebben a fejezetben az alábbi témákat ismerheted meg:

- A Windows Store működése és szabályai
- Próbaváltozatú alkalmazások készítése
- Vásárlás az alkalmazásokból
- Hirdetések beágyazása az alkalmazásunkba
- Telepítési és a jóváhagyási folyamat lépései, eszközei

> ***Figyelem!*** *Az alábbi fejezetben az éppen aktuális szabályokat és lehetőségeket ismertetjük. A Windows Store szabályrendszere változhat, ezért kérünk, mindig nézz utána az aktuális előírásoknak!*

A Windows Store

A Windows 8 egyik legfontosabb újítása a Windows Store, ahol alkalmazásokat vásárolhatunk, kipróbálhatjuk vagy akár ingyen is letölthetjük azokat. A Windows Store első publikus változata a Windows 8 Consumer Preview (Beta) változatában jelent meg. Nem tekinthetjük ezt egyszerű webes szoftverboltnak, ide nem kerülhet fel tetszőleges alkalmazás, csak és kizárólag olyan, amit a Microsoft hitelesített és megfelelőnek talált.

De miért jó a fejlesztőknek az alkalmazásaikat a Windows Store-ba publikálniuk? Van-e potenciál egy újabb piactérben? Hány emberhez juthat el az alkalmazásom? Ilyen és ehhez hasonló kérdések merülhetnek fel egy fejlesztőben vagy egy szoftverfejlesztéssel foglalkozó cégben. Nézzük meg, hogy mekkora piaca is lehet a Windows Store-nak, és vessük össze a többi platformmal!

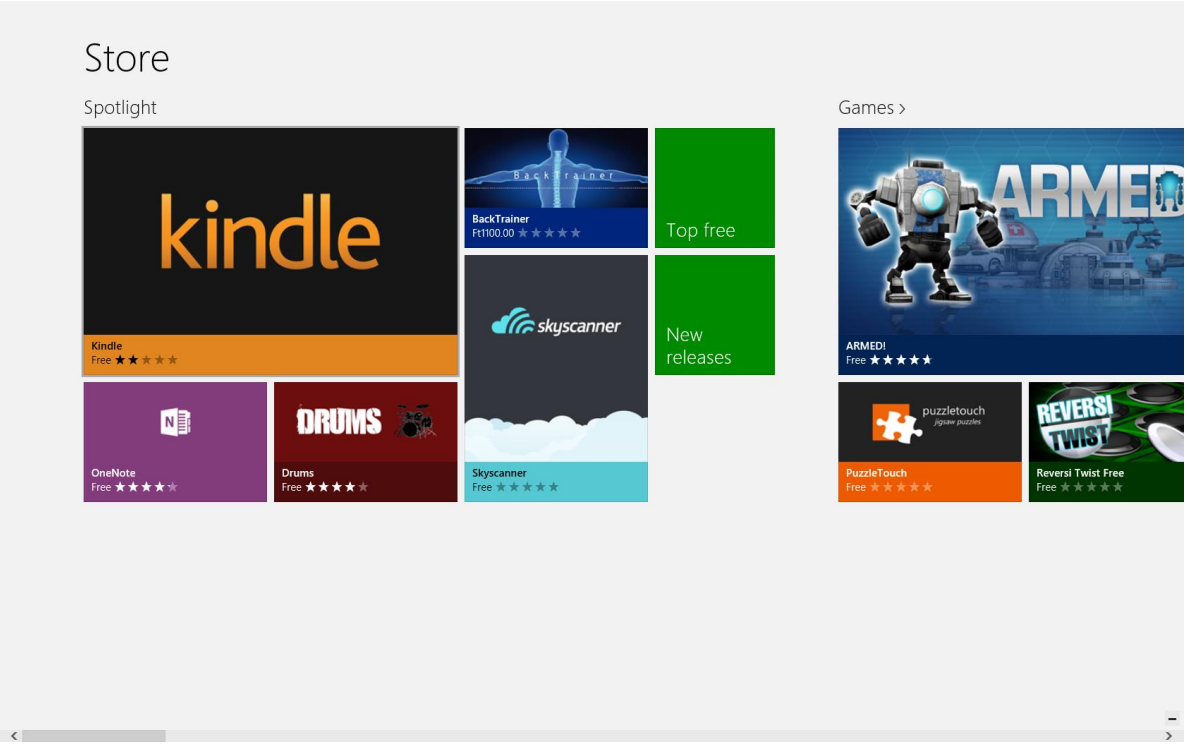
Jelenleg a világon a legnépszerűbb operációs rendszer a Windows. A Windows 7-tel több mint 500 millió felhasználó dolgozik, és ez a szám folyamatosan növekszik. Természetesen még mindig sokan használnak Windows XP-t vagy korábbi operációs rendszert, de az átállás folyamatos és gyors. Ám maradjunk csak a Windows 7 felhasználóinál, leginkább ők lesznek hajlandóak a Windows 8-ra váltani! Több tanulmány szerint már a megjelenés évének végéig, azaz szűk 2-3 hónapos időszakban a felhasználók 10%-a rögtön váltani fog Windows 7-ről Windows 8-ra. Magyarországon ez a szám becslések szerint 600.000 számítógépet jelent, ami a teljes magyar lakosság durván 6%-a. De mit jelent ez a cégek és fejlesztők számára? Hosszú távon 500 millió potenciális vásárlót világszerte, Magyarországon pedig több millió emberhez tudják eljuttatni a terméküket a Windows Store segítségével. Soha egyik elektronikus áruházban se volt ekkora potenciál, mint most a Windows Store-ban van!

De vessük össze a többi piactérrel is! Andoridos telefonból jelenleg 240 millió, tabletből 15 millió, Apple iPhone-ból 120 millió, míg az Apple iPad-ból 40 millió eladott példányról beszélhetünk – és ez a szám folyamatosan nő. A két piactér összesen nem tud akkora számot lefedni, mint a Windows 7. Ezenkívül a Windows ott lesz minden asztali számítógépen, nem kell külön készüléket venni, hisz már vagy OEM-ként érkezik, vagy a felhasználók frissítik majd az operációs rendszerüket.

A Windows Store a nyitás pillanatában több mint 200 országból érhető el, és jelenleg több mint 70 országban helyi pénznemmel is lehet fizetni. Így például, ha magyar Microsoft fiókot (Microsoft Account) használunk, akkor a fizetés forintban fog történni, nem pedig euróban vagy dollárban. Az elérhető országok listája pedig folyamatosan bővül.

A felhasználók Windows 8 stílusú alkalmazásokat gyakorlatilag csak a Windows Store-ból tudnak beszerezni. Ehhez a Windows 8-ban megtalálható – az operációs rendszerrel együtt települő – Store

alkalmazás áll a rendelkezésünkre, amely saját maga is a modern alkalmazásoknál megszokott letisztult felületet biztosítja a felhasználóknak az egyszerű és gyors vásárláshoz, ahogyan azt a 13-1 ábra is mutatja.

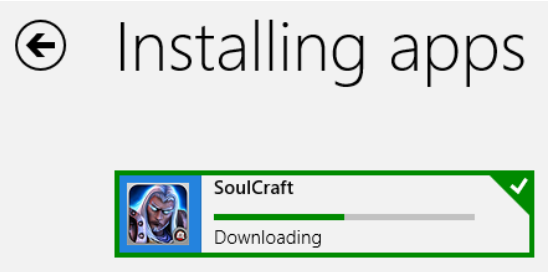


13-1 ábra: A Windows 8 Store alkalmazása

Amikor megnyitjuk a Store-t, a kiemelt alkalmazások kerülnek a középpontba. Láthatjuk továbbá, hogy az alkalmazások kategóriákba vannak rendezve, például a hírolvasók, játékok, közösségi alkalmazások stb. A kategóriák horizontálisan vannak elválasztva, így ha jobbra görgetjük a lapot, a további kategóriákat is láthatjuk. Mindegyiknek van egy kiemelt ajánlata! Természetesen kereshetünk is a Windows Store-ban, ez a Windows 8 saját integrált keresőfelületén történik, amelyet a korábban már megismert Charm bar biztosít.

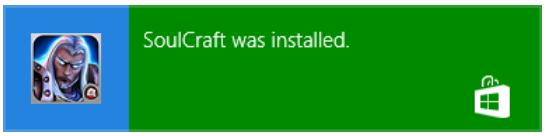
Amennyiben sikerült megtalálnunk a keresett alkalmazást, egyszerűen csak kattintsunk rá! Ekkor megjelenik az adott alkalmazás részletes nézete. Itt láthatjuk az alkalmazás leírását, képernyőképeit, itt írhatunk észrevételt az adott alkalmazásról, és persze itt tölthetjük le az alkalmazás próba- (*Trial*) vagy teljes változatát. Figyeljük meg, hogy a program leírásánál fel van tüntetve, hogy az rendszerünk milyen képességeihez férhet hozzá! Például, hogy az adott alkalmazás hozzá fog férni a mikrofonhoz vagy épp a webkamerához. Így magunk is meggyőződhetünk arról, hogy az adott alkalmazás nem akar-e valami olyan erőforrást elérni a háttérben, amit nem szeretnénk – például nem fogja illetéktelenül a webkamerát használni. Ezeket az erőforrásokat az alkalmazás készítőjének a Capabilities listában fel kell tüntetnie! Amennyiben az alkalmazás mégis megpróbálna elérni egy ilyen speciális erőforrást – anélkül, hogy a listán igényelte volna –, a rendszertől hibaüzenetet kapunk, miszerint az alkalmazás nem férhet az adott eszközhöz vagy funkcióhoz. Ezt a Windows Store-ba feltöltött alkalmazás tesztelésekor is ellenőrzik a Microsoft munkatársai, ilyen módon tehát nem lehet csalni.

Az alkalmazás telepítése nagyon egyszerű. Ingyenes alkalmazás esetén az Install gombra kell rákattintanunk, és máris elindul annak letöltése majd telepítése, ahogyan azt a 13-2 ábra is mutatja.



13-2 ábra: Alkalmazás telepítése a Windows Store-ból

Fizetős alkalmazás esetén végig kell mennünk a vásárlás folyamatán. A legelső vásárlásnál meg kell adnunk a hitelkártyánk adatait, és rendelkezhetünk úgy, hogy adatainkat tárolja el a rendszer, és így a későbbiekben nem lesz szükség a kártyaadatok újbóli megadására. Természetesen nemcsak ingyenes vagy fizetős alkalmazást lehet a Windows Store-ból beszereznünk, akár próbaváltozatot is letölthetünk. Próbaváltozatot támogató alkalmazás készítéséhez néhány egyszerű osztályt kell majd felhasználnunk, ezeket a későbbiekben részletesen megnézzük. Amint elindítjuk a telepítést, a Store alkalmazást be is zárhatjuk. Amikor a telepítés befejeződik, értesítést kapunk a rendszertől a folyamat sikerességéről (13-3 ábra). Az alkalmazás letöltését is figyelemmel kísérhetjük: láthatjuk, hogy hol tart a letöltés, szüneteltethetjük vagy akár le is állíthatjuk azt.



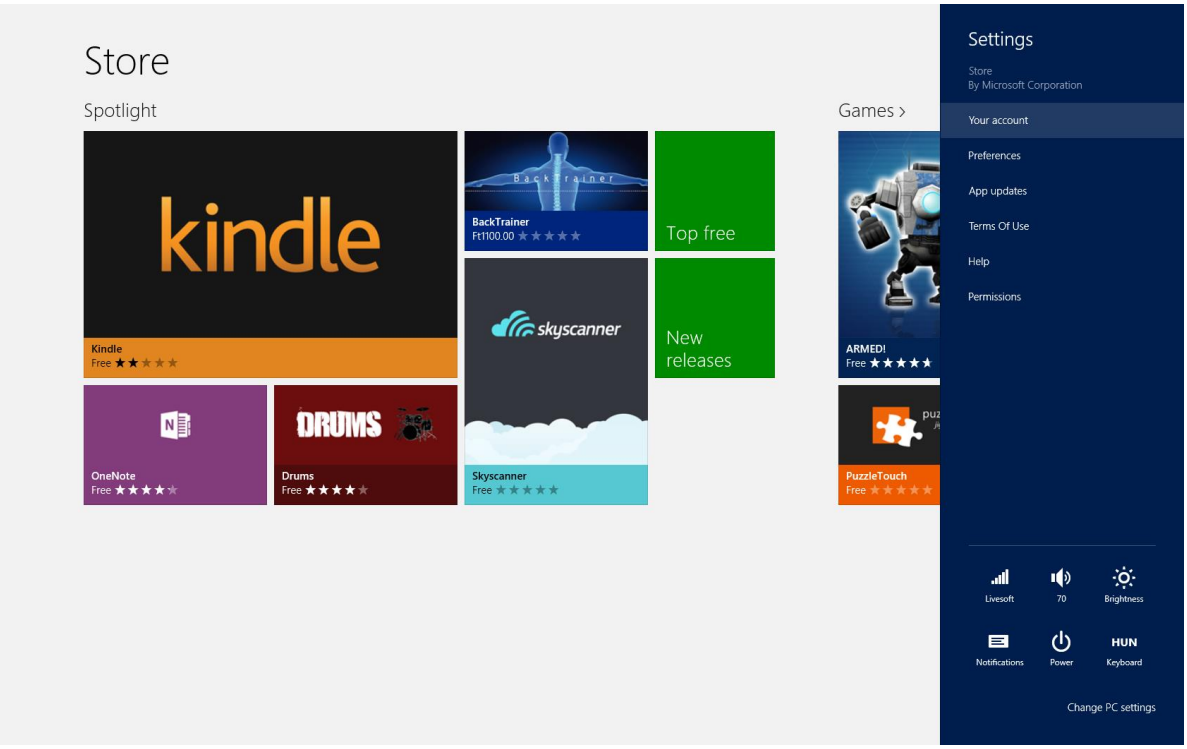
13-3 ábra: Értesítés az alkalmazás sikeres telepítéséről

A Windows Store-ból csak Windows 8 stílusú alkalmazásokat szerezhetünk be, ezek frissítését is támogatja a rendszer. Jogosan merül fel a kérdés, mi van a korábban már létrehozott asztali (*desktop*) alkalmazásokkal? Fel lehet-e azokat tölteni a Windows Store-ba? A Consumer Preview változatban nem lehetett, csak és kizárólag Windows 8 stílusú (a Windows Runtime-ot használó) alkalmazásokat lehetett beszerezni a Windows Store-ból. A Release Preview-val a helyzet valamelyest változott, bár asztali alkalmazást ugyanúgy nem lehet letölteni a Store-ból. De ha fejlesztő cég átesik egy validációs eljáráson, a Store-ban az alkalmazás megjelenhet, viszont csak közvetlen linken keresztül lehet majd letölteni. Vásárolni azonban nem lehet a Store-ból (a jelen információ szerint), azt saját magunknak kell megoldani, esetleg egy disztribútor segítségével.

A Windows Store beállításai

A Store beállításai között (13-4 ábra) válthatunk felhasználót, azaz megadhatjuk, hogy melyik Microsoft Accounttal (korábbi nevén Live ID-val) szeretnénk vásárolni. Megadhatjuk azt is, hogy tárolja el a fizetési opciónkat, vagy minden vásárlás esetén kérdezze meg. Az utóbbi beállítás főleg azoknak a felhasználóknak hasznos, akinek a készülékét többen használják. Így ha a felhasználó valamelyik ismerőse vagy családtagja felmegy a Windows Store-ra, nem tud automatikusan vásárolni a felhasználó terhére.

Ezen a felületen figyelhetjük azt is, hogy az alkalmazás mely készülékeken lesz elérhető. Ugyanis amikor egy alkalmazást vásárolunk, az a megvásárolt Live ID-hoz fog tartozni. A Windows 8 rendszernél van egy megkötés, miszerint egy alkalmazás maximum **öt** különböző készülékre vásárolható meg ugyanazzal a Microsoft Account-tal.



13-4 ábra: A Store alkalmazás beállításai

Ezenkívül menedzselhetjük az alkalmazásainkat, ehhez a Release Preview-ban jelent meg egy Application bar. Kattintsunk a jobb egérgombbal egy üres területen, és a bal felső részén beúszik egy Application bar! A Your apps menüpont alatt kezelhetjük a már megvásárolt, letöltött alkalmazásainkat.

Windows Store fejlesztői regisztráció és szabályok

Ahhoz, hogy fejlesztőként a Windows Store-ban alkalmazásokat tegyünk közzé, szükségünk lesz egy *Windows Store Developer Licence*-re. Fejlesztő cégek esetén ez az összeg 99 USD (kb. 22 000 Ft) évente. Magánszemélyek esetén ez a díj kedvezőbb, 49 USD (kb. 10 800 Ft). Mivel Magyarországról lehet helyi pénzben fizetni, így a fizetés forintalapú. Az alkalmasok árazása egyszerűen történik. Amikor feltöltünk egy alkalmazást a piactérre, meg kell határoznunk annak az árát. A minimális ára 1.49 USD (kb. 330 Ft), míg egy alkalmazás maximális ára 999.99 USD. A befolyt összegek elosztása a következőképpen alakul: minden eladott alkalmazás árának 70%-a a fejlesztő cégé, 30%-a a Microsofté. Azaz, ha egy alkalmazást 10 USD-ért adunk el, abból 7 USD a fejlesztő cégé (személyé), a maradék 3 USD pedig a Microsofté. Viszont ha az alkalmazásunk sikeres, és azt legalább 25.000 USD értékben adjuk el, akkor a fejlesztő cég vagy személy a Microsoft számára kiemelt partner lesz, és innentől minden eladás után az összeg 80%-át kapja a fejlesztő, a maradék 20%-ot a Microsoft. Ez a modell csak az alkalmazás eladásából a fejlesztő céghez befolyó összeggel számol! Az egyszerű „vedd meg és használd” modell mellett a fejlesztő más üzleti modelleket is használhat.

A Windows Store üzleti modelljei

Vegyük sorra milyen üzleti modellek léteznek Windows 8 Metro alkalmazásoknál. Itt most a négy alap üzleti modellt fogjuk bemutatni.

Egyszeri vásárlás

Ezt a modellt mindenki ismeri. A felhasználó bejelentkezik a Windows Store-ba, kiválasztja a neki megfelelő alkalmazást, és ha tetszik, megveszi azt. Ekkor az összeget levonják a felhasználó számlájáról, és az megjelenik a mi virtuális egyenlegünkön. Ezt a modellt tovább lehet bonyolítani azzal, hogy az alkalmazásunkat próbamóddal is ellátjuk, amely funkció- vagy időalapú is lehet. A részletekkel a fejezet későbbi részeiben közelebbről is megismerkedhetünk.

A termék lejárata

A Windows Store lehetőséget biztosít arra, hogy az adott alkalmazás egy adott idő eltelte után lejárjon, azaz azt időről időre újra meg kelljen vásárolni. Például ha megveszünk egy alkalmazást, azt csak egy éven keresztül használhatjuk. Ha alkalmazásunk belső tőzsdei híreket és elemzéseket szolgáltat a felhasználónak, van értelme ebben a modellben gondolkodnunk. Ennek a modellnek a kezeléséhez is kínál API-t a Windows Store a fejlesztők számára.

Előfizetéses modell

Ez a modell azt teszi lehetővé, hogy ha az adott cégnek vagy szolgáltatásnak már van egy meglévő felhasználói bázisa, akkor azt felhasználhatja a Windows 8 stílusú alkalmazásnál is. Például a megrendelő olyan alkalmazást szeretne, ahol egy újság már meglévő előfizetői annak digitalizált változatát olvashatják. Ebben az esetben a cégnek már megvan a saját felhasználó bázisa, és szeretnék a felhasználók életét még kényelmesebbé tenni, hogy azok akár a Windows 8-as táblagépükön is olvashassák a cikkeket. Itt az alkalmazásért a felhasználóknak nem kötelező fizetniük. A megrendelő megadhatja, hogy az alkalmazás ingyenes, de az újságcikkek elolvasásához be kell jelentkezni az előfizetőnek a már meglévő felhasználói névvel és jelszóval. Sőt akár közvetlen módon is fizethet, például az újságot forgalmazó cég saját CRM rendszeréhez kapcsolódva.

Ez is speciális modell, mivel itt meglehet kerülni a Windows Store-t, és a fizetési modell is változik, hisz az összeg ebben az esetben rögtön a saját banki megoldáson keresztül érkezik. Természetesen a saját megoldások sem ingyenesek, azoknak is vannak költségei (valamiből el kell készíteni, javítani és folymatosan fejleszteni azokat), mérlegeljük, hogy számunkra melyik az ideálisabb megoldás! Jogosan merülhet fel a kérdés, hogy mégis a Microsoftnak ebből az előfizetéses modellből mi haszna van? A válasz egyszerű: hivatalos közlés szerint az, hogy a felhasználók sok kiváló alkalmazást tölthetnek le, és sokkal elégedettebbek lesznek, ha a megszokott termékeket jó minőségben megkaphatják.

Reklámok

Az egyik legérdekesebb modell a reklámok használata, ahol a pénz az alkalmazás által megjelenített reklámokból származik. Bár sokan szkeptikusak ezzel a modellel kapcsolatban, de vannak a reklámok bevételéhez kötődő sikertörténetek. Ez a modell úgy működik, hogy az alkalmazásunkban egy adott helyen megjelenik egy-egy reklám. A reklámszolgáltatók több feltétel alapján fizethetnek az alkalmazás fejlesztőjének, például az adott reklámok megjelenési időtartama vagy a felhasználói kattintások száma alapján.

A reklámok főleg olyan alkalmazásoknál lehetnek sikeresek, ahol az alkalmazást nemcsak néhány másodpercre nyitják meg, hanem ahol hosszú órákig is használhatja a felhasználó, tipikusan ilyenek a játékok.

A reklámszolgáltató lehet a *Windows Advertising* szolgáltató, de lehet akár saját lokális szolgáltató is, erre nincs megkötés. A Windows Advertisingnak annyi előnye lehet a többi szolgáltatóhoz képest, hogy fejlesztői szempontból elég egyszerű API-t biztosít, ráadásul Windows 8-as HTML/XAML támogatása is van. Megrendelői oldalról pedig kiemelendő a gazdag riportkészítési lehetőség, valamint az, hogy több országban helyi pénznemben történik a kifizetés. Erre a szolgáltatásra a http://**`pubcenter.microsoft.com`** oldalon lehet regisztrálni, az SDK-t pedig a **`http://www.windowsadvertising.com`** oldalon tölthetjük le. A könyv írásának pillanatában ez a szolgáltatás Magyarországról nem érhető el, de ígéret van arra, hogy hamarosan elérhetővé válik. Amennyiben reklámbevételekre szeretnénk szert tenni, más szolgáltató után kell néznünk, ez lehet magyar vagy külföldi is.

A modellek összefoglalása

Mint láthattuk, a Windows Store-ban az alkalmazásunk elég sokféle üzleti modellt használhat. Sőt ezek a modellek szabadon kombinálhatóak! Például az alkalmazásunkat egyszeri vásárlással veheti meg a felhasználó, de biztosítunk a számára egy funkcióiban korlátozott próbaváltozatot úgy, hogy folyamatosan új reklámokat kell nézni. Ebből is látszik, hogy ez a rendszer rendkívül rugalmas, a fejlesztők és a cégek dolgát egyaránt megkönnyíti. A következőkben ezeknek a Windows Store API-knak a felhasználásával is részletesebben meg fogunk ismerkedni.

Alkalmazások összekapcsolása a weboldalunkkal

A Windows 8-ban bevezetésre került egy új weboldal összekapcsolási lehetőség, amivel még közelebb hozhatjuk felhasználóinkat a Windows 8 ökoszisztémához. Felhasználóink az alkalmazásokat eddig szinte csak weben keresztül szerezték be. Megszokták, hogy ha ellátogatnak egy-egy termék honlapjára, akkor onnan letölthetik kedvenc alkalmazásukat. Mivel a Windows 8 stílusú alkalmazást csak a Windows Store-ból tudják beszerezni, ezért ezen a szokásukon változtatniuk kellene. A fejlesztő persze belinkelheti a Windows Store-ban található alkalmazás elérési címét, de ez a megoldás mégsem nyújtja ugyanazt az érzést. Nézzük meg, hogy értesíthetjük a felhasználókat arról, hogy ha már úgyis ellátogatnak a honlapunkra, akkor szerezzék is be az általunk készített alkalmazást!

Ha a felhasználó a készülékén elindítja az Internet Explorer Start képernyőn lévő változatát – az itt leírt funkciók csak és kizárólag az Internet Explorernek ebben a változatában érhetőek el, az asztalról indítottról nem –, és az oldalunkra navigál, az Internet Explorer felderíti az egyedi metainformációt. Ha az Internet Explorer az oldalunkon megfelelő metainformációt talál, akkor a „page tool" ikon mellett megjelenik egy kis pluszjel (13-5 ábra), arra utalva, hogy ezen az oldalon valamilyen extra funkciót lehet elérni. Ha erre a gombra kattintunk, akkor a felugró menüben ott lesz a „Get app for this site" menüpont. Amint rákattintunk, elindul a Store alkalmazás, és letölthetjük (vagy épp megvehetjük) azt.

13-5 ábra: Get app for this site

Ha sikerült letöltenünk az alkalmazást, és visszatérünk az oldalra, észrevehetjük, hogy most már ha rákattintunk a „page tool" gombjára, nem a „Get app for this site" menüpont fogad, hanem a „Switch to ... app" menüpont. Ha erre rákattintunk, akkor a gépünkre korábban már telepített alkalmazás (13-6 ábra) indul el.

De mit is kell ehhez módosítani az oldalunkon? Tulajdonképpen csak a **<head>** elemben kell speciális metainformációkat elhelyezni.

13-6 ábra: Váltás a Cute the Rope alkalmazásra

Íme, egy egyszerű példa:

```
<meta name="msApplication-ID" content="microsoft.build.App"/>
<meta name="msApplication-PackageFamilyName" content="microsoft.build_8wekyb3d8bbwe"/>
```

Ha az Internet Explorer-rel navigálunk el az oldalra, akkor a rendszer ellenőrzi, hogy ez az alkalmazás telepítve van-e már. Ha nincs, akkor az alkalmazás Windows Store helyére mutató közvetlen linket kapunk. Ha az alkalmazás telepítve van, akkor elindíthatjuk azt. Az Internet Explorer asztali változatánál ez a funkció nem működik.

Az előzőekben bemutatott metainformációk mindenképp szükségesek, legalább az **`msApplication-Id`** és az **`msApplication-PackageFamilyName`** jellemzőket feltétlenül meg kell adnunk. De ezen kívül vannak még további metainformációk is, ahogyan azt a 13-1 táblázat összefoglalja. Ezek értékeit a Windows Store oldalról tudjuk kimásolni, majd metainformációként weblapunkba beírni, miután az alkalmazásunkat már feltöltöttük.

13-1 táblázat: A Windows Store alkalmazásokra utaló metainformáció

Megnevezés	Leírás
`msApplication-ID`	Az alkalmazás egyedi azonosítója (*Application Manifest*), kötelező megadni.
`msApplication-PackageFamilyName`	A *Package Family Name*-et a Visual Studio készíti el, amikor az alkalmazást publikálja. Kötelező megadni.
`msApplication-Arguments`	Megadhatunk további argumentumokat. Ha nem adunk meg ilyet, az Internet Explorer alapértelmezett módon az oldal URL-jét illeszti be. Ennek a jellemzőnek a használata opcionális.
`msApplication-MinVersion`	Rákényszeríthetjük a felhasználót arra, hogy csak bizonyos verziószám felett tudja az oldalon a „Switch App” funkciót használni. Ha a felhasználó az alkalmazás egy korábbi verziójával rendelkezik, akkor az Internet Explorer átirányítja a Windows Store-ba az alkalmazáshoz, hogy a felhasználó először frissítse a régebbi változatot. Ennek a jellemzőnek a használata opcionális.
`msApplication-OptOut`	Meghatározhatjuk, hogy milyen műveleteket ajánlunk fel a „page tool” gombnál a böngészőben. Az install érték azt jelzi, hogy csak a telepítést ajánljuk fel. A switch arra utal, hogy ugyan az oldalról az alkalmazást indíthatjuk, de a telepítést nem ajánlja fel. A both érték esetében mindkét lehetőség elérhető.

Látható, hogy alkalmazásunkat egyszerűen integrálhatjuk a weboldalunkkal. Sőt akár további beágyazási lehetőségeink is vannak az **`msApplication-Arguments`** segítségével.

Windows Store szabályok

Ebben a részben bemutatjuk, hogy milyen feltételeket kell az alkalmazásunknak teljesítenie ahhoz, hogy az a Windows Store-ba bekerüljön. Mivel ezek a szabályok változhatnak és kiegészülhetnek, kérünk, mindig nézz utána az aktuális állapotnak!

A szabályokat és a felhasználási feltételeket az alábbi oldalon tekinthetjük meg:
http://msdn.microsoft.com/en-us/library/windows/apps/hh694083.aspx

Az alábbiakban néhány fontosabb szabályt emelünk ki.

1. Az alkalmazásnak teljes funkcionalitásúnak kell lennie!

   Ezt a feltételt a Microsoft mérnökei az alkalmazás beküldésekor tesztelik és ellenőrzik. Nem lehet olyan alkalmazást feltölteni, amelyben vannak még nem működő funkciók.

   Abban az esetben, ha olyan alkalmazásról van szó, amihez egy adott külső rendszerbe be kell jelentkezni, a beküldéskor meg kell adni egy teszt célú hozzáférést, amit a mérnökök majd a tesztelés során használhatnak. Ha esetleg valamilyen szervert kell elérnie az alkalmazásnak, akkor adjuk meg annak elérését is!
2. Az alkalmazás nevének egyedinek kell lennie!

   Az alkalmazás csak olyan nevet használhat, amely Windows Store-ban teljesen egyedi. A nevet megadhatjuk több nyelven is, de ezeknek minden nyelven egyedinek kell lennie!
3. Fel kell tüntetnünk a terméktámogatás elérhetőségét is.

   A Support contact info mezőben kell megadni ezt az információt az alkalmazás feltöltésekor.
4. Legalább egy olyan piacot meg kell határoznunk, ahonnan az adott alkalmazást le lehet tölteni és használatba lehet venni.
5. Az alkalmazás feltöltésekor legalább egy képernyőképet meg kell adnunk az alkalmazásról.

   Minden kép formátuma PNG (Portable Network Graphics) kell, hogy legyen 1366x768 pixeles méretben.
6. Egy alkalmazás nem állhat csak reklámokból!
7. Reklámokat nem jeleníthetünk meg a csempéken, az Application báron, és értesítésként sem érkezhetnek reklámok!
8. Az alkalmazás nem tartalmazhat felnőtt tartalmat!
9. Az alkalmazás nem lehet gyűlöletkeltő, nem buzdíthat erőszakra, nem sérthet faji, etnikai, nemzeti, nyelvi, vallási vagy más társadalmi csoportot!
10. Az alkalmazás nem tartalmazhat olyan tartalmat vagy funkciót, amely a felhasználót jogellenes magatartásra ösztönzi.
11. Az alkalmazás nem tartalmazhat semmilyen obszcén és trágár tartalmat.
12. Az alkalmazás nem tartalmazhat olyan tartalmat, amely ösztönzi vagy elősegíti a túlzott vagy felelőtlen alkoholfogyasztást és a dohánytermékek fogyasztását. Nem ösztönözhet továbbá kábítószerre és fegyverek használatára.
13. Csak és kizárlóag Windows Runtime API-t lehet használnunk! A Windows-ból származó API-kat (Win32, Kernel32 stb.) nem használhatunk!
14. Az alkalmazásba nem lehet kezeletlen kivétel, az nem fagyhat le a futtatás során!
15. Ha frissítjük az alkalmazásunkat, nem csökkenthetjük annak funkcionalitását!
16. Az alkalmazásunknak érintés gesztusokat, billentyűzetet és egeret egyaránt támogatnia kell!
17. Az alkalmazásnak támogatnia kell a „snapped” képernyőmódot!
18. Az alkalmazásnak minimum 1024x768-as felbontást kell támogatnia.

19. Bizonyos performancia kritériumokat is teljesítenie kell az alkalmazásnak:
    - 5 másodpercen belül el kell indulnia,
    - 2 másodperc alatt el kell aludnia.

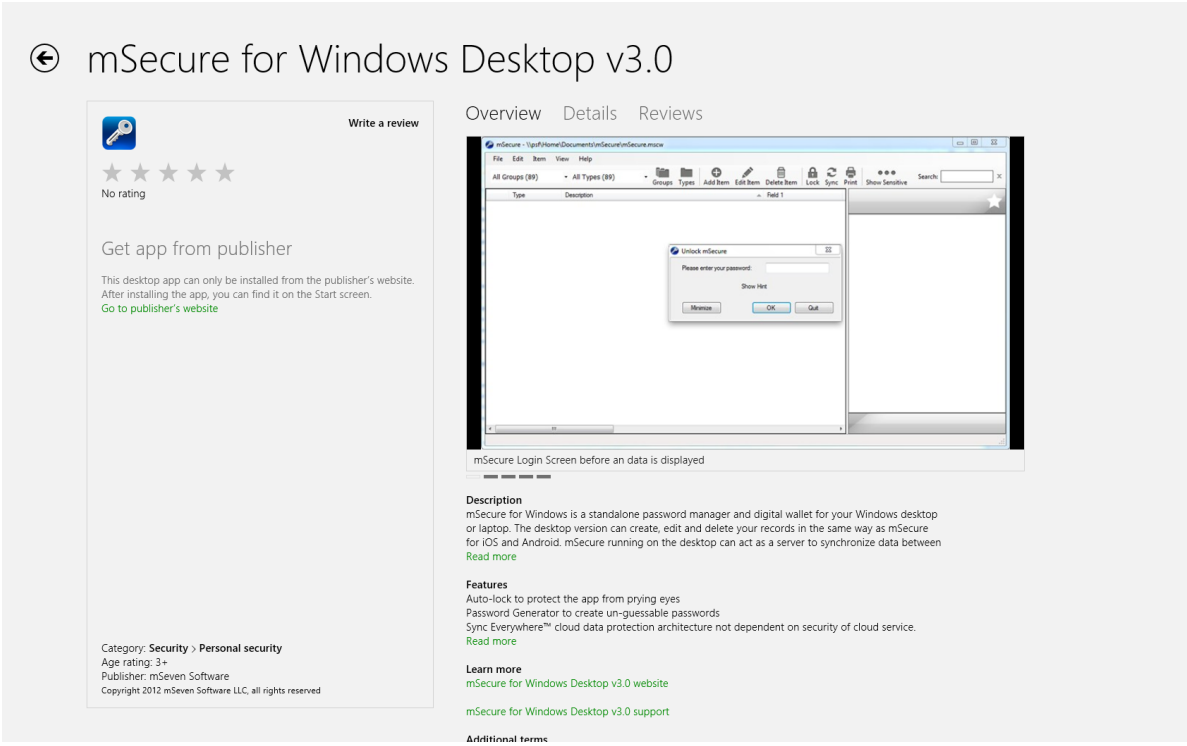
    Ezeket a kritériumokat egy specifikált lassú gépen is tartani kell! Ennek a tesztelésére a fejezet egy későbbi részében részletesen kitérünk.

Az asztali alkalmazásokra más szabályok vonatkoznak:

Bár letölteni nem lehet azokat a Windows Store-ból, de ott elhelyezni, azokon hirdetni engedélyezett. Ehhez az alkalmazásnak át kell esnie a Windows Desktop App Certification eljáráson. Ennek a követelményeiről itt olvashatunk: http://msdn.microsoft.com/en-us/library/windows/desktop/hh749939.aspx.

1. Meg kell adnunk a közvetlen linket az alkalmazás letöltési helyére, köztes oldalt nem lehet használni! Így nincs lehetőség az oldal linkjének meghamisítására.
2. Két hozzáférési linket is megadhatunk. Egyiket a 32 bites, a másikat pedig a 64 bites változat számára.

   A megadott adatoknak egyeznie kell a Windows Store-on (13-7 ábra) és a saját oldalunkon, például az alkalmazás nevének, árának és verziószámának nem szabad eltérnie!



13-7 ábra: Az alkalmazás adatai a Store felületén

Próbaváltozat készítése (Trial mód)

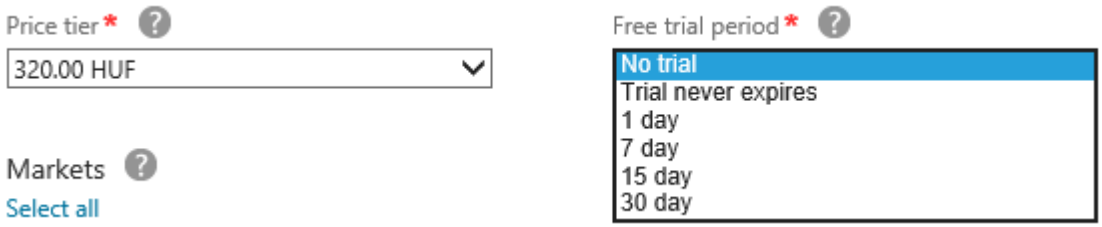
Eddig áttekintettük a Windows Store funkcióit, működését és felhasználási feltételeit. Itt az ideje felkészítenünk az alkalmazásunkat arra, hogy a Windows Store lehetőségeit kihasználja! Először megismerkedünk a próbaváltozatú alkalmazások elkészítésével.

Több piactérrel ellentétben a Windows Store-ra nem kell külön alkalmazást készíteni ahhoz, hogy az alkalmazás próbaváltozatát letöltse a felhasználó. Más piactereken ezért van egy-egy alkalmazásból „Light", „Trial" vagy „Free" utótagú alkalmazás. Ott a fejlesztőknek külön kell fordítaniuk egy alkalmazást, amelyből kiszedik vagy inaktívvá teszik azokat a funkciókat, amiket a próbaváltozatú alkalmazásban nem akarnak elérhetővé tenni.

A Windows Store esetében erre nincs szükség! A felhasználó ilyenkor is a teljes alkalmazást fogja letölteni, ugyanúgy mint azok, akik azt megvásárolják, de a próbaváltozatú alkalmazásnál az **IsTrial** tulajdonság értéke **true** lesz. A fejlesztő így egyszerűen kezelheti, hogy mely funkciók és kódrészletek érhetők el próbamódban és melyek nem. Ha a felhasználó megvásárolja az alkalmazást, nem kell egy újabb változatot letöltenie, ilyenkor a Windows Store egyszerűen csak átállítja az **IsTrial** kapcsolót **false** értékűre, és máris használható a teljes értékű alkalmazás.

Sokan nem is gondolják, hogy fontos lenne az alkalmazásukat próbamóddal ellátni, mondván, ha a felhasználó úgy gondolja, akkor vegye meg azt. De sokan inkább kipróbálnák az alkalmazást, mielőtt pénzt adnának ki érte. Ráadásul a próbamódú alkalmazásokat a többi piactér statisztikája szerint 70-szer többen töltik le, mint az azonos teljes alkalmazásokat! A letöltők átlagosan 10%-a meg is veszi az alkalmazást a próbaváltozat használata után. Fontos tehát a próbamód és a hozzá jól megválasztott stratégia. A Windows Store esetén alap esetben kétféle ilyen stratégiáról beszélhetünk. Az egyik az idő alapú, a másik a funkció alapú. Ismerkedjünk meg ezekkel!

Az idő alapú trial stratégia a legegyszerűbb: itt a fejlesztőnek igazából nem is kell kódolnia. Elég csak a Windows Store-ban beállítania a „Free trial perdiod” jellemzőt, és a Store a többit elintézi (13-8 ábra).



13-8 ábra: A próbamód lejártának beállítása

Megadhatjuk, hogy 1, 7, 14 vagy 30 nap múlva járjon le az alkalmazás, és arra is van lehetőség, hogy sohasem. Ez nagyon egyszerű stratégia: a felhasználó meghatározott ideig ingyen és teljes funkcionalitással használhatja alkalmazásunkat.

A funkcionálisan korlátozott változattal a fejlesztőnek már több dolga van. Meg kell határoznia, hogy mely funkciók érhetőek el a próbaváltozatban és melyek nem. Ehhez szükség van a Windows.ApplicationModel.Store névtérben található típusokra.

```
using Windows.ApplicationModel.Store;
```

Ebben a névtérben a Store kezeléshez szükséges osztályok találhatók. Először két kulcsfontosságú osztállyal ismerkedjünk meg! Az egyik a **CurrentApp,** a másik pedig a **CurrentAppSimulator**. A két osztály tulajdonképpen pontosan ugyanazt tudja, csak a **CurrentAppSimulator** osztályt fejlesztéshez és teszteléshez kell használnunk, az a Windows Store-t szimulálja a fejlesztői gépen. A **CurrentApp** osztályt akkor használhatjuk, ha már véglegesítettük az alkalmazást.

> *Fontos, hogy a Windows Store-ba beküldött kész alkalmazásoknál csak a **CurrentApp**-ot szabad használni! Amennyiben a **CurrentAppSimulator**-t benne hagyjuk a kódban, visszautasítják az alkalmazásunkat.*

Nézzünk egy egyszerű példát arra, hogy hogyan is lehet megállapítani, hogy az alkalmazásunk **Trial** módban fut-e vagy sem:

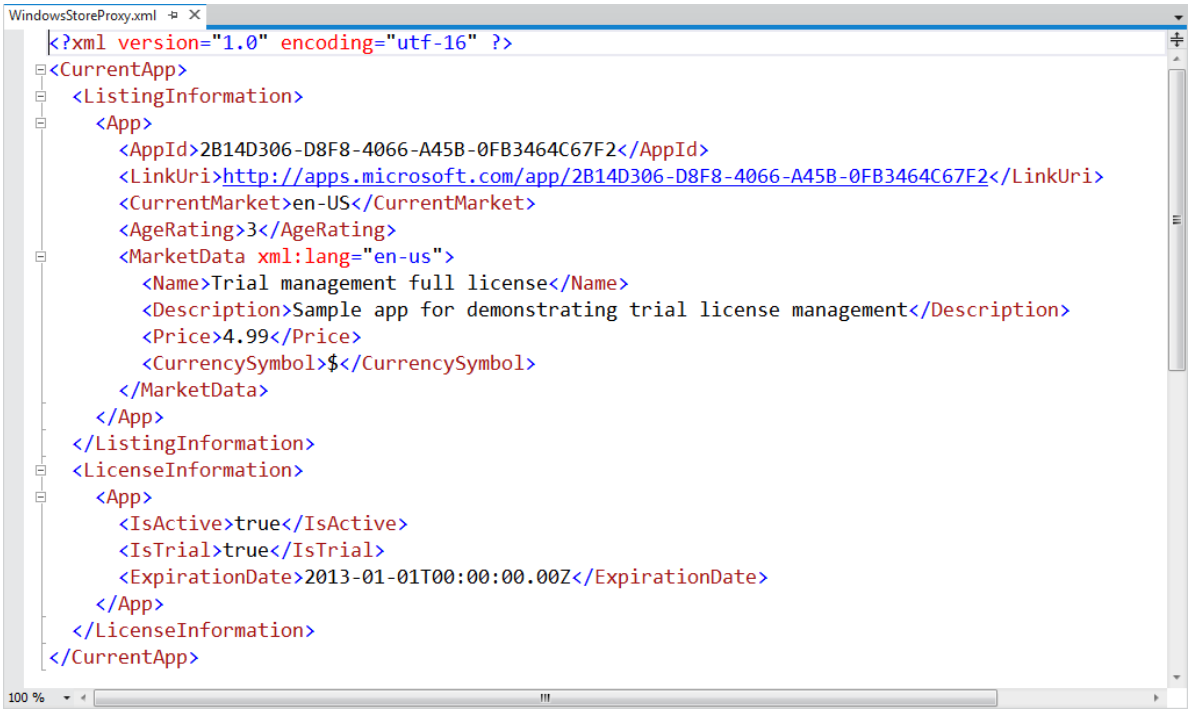
```
LicenseInformation myLic = CurrentAppSimulator.LicenseInformation;
if (myLic.IsActive)
{
    if (myLic.IsTrial)
    {
        //Trial mód
    }
    else
```

```
    {
        //Teljes
    }
}
else
{
    //Érvénytelen licenc
}
```

Először a **`CurrentAppSimulator.LicencInformation`**-nal lekérdezzük az alkalmazásunk licenc információját. Ezt követően megvizsgáljuk, hogy a licenc aktív-e vagy érvénytelen. Ennek nincs köze a próbamódhoz. Egyszerűen figyelhetjük, hogy a felhasználó valóban érvényes licenccel rendelkezik-e. Ezt mindig célszerű ellenőrizni. Majd megvizsgáljuk az **`IsTrial`** tulajdonság segítségével, hogy próbaváltozatú-e az alkalmazás. Abban az esetben, ha igen, akkor letilthatunk bizonyos funkciókat, vagy feltételeket határozhatunk meg. Például ha van egy játékunk, meghatározhatjuk, hogy csak az első 10 pályát lehet próbamódban játszani, a többi pálya eléréséhez a felhasználónak meg kell vennie az alkalmazást.

Ez eddig egyszerű, de mégis hogy lehet ezt a működést kipróbálni? Hogyan lehet módosítani a tesztelés során azt, hogy most az alkalmazás éppen próbamódban fut-e vagy sem? Hogyan lehet befolyásolni, hogy mikor jár le?

A CurrentAppSimulator használatával az alkalmazáshoz tartozó licencet szimulálni tudjuk a fejlesztői gép **%userprofile%\appdata\local\packages\<package-moniker>\localstate\microsoft\Windows Store\Apidata** mappájában lévő **`WindowsStoreProxy.xml`** állomány segítségével (13-9 ábra). Az állomány módosításával az alkalmazásunk licencét módosítjuk a fejlesztői gépen.



13-9 ábra: A WindowsStoreProxy.xml fájl a szerkesztőben

A **`<LicenseInformation> <App>`** elemén belül láthatjuk az **`IsActive`**, az **`IsTrial`** és az **`ExpirationDate`** bejegyzéseket. Ha itt például az **`IsTrial`** értékét **`false`**-ra állítjuk, akkor az alkalmazás következő indulásakor azt már teljes funkcionalitásúnak látjuk. De ugyanúgy szimulálhatjuk azt is, hogy ha érvénytelen a licenc, vagy az alkalmazás használati ideje lejárt.

Ne feledjük, hogy ha fel akarjuk tölteni az alkalmazásunkat a Windows Store-ra, a **`CurrentAppSimulator`**-t le kell cserélnünk a **`CurrentApp`**-ra!

```
LicenseInformation myLic = CurrentApp.LicenseInformation;
```

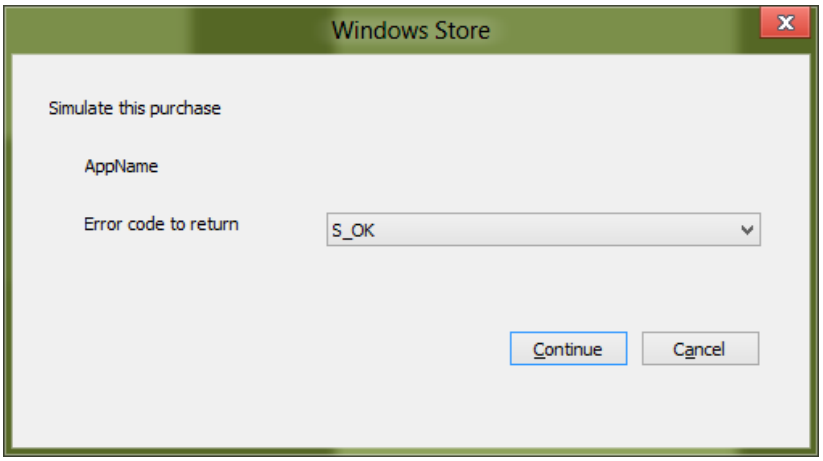
Vásárlás az alkalmazásból

Egyszerűen meg tudjuk állapítani azt, hogy az alkalmazás próbaváltozatként érhető-e el vagy sem. Alkalmazásunkat arra is fel kell készítenünk, hogy ha a felhasználónak tetszik az, akkor meg is tudja vásárolni! Persze közvetlen link segítségével a piactérre navigálhatunk, de az kényelmetlen lehet, mert a felhasználó kiesik a munkamenetéből. Ráadásul fejlesztési időben is célszerű volna vizsgálni, hogy az alkalmazásunk hogyan reagál arra, hogy a felhasználó megveszi az alkalmazást. A **`CurrentAppSimulator`** segítségével ezt is megtehetjük.

Ahhoz, hogy az alkalmazásunkon belülről vásárolhasson a felhasználó, mindössze a **`CurrentAppSimulator`** osztály **`RequestAppPurchaseAsync`** műveletét kell meghívnunk. Figyeljünk arra, hogy ez a metódus aszinkron végrehajtású! Abban az esetben, ha a művelet egyetlen argumentuma (**`includeReceipt`**) **`true`** értékű, egy XML formátumú szöveget kapunk vissza a vásárlás részleteivel. A **`false`** értékű argumentum mellett üres szöveg jön vissza:

```
CurrentAppSimulator.RequestAppPurchaseAsync(false);
```

Ha ez a kódsor lefut, akkor a 13-10 ábrán szereplő ablak jelenik meg, kizárólag csak a **`CurrentAppSimulator`** használata esetén. Itt kiválasztjuk, hogy a vásárlás milyen státusszal térjen vissza. Ha az **`S_OK`** kódot választjuk ki, és a Continue gombra kattintunk, akkor azt szimuláljuk, hogy minden rendben lezajlott a vásárlással, és a felhasználó megvette az alkalmazásunkat. Természetesen szimulálhatjuk azt is, hogy a felhasználó mégsem vette meg az alkalmazást, vagy valamilyen egyéb hiba történt a vásárlás során.



13-10 ábra: Visszatérési érték kiválasztása a szimulált vásárlás során

Ha megvettük az alkalmazást, akkor azok a funkciók, amelyek eddig próbamódban nem működtek, most már menni fognak, hiszen az **`IsTrial`** tulajdonság értéke **`false`**. A felhasználót azonban még nem értesítettük arról, hogy az alkalmazás licencével kapcsolatban változás történt. A **`LicenseInformation`** osztálynak van egy **`LicenseChanged`** eseménye, amire feliratkozhatunk. Ezt az eseményt kezelve például az alkalmazásunkon belül értesíthetjük a felhasználót a sikeres vásárlásról.

```
LicenseInformation myLic = CurrentAppSimulator.LicenseInformation;
myLic.LicenseChanged += myLic_LicenseChanged;

...

void myLic_LicenseChanged()
{
    //A licenc megváltozott!
}
```

Funkciók vásárlása

Az, hogy egy alkalmazást megvásárolunk, tulajdonképpen megszokott dolog. A Windows Store arra is lehetőséget biztosít, hogy ne csak a teljes alkalmazást tudjuk eladni, hanem akár bizonyos funkciókat vagy tartalmakat részenként. De akár azt is megtehetjük, hogy az alkalmazásunk alapja ingyenes, de ha valamilyen extra funkcióhoz szeretne a felhasználó hozzájutni, akkor azt külön meg kell vásárolnia, akár darabonként. Például egy játékot kibocsáthatunk úgy, hogy a felhasználó ingyen megkap egy adott pályacsomagot. Ha viszont a többi pályára is kíváncsi, akkor azokat egyesével (csomagokban) meg kell vennie. Így nem egyszer keresünk a terméken, hanem annyiszor, ahányszor az adott tartalmakat eladtuk. Persze ennek menetét szabadon kombinálhatjuk. Ahhoz, hogy egy-egy funkció megvásárlását tesztelni tudjuk, ezt a folyamatot a fejlesztői gépen valahogy szimulálnunk kell. Ennek a tesztelésére is a **`CurrentAppSimulator`** osztály és a hozzá tartozó **`WindowsStoreProxy.xml`** állományt használhatjuk. Először nyissuk meg a **`WindowsStoreProxy.xml`** fájlt (**%userprofile%\appdata\local\packages\<package-moniker>\localstate\microsoft\Windows Store\Apidata)**! Ebben az XML-ben a **`<ListingInformation>`** elemen belül már van egy **`<Product>`** bejegyzés, és ennek a **`ProductId`** jellemzőjének értéke 1. Egy **`<Product>`** elem mindig egy adott funkciót és az ahhoz tartozó tulajdonságokat írja le. Itt határozzuk meg az adott funkció nevét és árát. Tetszőleges számú **`<Product>`** elem lehet, egyszerűen csak létre kell hoznunk az XML sémának megfelelő új elemeket. Például egy játékalkalmazásunk kapcsán itt írhatjuk le, hogy az adott pályacsomag – amit szeretnénk a felhasználónak eladni – mennyibe kerül és mi a publikus neve:

```
<ListingInformation>
...
   <Product ProductId="SpaceMap" LicenseDuration="0">
      <MarketData xml:lang="en-us">
         <Name>Space Map - Moon</Name>
         <Price>3.00</Price>
         <CurrencySymbol>$</CurrencySymbol>
      </MarketData>
   </Product>
</ListingInformation>
```

Fejlesztési időben azt is szimulálnunk kell, hogyan reagál az alkalmazás arra, ha a felhasználó megvásárolja ezt a funkciót. A **`WindowStoreProxy.xml`**-ben a **`<LicenseInformation>`** elemen belül láthatjuk az **`<App>`** után a **`<Product>`** bejegyzéseket. Az itt található **`ProductId`** értékeknek meg kell egyeznie a **`<ListingInformation>`** elemben található **`ProductId`**-val, hogy a rendszer a hivatkozásokat egyeztetni tudja. Ennek a **`<Product>`**-nak is van egy IsActive tulajdonsága. Ha ennek értéke **`true`**, akkor ez a csomag (funkció) a felhasználó birtokában van, ha **`false`**, akkor nincs, és meg kell vennie.

> *Ebben az esetben is a teljes alkalmazás van a felhasználónál, és pusztán egy flag beállításával érvényessé válik a licenc. A felhasználó így külön letöltés nélkül használhatja a funkciót.*

Most, hogy a **`WindowsStoreProxy.xml`** fájlt felparamétereztük, nézzük meg, hogyan lehet az alkalmazásunkon belülről ellenőrizni, hogy a felhasználó megvette-e ezt a funkciót, és ha nem, akkor hogy tudjuk a funkció megvásárlásának folyamatát leellenőrizni.

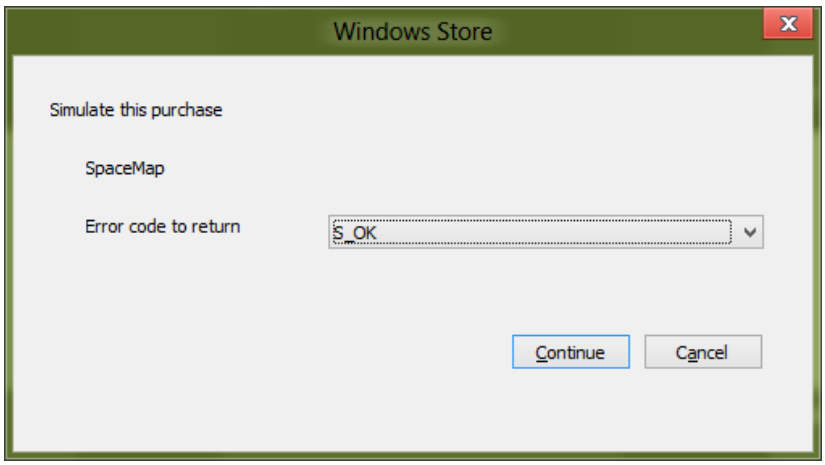
Ehhez a **`CurrentAppSimulator.Licenseinformation.ProductLicense`** gyűjteményt kell használnunk. Meg kell adnunk az adott funkció nevét, ez a név fejlesztési időben ugyanaz, amit a **`WindowsStoreProxy.xml`** fájlban megadtunk. A licenceket indexerrel vagy a **`TryGetValue`** metódussal is lekérdezhetjük. Mindkét módszer estében a visszatérési érték egy **`ProductLicense`**, amelynek az **`IsActive`** tulajdonságával meghatározhatjuk, hogy az adott funkció a felhasználó birtokában van-e vagy sem. Ha nincs, tiltsuk le a kapcsolódó funkciót vagy menüpontot! Ha megvette, akkor engedélyezzük azt számára!

```
ProductLicense myProduct = CurrentAppSimulator.LicenseInformation.ProductLicenses["SpaceMap"];
if (!myProduct.IsActive)
{
    //Még nincs megvásárolva a funkció!
    //Tiltsuk le az adott funkciót.
}
else
{
    //A funkció a birtokunkban van!
    //Engedélyezzük a felhasználó számára a használatát.
}
```

Az előzőekben már láttuk, hogyan vásárolhatunk egy alkalmazást a programunkon belül, most nézzük meg a funkciók megvásárlását! Az elv tulajdonképpen ugyanaz, csak jelen esetben a **`RequestProductPurchaseAsync`** műveletet kell használnunk. A metódusnak két argumentumot adhatunk át: az első a funkció egyedi azonosítója (**`ProductId`**), a második pedig a korábban már megismert **`includeReceipt`**.

```
CurrentAppSimulator.RequestProductPurchaseAsync("SpaceMap", false);
```

Amikor a felhasználó úgy dönt, hogy meg akarja vásárolni a funkciót, fejlesztési időben ugyanaz az ablak fog felugrani, mint ami az alkalmazás vásárlásakor (13-11 ábra).



13-11 ábra: Visszatérési érték kiválasztása funkció vásárlásakor

A fejlesztő így tesztelheti, hogyan viselkedik az alkalmazása, ha az adott funkciót megvásárolják, vagy ha a vásárlás közben valamilyen hiba történik, esetleg az adott funkciót a felhasználó már megvásárolta.

> *A LicenseChanged esemény akkor is bekövetkezik, ha csak egy funkciót vásároltunk, és nemcsak a teljes termék megvásárlásakor.*

Láthatjuk, hogy nemcsak úgy adhatunk el terméket, hogy magát az alkalmazást adjuk el, hanem úgy is, hogy funkciónként adjuk el azt a Windows Store segítségével.

További termékspecifikus információ lekérdezése

Gyakran szükségünk lehet arra is, hogy a piactérre feltöltött alkalmazásunk kapcsán annak ott lévő beállításait le tudjuk kérdezni. Például ha van egy alkalmazásunk, amely próbamódban az indítás után felajánlja számunkra, hogy vegyük meg 10 USD-ért. Igen ám, de mi a Windows Store-ban akcióztuk az alkalmazást, és most már csak 5 USD-be kerül! Ha az alkalmazásunkba beégettük az árát, a felhasználó nem fog értesülni erről az akcióról, és lemaradunk egy vásárlásról. A Windows Store-ból ezeket az

alkalmazásspecifikus információkat bármikor lekérdezhetjük, és nemcsak az alkalmazás egészére nézve, hanem akár az egyes funkciókra is.

Ehhez a **`LoadListingInformationAsync`** műveletet használhatjuk fel:

```
ListingInformation appInfo = await CurrentAppSimulator.LoadListingInformationAsync();
```

Nézzük meg, hogy milyen fontosabb tulajdonságai vannak a **`ListingInformation`** osztálynak – mindegyik csak olvasható –, amit a 13-2 táblázat foglal össze!

13-2 táblázat: A ListingInformation osztály tulajdonságai

Tulajdonság	Leírás
`AgeRating`	Az alkalmazás korhatárát kérdezhetjük le vele.
`CurrentMarket`	A felhasználó piaci beállítása, azt mondja meg, hogy az adott felhasználói fiók milyen piactérhez tartozik. Például egy az USA-ban lévő felhasználó esetében ez az **`en-us`** érték.
`Description`	Az alkalmazás leírása, annak függvényében jelenik meg, hogy mi a felhasználó piaci beállítása.
`FormattedPrice`	Az alkalmazás ára, a felhasználó piacterének megfelelő valutajellel van formázva.
`Name`	Az alkalmazás neve, annak függvényében jelenik meg, hogy mi a felhasználó piaci beállítása.
`ProductListing`	Az alkalmazáshoz tartozó funkciók vagy termékek listája. Ezeket a funkciókat vásárolhatja meg az alkalmazáson belülről a felhasználó.

Nézzünk egy egyszerű példát arra, hogy hogyan kérdezzük le ezeket az információkat a Windows Store-ból:

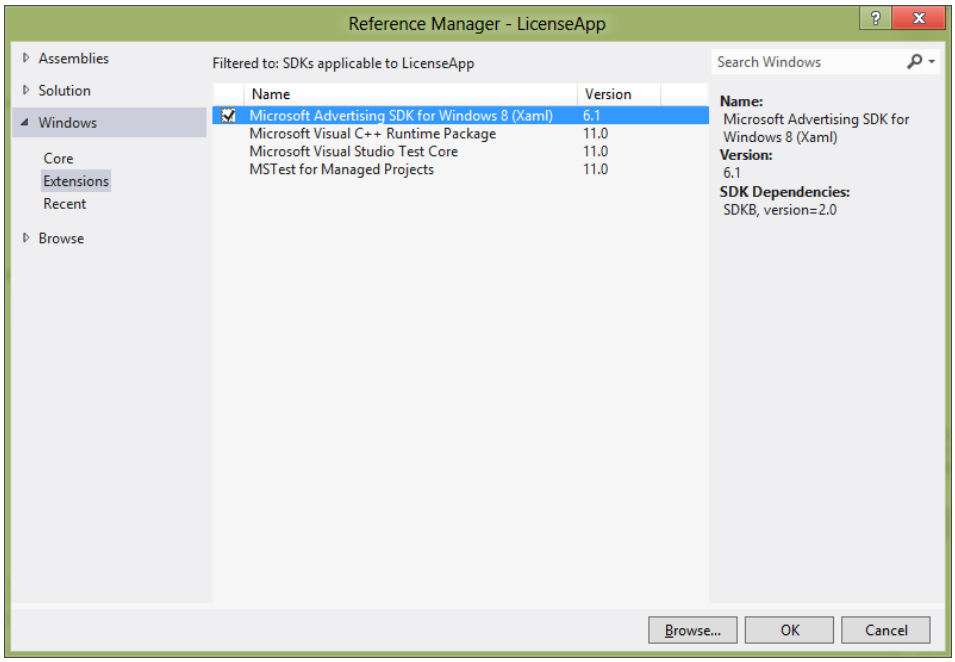
```
ListingInformation appInfo = await CurrentAppSimulator.LoadListingInformationAsync();
string price = appInfo.FormattedPrice;
string description = appInfo.Description;
string currentMarket = appInfo.CurrentMarket;
string name = appInfo.Name;
uint ageRating = appInfo.AgeRating;
IReadOnlyDictionary<string, ProductListing> products = appInfo.ProductListings;
```

Értelemszerűen, ha **`CurrentApp`**-ot használunk, akkor a Windows Store-ból, míg ha **`CurrentAppSimulatort`** használunk, akkor a helyi gép **`WindowsStoreProxy.xml`** fájlból kérdezi le a kód az információt.

Reklámok beágyazása

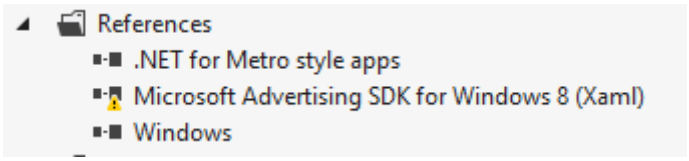
Pénzt nemcsak az alkalmazások eladásával tudunk keresni, hanem akár reklámokból is. Ehhez használhatjuk a Microsoft Advcenterét, vagy ha van saját szolgáltatásunk, akkor akár azt is. Az AdvCentert azért érdemes használni, mert a fejlesztők számára egy SDK-t biztosít, amivel egyszerűen jeleníthetik meg a reklámjaikat. Ebben a fejezetrészben az Advcenter beágyazását mutatjuk be.

1. Töltsük le és telepítsük fel az AdCenter SDK-t (http://go.microsoft.com/?linkid=9800248)!
2. Ha eddig még nem volt PubCenter-en regisztrációnk, akkor regisztráljunk a https://pubcenter.microsoft.com/ linken! (A könyv írásakor ez a szolgáltatás még Magyarországról nem volt elérhető, de az ígéretek szerint hamarosan elérhetővé válik.)
3. Ahhoz, hogy használni is tudjuk az AdCentert, vegyük fel a projekt referenciái közé a Microsoft Advertising SDK for Windows 8 (Xaml) szerelvényt (13-12 ábra)! Mint látható, jelenleg csak XAML vezérlő létezik. A JavaScriptes megoldás a későbbiek folyamán fog érkezni.



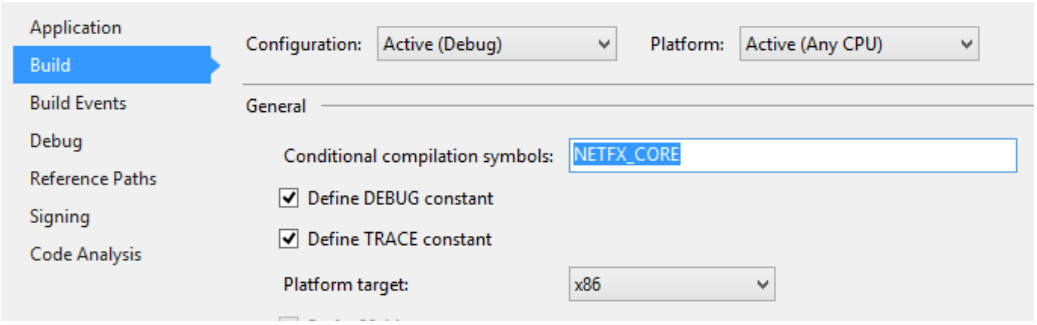
13-12 ábra: Az Advertising SDK meghivatkozása

4. A jelenlegi SDK csak x86 módban tud fordulni, tehát abban az esetben, ha Any CPU, x64 vagy ARM konfiguráció van a projektben beállítva, ha nem, akkor ezt a hibát jelzi a fordítóprogram és a Visual Studio is (13-13 ábra).



13-13 ábra: Az assembly referencia egy figyelmeztetéssel jelenik meg

5. Ezt a problémát úgy küszöbölhetjük ki, hogy a projekten jobb egérgombbal a Properties ablakban a jobb oldali sávra kattintva a Build kategóriát kiválasztjuk, majd a Platform target tulajdonságot **x86**-ra állítjuk (13-14 ábra). Ezt követően már a fordítás is működni fog. Az SDK későbbi változataiban várhatóan ez a működési mód majd megváltozik.



13-14 ábra: Az x86-os célarchitektúra kiválasztása

Figyelem! Ahhoz, hogy az ***`AdControl`*** *vezérlőt használhassuk, a Capabilities elemek között az Internet (Client) opciónak szerepelnie kell! Ellenkező esetben a vezérlő nem tudná letölteni a hirdetéseket.*

6. A kiválasztott oldalhoz adjuk hozzá vagy oldalak XML névterei közé vegyük fel a **`Microsoft.Advertising.WinRT.UI`** névteret!

```
xmlns:adv="using:Microsoft.Advertising.WinRT.UI"
```

7. Ezt követően XAML-ből már elérhetjük az **`AdControl`** vezérlőt.

```
<adv:AdControl x:Name="AdControl1" ApplicationId="YOUR_APPLICATION_ID" AdUnitId="YOUR_AD_UNIT_ID"
Width="250" Height="250" />
```

Itt az **`ApplicationId`** az alkalmazás PubCenterben megadott egyedi azonosítója. Az **`AdUnitId`** érték pedig meghatározza, hogy milyen reklámot szeretnénk megjeleníteni. A 13-3 táblázat a jelenleg támogatott reklámokat mutatja be.

13-3 táblázat: Az AdCenter által jelenleg támogatott hirdetések

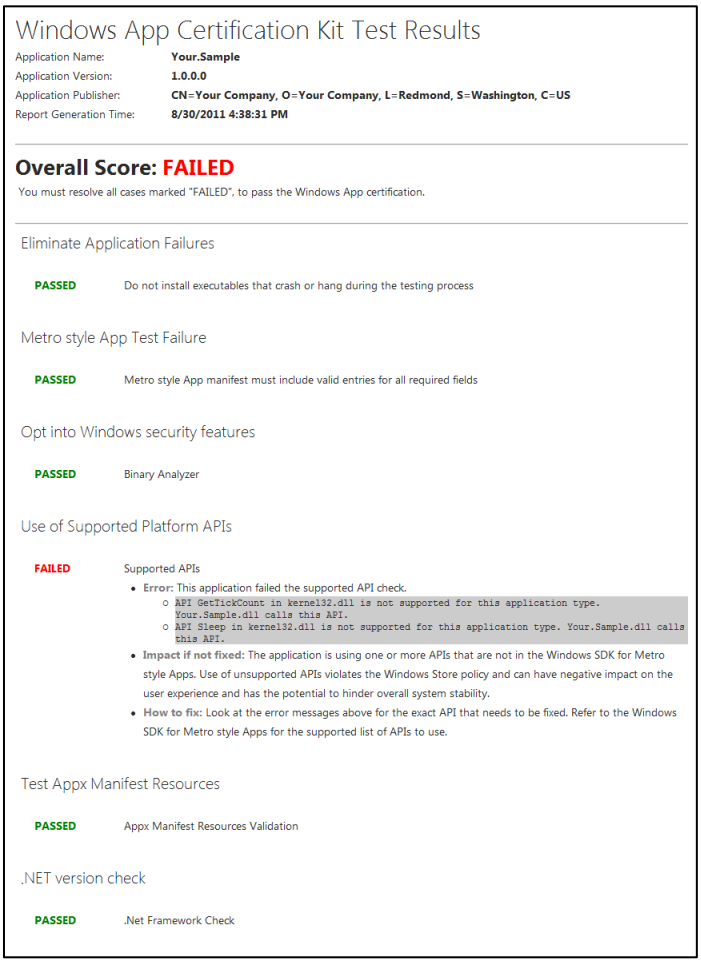
AdUnitID	Méret	Típus	Interakció
`Image_160x600`	160 x 600	Image	Böngészés
`Image_250x250`	250 x 250	Image	Teljes képernyős kép
`Image_300x250`	300 x 250	Image	Teljes képernyős video
`Image_300x600`	300 x 600	Image	Böngészés
`Image_320x480`	320 x 480	Image	Böngészés
`Image_320x50`	320 x 50	Image	Böngészés
`Image_728x90`	728 x 90	Image	Böngészés
`ImageText_160x120`	160 x 120	Image + Text	Böngészés
`ImageText_160x160`	160 x 160	Image + Text	Böngészés
`ImageText_250x250`	250 x 250	Image + Text	Böngészés
`ImageText_320x50`	320 x 50	Image + Text	Böngészés
`Text_300x250`	300 x 250	Text	Teljes képernyős video
`Text_320x250`	320 x 250	Text	Teljes képernyős video
`Text_320x50`	320 x 50	Text	Böngészés
`Video_300x250`	300 x 250	Video	Beágyazott video
`Video_320x480`	320 x 480	Video	Beágyazott video

Láthatjuk, hogy nagyon egyszerűen ágyazhatunk be reklámokat az alkalmazásunkba, és akár csak önmagukban a reklámokkal is kereshetünk pénzt. Persze nem kötelező a Microsoft Advertisingot használni. Használhatunk saját szolgáltatót is, ott viszont a reklámok megjelenítése ránk hárul, amennyiben az adott szolgáltatónak nincs ahhoz megfelelő vezérlője. A Microsoft szolgáltatása ilyen téren kényelmes, hisz néhány kattintás alatt beilleszthetjük a reklámokat az alkalmazásunkba.

Windows App Certification Kit

Ha az alkalmazásunkat elkészítettük, a Windows Store-ba való feltöltése előtt célszerű lefuttatni néhány egyszerű automatizált tesztet. Ehhez a Microsoft a fejlesztők számára egy alkalmazáscsomagot,a Windows App Certification Kit-et biztosítja. Több olyan potenciális problémát is kiszűr, amelyek miatt a Microsoft mérnökei visszautasíthatják a Windows Store-ba ellenőrzésre feltöltött alkalmazást. Ilyen például annak ellenőrzése, hogy az alkalmazásunk fut-e gyengébb számítógépen is, és ott a megfelelő teljesítményt mutatja-e, azaz elindul-e 5 másodperc alatt, és elalszik-e 2 másodpercen belül. Azt is ellenőrzi, hogy csak Windows Runtime API-kat használ-e az alkalmazás. Az alkalmazás ellenőrzését az alábbi lépésekkel végezhetjük el:

1. Indítsuk el a Windows App Cert Kit-et! Az alkalmazás alapértelmezett útvonala: **`C:\Program Files\Windows Kits\8.0\App Certification Kit`**. Itt mind grafikus felületet használó alkalmazást (**`appcertui.exe`**), mind parancssorosat (**`appcert.exe`**) találhatunk. Az alkalmazás elindításához adminisztrációs jogra van szükség!
2. A megjelenő ablakban válasszuk ki a Validate a Windows Store App menüpontot! Ezt követően az alkalmazás kilistázza az összes telepített Windows 8 stílusú alkalmazást.
3. Válasszuk ki a listából azt az alkalmazást, amit tesztelni szeretnénk, majd kattintsunk a Next gombra!
4. A teszt néhány percig eltarthat. Közben az App Cert Kit elindítja majd az alkalmazást, és be is zárja a teszt közben. Hagyjuk futni, és ne zavarjuk a működését!
5. Miután a teszt befejeződött, az eszköz egy HTML riportot készít számunkra (13-15 ábra). Ha minden rendben van, akkor feltölthetjük az alkalmazást a Windows Store-ba. Ha viszont csak egyetlen esetben is „FAIL” lett a teszt eredménye, azt ki kell javítanunk. Ez azt jelenti, hogy a Windows Store-ba küldés után mindenképpen visszadobnák az alkalmazásunkat.



13-15 ábra: A Windows App Cert Kit tesztek eredménye

> ***Fontos**! Ha a Windows App Cert Kit eredménye jó, attól az alkalmazást még visszautasíthatják a Microsoft mérnökei a tesztjeik során! Itt csak az alapvető dolgokat ellenőrzi a segédeszköz. Asztali alkalmazásokat is a Windows App Cert Kit-tel kell validálni!*

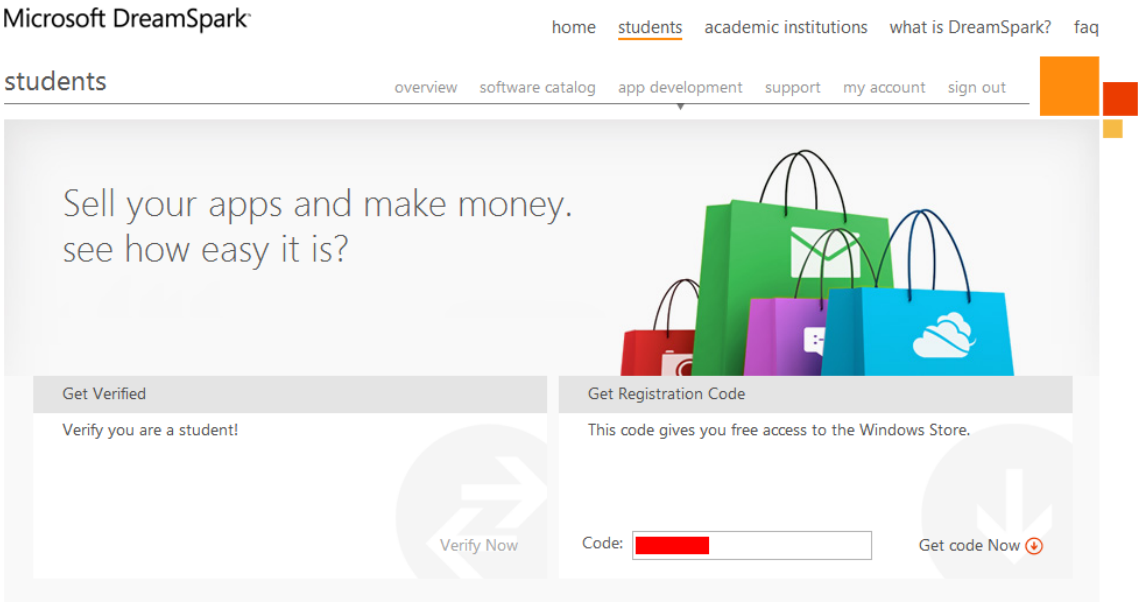
Alkalmazás feltöltése a Windows Store-ba

A folyamat legutolsó lépése az alkalmazás publikálása a Windows Store-ba. Mire is van ehhez szükségünk? Kifejlesztettük az alkalmazásunkat, amelyet alaposan le is teszteltünk – szem előtt tartva a Microsoft Windows 8 stílusú alkalmazásokkal szemben támasztott követelményeit. A Visual Studio 2012 eszközeit

segítségül híva, a minőségbiztosítási folyamat ránk háruló részét is elvégeztük, a Windows Certification Kit pontos képet adott az alkalmazás állapotáról. Nincs más hátra, mint beküldeni alkalmazásunkat hitelesítésre, és utána elérhetővé tenni a felhasználók számára is!

Ahhoz, hogy a Windows Store-ba publikálhass, regisztrált Windows 8 fejlesztőnek kell lenned. Látogass el a https://appdev.microsoft.com/StorePortals/ oldalra, majd a Microsoft Account azonosító segítségével lépj be! (Elképzelhető, hogy SMS-ben vagy e-mailben küldött kóddal hitelesítened kell magad, de ettől ne ijedj meg !)

A fejlesztői státusznak éves díja van, amely a könyv írásának pillanatában 99 USD egy teljes évre. Fontos azonban megjegyezni, hogy ha rendelkezel MSDN előfizetéssel, vagy a diákoknak szóló **`Dreamspark.com`** oldalon érvényes regisztrációd van, akkor ezt a díjat nem kell kifizetned, az első év teljes mértékben ingyenes lesz. Nincs más dolgod, mint MSDN vagy Dreamspark előfizetésedbe belépni, majd egy hozzáférési kódot igényelni, ahogyan az a 13-16 ábrán is látható.



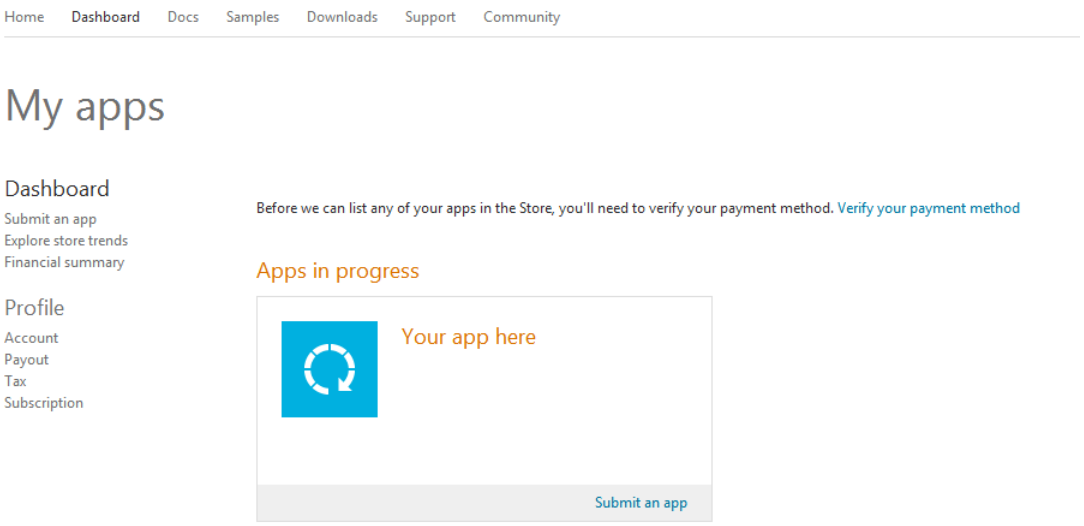
13-16 ábra: A Dreamspark program által a diákoknak ingyenes a regisztráció

A regisztráció lépéseit nem ismertetjük részletesen, mivel a folyamat nagyon jól dokumentálja magát. Arra azonban mindenképpen oda kell figyelned, hogy cég vagy magánszemély regisztrációját szeretnéd-e végrehajtani, ugyanis a regisztráció utáni hitelesítési folyamat a két esetben különbözik!

Ha a regisztrációval végeztél, a fenti menüsoron válaszd a Dashboard gombot! Arra kattintva az a felület fogad téged, ahol az új alkalmazást tudod majd publikálni, illetve lekérdezni és módosítani egy korábban már publikált alkalmazásod jellemzőit. Itt is van lehetőséged a profilod részleteinek megadására is, illetve a bankszámla-információk felvitelére, amelyek segítségével a fizetős alkalmazásaid ellenértékéhez juthatsz hozzá. Amíg nincs számlainformáció megadva, addig nincsen lehetőséged fizetős alkalmazás publikálására! Az oldal továbbá tájékoztat az aktuális egyenlegedről is, illetve a folyamatban lévő kifizetések státuszáról (13-17 ábra).

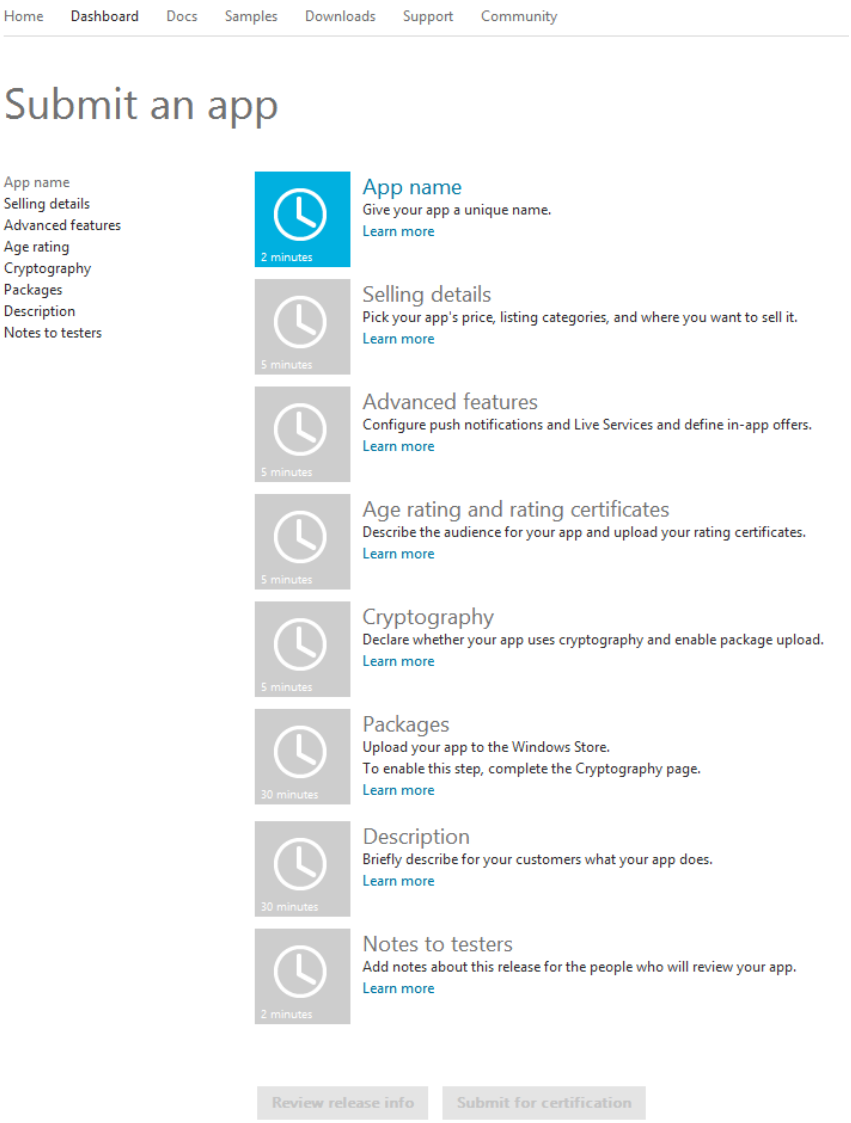
Miután nincs még feltöltve a programod, kezdd is el a publikálást a Submit an app gombra kattintva (13-17 ábra)! Az ezután megjelenő felületen lesz lehetőséged az új alkalmazás minden egyes jellemzőjét megadni és testre szabni (13-18 ábra).

A folyamat a program nevének megadásával kezdődik. Miután már a fejlesztés során is szükséges biztosan tudnunk, hogy a publikálás pillanatában nem lesz-e foglalt a választott név, így a név megadása önmagában előjegyzést is jelent. Ha lefoglaltunk egy nevet, azt nem használhatja már senki más akkor sem, ha még nincs publikálva a programunk. Ez a lépés akár Visual Studio-ból is elvégezhető a Store menüpont segítségével.



13-17 ábra: A Dashboard felülete (még nem volt alkalmazás publikálva)

Az összes lépést nem ismertetjük teljes terjedelmében, azonban a 13-4 táblázat összefoglalja azokat a jellemzőket, amelyek megadására az esetek jelentős részében szükségünk van.



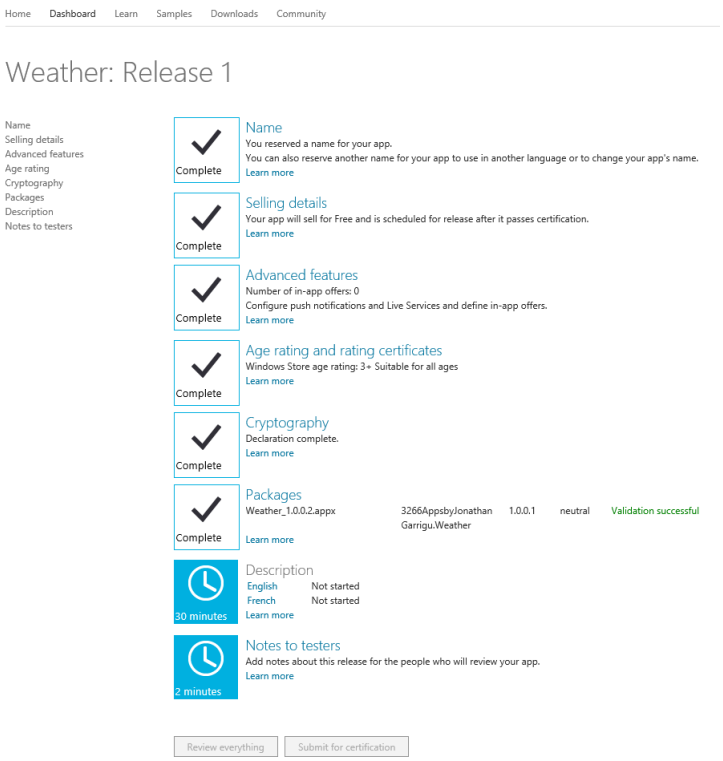
13-18 ábra: Az alkalmazás publikálásához szükséges információk

13-4 táblázat: Az alkalmazás publikálásához szükséges információk listája

Kategória	**Kitöltendő mező neve**	**Tartalma**
Name	App name	Az alkalmazás neve, maximálisan 256 karakter hosszú lehet.
Selling details	Price tier	
	Free trial period	Próbaidőszak támogatása
	Countries/regions	Mely országok piacterén legyen elérhető az alkalmazás?
	Release date	Kiadási dátum
	Category Subcategory	Alkalmazáskategória és alkategória. Pl. Játékok → Akciójátékok.
	Accessible app	Támogat-e korlátozott lehetőségekkel rendelkező felhasználókat is az alkalmazás?
	Minimum DirectX feature level	Minimális DirectX követelmények deklarálása
	Minimum System RAM	Az alkalmazás által igényelt minimális memóriamennyiség
Advanced features	In-app offers	Alkalmazáson belüli promóciók és kiegészítő szolgáltatások támogatása
Ratings	Age rating	Korhatár
	Rating certificates	Tanúsítvány a korhatárnak való megfelelésről
Cryptography		Használ-e a program különböző kódolásokat is?
Packages	Package upload control	A Visual Studio 2012-ben elkészített alkalmazáscsomag elérési útja
Description (*minden támogatott nyelvhez külön meg kell majd adni!*)	Description	Az alkalmazás leírása, maximálisan 10.000 karakter hosszan megfogalmazva
	App features	Alkalmazásfunkciók felsorolása (maximum 20 db)
	Keywords	Az alkalmazásra jellemző kulcsszavak megadása (maximum 7 db)
	Description of update	Ha frissítést adtunk ki, akkor a javítások és új funkciók leírása
	Copyright and trademark info	Szerzői jogok felsorolása
	Additional license terms	A jogi információk részletes leírása (például felhasználói nyilatkozat)
	Screenshots	1366×766 pixeles méretű képernyőképek az alkalmazásból,

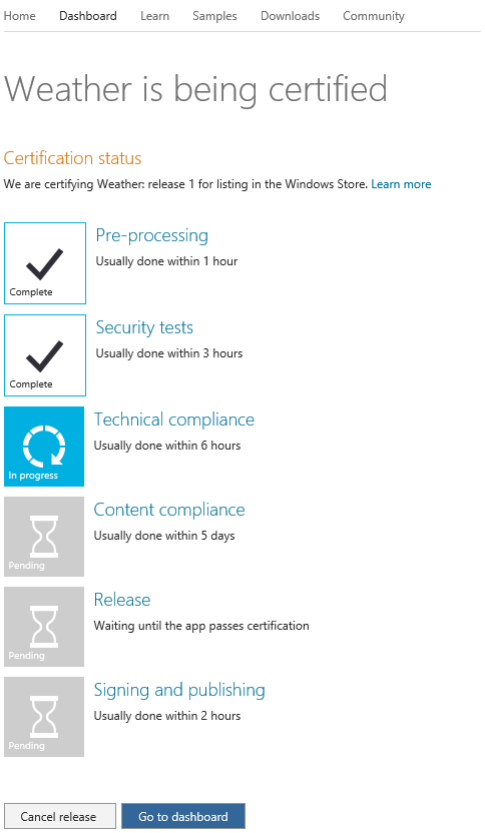
Kategória	Kitöltendő mező neve	Tartalma
		maximum 200 karakternyi leírással. Összesen 8 darabot adhatunk meg.
	Promotional images	Az alábbi méretezések valamelyikét használó promóciós képek: 414x180, 414x468, 558x756, 846x468
	Recommended hardware	Leírás az alkalmazás hardverkövetelményeiről, maximálisan 11 db.
	App website	Az alkalmazás weboldala
	Support contact	Ha támogatásra van szüksége a felhasználónak, akkor ezt az elérhetőséget kell használnia.
	Privacy policy	Adatkezelési nyilatkozat
	In-app offer description	Alkalmazáson belüli kiegészítő szolgáltatások leírása
Notes to testers	Notes	Megjegyzés a Microsoft tesztelői számára, akik az alkalmazást hitelesíteni fogják.

Miután készen vagy az összes mező kitöltésével, nézd át még egyszer a bevitt adatokat, erre az oldal is figyelmeztetni fog téged! Ha úgy gondolod, hogy minden készen áll a publikálásra, akkor nincs más hátra, mint beküldened az alkalmazáscsomagot (13-19 ábra).



13-19 ábra: Az alkalmazás készen áll a feltöltésre

A folyamat innentől fogva már nem lesz automatikus, mivel a Microsoft szakemberei manuálisan tesztelik az alkalmazásodat az MSDN-en megtalálható és a jelenlegi fejezetben is leírt szempontok alapján, illetve számos más minőségbiztosítási elvet is használnak (13-20 ábra). Ha minden rendben van, akkor egy-két héten belül értesítenek a programod elfogadásáról és azt publikálhatod a Windows Store-ban.



13-20 ábra: Az alkalmazás elfogadása folyamatban van

Összegzés

Ebben a fejezetben részletesen végignéztük a Windows 8 alkalmazás-értékesítési modelljét. A felhasználói nézőpontokat szem előtt tartva indultunk el, megismertük a Store alkalmazás működését és viselkedését. Átnéztük azokat a részleteket, amelyeket saját alkalmazásunkban is meg kell valósítani ahhoz, hogy a Windows Store-ba publikálhassuk.

Az üzleti modellt és a technikai oldalt összekapcsolják a Windows Store fejlesztői komponensei, amelyek működésére láthattunk néhány egyszerű kódpéldát. Ezek segítségével az alkalmazáson belül lekérdezhetünk piactér információkat, és ezek segítik a próbaváltozatú alkalmazásverzió létrehozását is.

Természetesen számos új elvárásnak kell megfelelnünk, így a minőségbiztosítási szempontok és az alkalmazástesztelés mechanizmusát is részben át kell alakítanunk. Megnéztük, milyen lépések szükségesek a programok megfelelő minőségű kivitelezéséhez és a lehetséges hibák számának minimálisra csökkentéséhez.

Megismerhetted, hogyan válhat belőled regisztrált Windows 8 fejlesztő, aki az elkészített alkalmazásait valóban képes publikálni a piactérre. Publikálás után pedig nincs más dolgunk, mint várakozni az alkalmazás elfogadására és a Windows Store-ban való megjelenésére.