Szakdolgozat

Fésüs Miklós

Debrecen

2011

Debreceni Egyetem
Informatika Kar

Fejlesztés Windows Phone 7 platformon

CellPark Alkalmazás kifejlesztése

Témavezető:

Dr.Juhász István

Egyetemi adjunktus

Készítette:

Fésüs Miklós

programtervező informatikus MSc

Debrecen

2011

Bevezetés

Jelen dokumentum célja, hogy betekintést nyújtson a windows phone 7 alkalmazások fejlesztésébe egy konkrét alkalmazás kifejlesztésén keresztül.

A tárgyalt alkalmazás a Cellum Innovációs és Szolgáltató Zrt. CellPark nevű alkalmazása, mely a budapesti mobil parkolást hívatott megkönnyíteni az okostelefont használók számára. Az alkalmazás rendelkezésre áll az összes mai korszerű okostelefonra, így Iphone és Android rendszerekre is. Az alkalmazás fejlesztésénél fontos elv volt, hogy az adott platformnak megfelelő stílusban készüljenek el az alkalmazások, így a megvalósítások mind dizájnban, mind – bár kevésbé – logikában eltérnek.

Windows Phone 7

Specifikáció

A windows phone 7 operációs rendszer a Microsoft legújabb mobil operációs rendszere. Silverlight 3-ban a Microsoft Compact Framework 3.5 és C# 4.0 áll a rendelkezésünkre a fejlesztésre. C# 4.0-ból egyedül a dynamic kulcsszó nem használható, hiszen ehhez a Dynamic Runtime Library –re volna szükség, ami nem része a Microsoft Compact Framework 3.5-nek.

Megjegyzés: A jövőbeli frissítések után, körülbelül 2011. szeptembertől már Silverlight 4 platformon lehet majd programozni az operációs rendszert.

Silverlight alkalmazásokon kívül futtathatunk rajta XNA Game Studio 4.0 segítségével készült játékokat is.

Hardver

Az operációs rendszerhez a Microsoft által meghatározott minimális hardver igencsak erős, a mai csúcskategóriába számító hardverspecifikációkat határozta meg a Microsoft, ezzel

elkerülendő az alkalmazások esetleges hibás, lassú futást, mint Android esetén, melyet gyengébb hardverképességekkel rendelkező telefonra is engedélyez a Google.

A képernyőfelbontás rögzített, 800x480 pixel, melyből bizonyos elemek elfoglalnak néhány pixelt, mint például az Application Bar, vagy a System Tray, ezeket azonban nem kötelező használni alkalmazásainkban.

Mivel a felbontás rögzített minden jelenlegi és jövőbeli készüléken, a fejlesztőknek sokkal könnyebb dolguk van, elég egyetlen design-t elkészíteni, nem kell foglalkozni a különböző felbontásokból adódó nehézségekkel.

Szenzorok

A telefon tartalmaz minden modern szenzort, ami a mai modern okos telefonokban megjelenhet: gyorsulásmérő, GPS, fénymérő, közelségmérő, iránytű, kamera (egyenlőre csak hátlapi, újabb telefonoktól lesz előlapi is videó telefonáláshoz), flashlight. Még nem mindegyikhez engedélyez hozzáférést az operációsrendszer, illetve például kamerához és kontaktokhoz csak egy túlzottan szabályozott hozzáférés van.

Megjegyzés: A következő frissítés tartalmaz majd a szenzorokhoz közel 1500 új API-t.

Tombstoning

Az operációs rendszer jelenlegi verziója nem támogatja a multitaskingot, helyette egy igen kényelmetlen megoldás a Tombstoning az alternatíva. Eszerint az alkalmazásnak van egy Deactivated és Activeted eseménye, melyekben az alkalmazásnak el kell menteni, illetve vissza kell állítania az alkalmazás aktuális állapotát, mivel amint az alkalmazást „letesszük tálcára” az operációs rendszer meghívja a Deactivated eseményt, és lefutása után felszabadítja az alkalmazás által felhasznált memóriát.

Az alkalmazás indításakor lefut egy Launching esemény, itt az alkalmazás-specifikus folyamatokat futtathatjuk, valamint az alkalmazás bezárásakor lefut a Closing esemény, ahol perzisztálhatjuk adatainkat.

Az adatok perzisztálására azonban nem csupán a Closing eseményben van szükség, hiszen az alkalmazást letehetjük tálcára, és menüből indíthatunk egy új alkalmazáspéldányt, s ebben

az esetben az előző példányunk megsemmisül, s nem mentett adataink elvesznek, így a Closing eseményben történő adat mentést érdemes a Deactivated eseményben is megtenni.

Megjegyzés: A multitasking a következő nagy verziófrissítés után lesz elérhető, mint alternatíva.

IsolatedStorage

Az operációs rendszer korlátozza továbbá a fájlrendszerhez történő hozzáférést, csak úgy, mint Webes Silverlight esetében, csupán az alkalmazásunk számára létrehozott, a fájlrendszer többi részétől elkülönített, úgynevezett IsolatedStorage áll rendelkezésünkre. Ezen a tárolón belül szabadon hozhatunk létre, vagy törölhetünk fájlokat, illetve mappákat, valamint rendelkezésünkre áll egy speciális ApplicationSettings tároló, melybe objektumokat helyezhetünk el, és ezeket az operációs rendszer DataContractSerializer segítségével perzisztálja, ezért fontos az, hogy a behelyezett objektumaink megfeleljenek a DataContractSerializer által megszabott feltételeknek. A DataContractSerializer eredetileg lehetővé teszi a nem publikus tagok szerializálását és deszerializálását is, de mivel a windows phone 7 alkalmazások a desktop alkalmazásokkal ellentétben nem Full Trust-ban, hanem csupán Partial Trust módban futnak, így ez a lehetőség nem elérhető.

Megjegyzés: A következő frissítéstől a fejlesztők rendelkezésére fog állni egy SQL Server Compact Edition is.

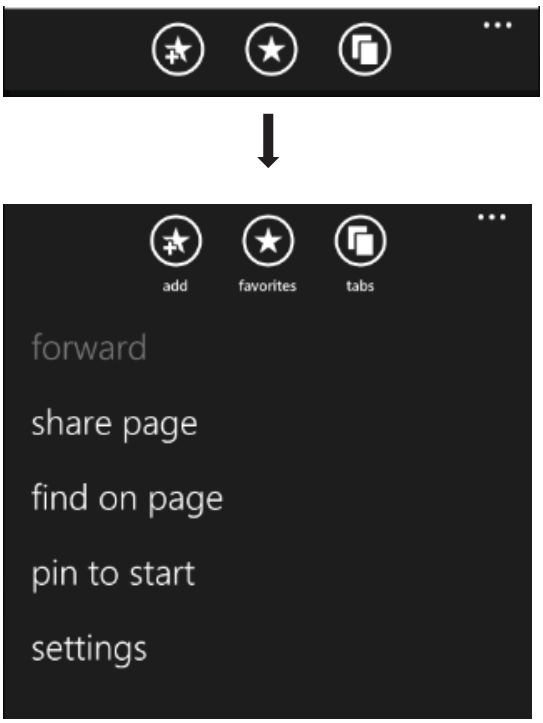
Általános alkalmazás felépítése

Egy windows phone 7 alkalmazás, csak úgy, mint egy Silverlight alkalmazás lapokból áll, ezek az úgynevezett PhoneApplicationPage-ek.

Ezek egy keretbe vannak foglalva, melyből alkalmazásonként egy van, ez a PhoneApplicationFrame. A lapok között navigálhatunk oda vissza, ugyanúgy, mintha egy böngészőben weboldalakat nézegetnénk.

Egy PhoneApplicationPage tartalmazza a különböző felhasználói felületi elemeket, vezérlőket.

Minden lapon rendelkezésünkre áll egy speciális vezérlő, az ApplicaitonBar, ami tulajdonképpen az adott laphoz tartozó legfontosabb parancsokat tartalmazza,



Az application bar tartalmazhat maximum 4 darab menügombot, megfelelő képpel és szöveggel, valamint tartalmazhat menüelemeket, melyekből nyitott állapotban egyszerre maximálisan 5 látható. Ennél több esetén egy görgethető menülista jelenik meg.

Minden egyes lapon külön-külön megadhatjuk hogy látható legyen e a System Tray, azaz a tálca, mely a térerőt, hálózati kapcsolatokat stb jeleníti meg.

A lapok háromféle orientációt ismernek, ezek a PortraitUp, a LandscapeLeft és a LandscapeRight. Nem kötelező mindet támogatni, ha igen, akkor figyelnünk kell rá hogy dizájnunk mindhárom elforgatás esetén megfelelő legyen, vagy pedig külön dizájnt készítünk az álló és fekvő esetekre.

Az alkalmazás elkészítése során törekedni fogunk rá, hogy támogassuk az elforgatott nézeteket is, ezzel növelve az ügyfélelégedettséget.

Alapvetően az elforgatásra egy esemény segítségével tudunk reagálni. Azonban ehhez arra volna szükség, hogy a változtató kódot teljesen code-behind-ban írjuk meg, vagy pedig XAML-ben írjuk Storyboard-ok segítségével, de akkor is manuálisan kellene lefuttatnunk, sőt semmilyen design time támogatásunk nem lenne.

Ezért felhasználjuk a Behaviour-ok segítségét. Definiálunk egy OrientationToStateBehaviourt, melynek feladata, hogy az esemény bekövetkezésekor az az oldalunkat egy, az oldalon definiált állapotba átvigye.

Ezekután nincs más dolgunk mint definiálni a különböző állapotokat, mint Portrait vagy Landscape, és innentől akár Expression Blend segítségével is rögzíthetjük az aktuális állapotnak megfelelő UI elrendezést, és a Blend saját maga fogja legenerálni számunkra az átalakításhoz szükséges animációkat.

Ezeket az animációkat pedig nem kell kézzel meghívnunk, hiszen a behaviour megteszi ezt helyettünk.

CellPark

Budapest több, kisebb-nagyobb parkoló zónára van osztva, melyeket más-más cégek kapnak meg fenntartásra, és pénzbeszedésre. Minden egyes parkoló zónának van egy úgynevezett zónakódja. Az autósok fizetség ellenében parkolhatnak, ehhez csupán meg kell keresniük a legközelebbi parkoló automatát (totem) mely esetenként nincs túl közel, rosszabb esetben pedig nem működik. Az autósok parkolhatnak még telefonjuk segítségével is, ehhez nincs más dolguk, mint egy megadott számra egy sms-t küldeni, melyben feltüntetik autójuk rendszámát.

Egy telefonszámról egyszerre egy érvényes parkolást lehet indítani, egy másik indítása törli az előzőt és újat indít.

A mobil parkolás hátránya, hogy kényelmi díj, azaz parkolásonként X forint többletköltséget jelent, viszont megannyi előnye mellett ez az X összeg sokszor eltörpül.

Előnyei:

- Ha a parkoló automata elromlott, akkor is a rajta található telefonszámon mobil parkolást még tudunk indítani.
- Ha nincs nálunk aprópénz, nem kell felváltatnunk, mobil egyenlegünk terhére azonnal parkolhatunk.
- Ha nem akarunk elmenni a legközelebbi parkolóóráig, viszont tudjuk mely zónában vagyunk (pl.: gyakran járunk ott, vagy útközben láttuk az automatát és megjegyeztük) akkor mobilról elindítjuk a parkolást és már mehetünk is.
- A mobilparkolásról minden esetben naplóbejegyzés készül a mobil parkolási rendszerben, és így utólagos esetekben, jogtalan büntetés kiszabás esetén könnyen ellenőrizhető, hogy volt e érvényes parkolása járművünknek avagy sem.

Alapvetően azonban a mobil parkolás még mindig nem a legkényelmesebb forma, hiszen:

- tudnunk kell a zónakódot, vagy el kell sétálnunk a legközelebbi automatához.
- minden alkalommal meg kell írni az sms-t és elküldeni a megfelelő telefonszámra

Erre a célra találta ki a Cellum Zrt. a CellPay alkalmazást. Az alkalmazás segítségével pillanatok alatt küldhetjük el a megfelelő sms-t a megfelelő számra. Az alkalmazás képes több rendszámot tárolni, és választhatunk belőlük, eltárolja a kedvenc zónakódjainkat, valamint GPS segítségével képes körülbelül meghatározni mely zónában is lehetünk. Amennyiben több zóna is szóba jöhet, a felhasználó a megjelenített térkép segítségével ki tudja választani, mely zónában is tartózkodik. Ezután az alkalmazás összeállítja az SMS-t és már parkolhatunk is.

Az ügyfeleknek lehetőségük van kétféle parkolásra, az egyik esetben meghatározott mennyiségű időt választanak, és ez az sms-be is bekerül.

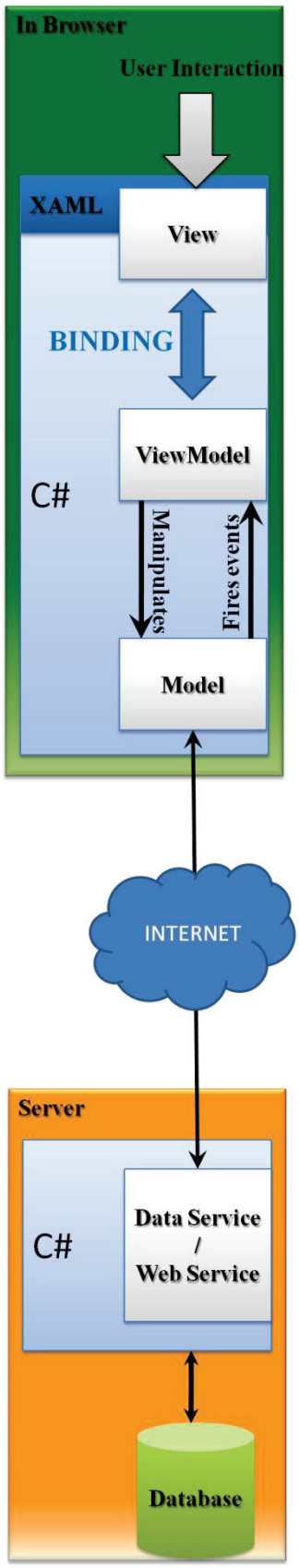
A másik eshetőség, hogy egyszerűen start-stop típusú parkolást indítanak, mely esetben az aktuális zóna maximális parkolási idejének összege előre lefoglalódik az egyenlegen, és amikor az ügyfél a parkolást leállítja, akkor az addig eltelt idő függvényében a nem elköltött egyenleg feloldásra kerül.

MVVM

Az MVVM (Model – View – ViewModel) programtervezési minta a Microsoft Patterns and Practises által kitalált alkalmazás-architektúra a XAML alapú alkalmazásokhoz.

A Model tartalmazza az adatréteget, a View a megjelenítés, azaz a felhasználói felület (UI).

A ViewModel feladata tulajdonképpen a kettő összekötése anélkül, hogy a kettőt szorosan összekapcsolná. A cél a lazán csatolt (loosely – coupled) architektúra. Ahhoz hogy ezt elérjük, a ViewModelünket mint adatkontextus-t átadjuk a View-nak, és ő a számára szükséges adatokat egy vagy kétirányú adatkötés segítségével beköti a



Model-View-ViewModel

vezérlőkbe.

Ezáltal nincs szükség arra, hogy a ViewModel ismerje a View-t, hogy frissítse, esetleg módosítsa azt. Így egy ViewModel-hez tulajdonképpen korlátlan számú View-t csatolhatunk, különböző céllal, vagy különböző megjelenéssel.

View

A View-t tisztán, vagy majdnem tisztán XAML segítésével implementáljuk, a mögöttes ún. Code-behind fájlba csak a legszükségesebb, UI specifikus kódot, vagy jobb esetben semmit nem írunk. A UI események kezelését Commanding segítségével oldjuk meg. A Commanding lényege, hogy a UI vezérlőknek átadhatunk egy ICommand interfészt implementáló parancs objektumot, melyet az esemény lefutásakor ő végrehajt.

Így a parancsobjektumunkat definiálhatjuk a ViewModel-ben, ezáltal szétválasztva a designt a mögöttes alkalmazáslogikától.

A ViewModel szétválasztásával nemcsak a függetlenséget nyerjük, hanem a könyebb tesztelhetőséget is. Van rá ugyanis lehetőség, hogy a Command objektumaink parancsait (metódusait) külön felhasználói interakció nélkül is kiváltsuk, ezáltal könnyebben tesztelhetjük a ViewModelünket.

Ezzel a szétválasztással az alkalmazásunkon egyszerre dolgozhat mind a fejlesztő, mind pedig a designer. A designer az Expression Blend segítségével könnyen átalakíthatja a megjelenítést, anélkül, hogy ez bármilyen hatással volna a mögöttes logikára, megváltoztathatja a UI vezérlők nevét, kicserélheti őket, hiszen egyetlen eseménykezelő sem köti őket a logikához.

Amint kicserél egy vezérlőt, csupán annyit kell tennie, hogy a megfelelő eseményhez rendeli a megfelelő Command objektumot.

ViewModel

A ViewModel-ben kezeljük az alatta található adatréteget. Ez a réteg felelős a View-k számára szükséges adatok kiolvasásáért, a módosítások végrehajtásáért.

A ViewModel implementálhatja az INofifyPropertyChanged interfészt, melynek egyetlen eseménye van a PropertyChanged, ami egy PropertyName paramétert vár. Amennyiben valamelyik property megváltozik a ViewModel-en akkor ahhoz hogy ez frissüljön a View-ban, szükség van rá hogy valamilyen módon értesítsük. Adatkötés-kor a View feliratkozik az eseményre, ha az rendelkezésre áll. Ha a View vezérlő számára kötött property nevével váltjuk ki az eseményt, akkor az adott vezérlő tudni fogja, hogy az aktuális értéke már nem érvényes, és újra lekérdezi a ViewModel-től a kötött értéket, majd frissül.

A Microsoft külön ilyen céllal implementált egy kollekciót, mely módosításakor kiváltja ezt az eseményt. Mivel a .NET kollekciói nem rendelkeznek álllapotjelző (Added, Removed, stb…) eseménnyel, így ezek esetén minden egyes módosulást figyelni kellene, és minden esetben kiváltani a megfelelő eseményt.

Ezt elkerülendő használható az ObservableCollection<T> generikus kollekció, ahol a T a generikus típusparaméter.

Model

A modellünk lehet tulajdonképpeni adatbázis, vagy egy adatbázishívásokat végző osztály, valamint akár egy Web-en keresztüli adatforrást is elérhetünk például Windows Communication Foundation segítségével, vagy http kérésekkel.

A modellünket is külön tesztelhetjük, a modell nem tud a ViewModel-ről, egy modellhez több ViewModell is tartozhat.

MVVM Light

Az alkalmazásban az MVVM light nevű Laurent Bugnion Silverlight MVP által kifejlesztett és fejlesztés alatt álló MVVM framework-ot használjuk. Ez a framework jelentősen megkönnyíti a ViewModel-ekkel történő munkát. Definiál egy Messaging framework-ot, melynek segítségével lehetőség nyílik a ViewModel-ek közötti lazán csatolt kommunikációra, az értékek módosulásának nem csak a View-kal hanem más ViewModelekkel való közlésére is. Minden ViewModel egy ViewModelBase osztályból származik, mely néhány plusz tulajdonságot ad. Mindenek előtt ad egy alapvető Messenger objektumot, melynek segítségével különböző típusú üzeneteket küldhetünk más ViewModel-eknek. Ezen üzenetek

lehetnek tájékoztató jellegűek, visszahívó jellegűek is. Adhatunk át értéket is rajtuk keresztül, vagy jelezhetjük egy érték megváltozását.

Implementálja az INotifyPropertyChanged interfészt, valamint ad egy többszörösen túlterhelt RaiseNotifyPropertyChanged metódust, mely a PropertyChanged eseményt váltja ki. A metódusnak van egy érdekes túlterhelése, melyben egy lamda kifejezéssel adhatjuk meg a property-t, ezzel kivédve az esetleges névelgépelést, és hosszú órákig tartó debuggolást.

Megjegyzés: Ez utóbbi esetben a nevet egy hosszabb műveletsor segítségével, egy kifejezésfával csomagolja ki a rendszer, mely időben többe kerül, mintha csak stringként adnánk át, viszont érdemesebb mégis ezt használni, hiszen a property átnevezésekor is a helyes értéket fogja adni.

Van még ezen túl egy IsInDesignMode propertyje melynek felhasználásával design módban más adatokat adhatunk meg, s megint mást éles futáskor. Ennek szerepe az, hogy az adatok már design módban is megjelennek mind visual studio-ban mind pedig Expression Blend-ben, és így a designer láthatja azokat, amikor alakítja a vezérlőket, hogy miként is fog kinézni például egy lista eleme. A Blend ugyan nyújt megoldást design data létrehozására, de nem támogatja a XAML a generikus osztályokat, így ezek esetében ez a property nagyon hasznos.

A Framework ad előre módosított Visual Studio template-eket is, csak úgy mint ViewModel létrehozására szolgáló code snippeteket, melyek jelentősen rövidíthetik le munkánkat.

Az MVVM Light egy általános MVVM keretrendszer mind WPF, SIlverlight és Windows Phone 7 alkalmazásokhoz, viszont számunkra egy Windows Phone 7 specifikus megközelítés kell, így egy kicsit átalakítjuk a ViewModel-t valamint a ViewModel-eket tartalmazó központi osztályt a ViewModelLocatort.

ViewModelLocatorBase

A ViewModelLocatorunkat ezen túl egy ViewModelLocatorBase osztályból származtatjuk, és ebben az osztályban végezzük el az alkalmazás egyik legfontosabb műveletének automatizálását, ez pedig az adatok perzisztációja és visszatöltése.

Ennek megvalósítására Reflection-t fogunk használni. A ViewModelLocator-unk tartalmazza az összes ViewModel-ünket, így tehát hozzáférése van az összes alkalmazással kapcsolatos adatunkhoz. Az aktuális ViewModelLocator-unkról reflekció segítségével lekérhetjük az összes publikus ViewModelünket, majd ezeket DataContractSerializer segítségével egy fájlba írhatjuk, ahonnan az alkalmazás újbóli futásakor kiolvassuk.

A ViewModelLocator-ban a ViewModel-eink létrehozásakor szeretnénk egy statikus idejű típusellenőrzött betöltést, ezért olyan betöltő metódust hozunk létre, melyben átadjuk a ViewModel-ünk típusát. Ehhez egy generikus metódust hozunk létre, melynek szignatúrája a következő:

```
protected static T LoadViewModel<T>(Expression<Func<T>> viewModelNameExpression)
```

Azáltal hogy egy Expression-t várunk, lehetőségünk van rá, hogy a már említett módon kifejezésből vegyük ki a ViewModel nevét az alábbi függvénnyel:

```
private static string ExtractPropertyName<T>(Expression<Func<T>> propertyExpresssi
on)
```

Ezután írhatjuk azt a ViewModelLocator-ban hogy:

```
mainVM = LoadViewModel<MainViewModel>(() => MainVM);
```

Az adatok perzisztálásához pedig feliratkozunk a két alkalmazás specifikus eseményre:

```
PhoneApplicationService.Current.Deactivated += new EventHandler<DeactivatedEventAr
gs>(this.Current_Deactivated);
            PhoneApplicationService.Current.Closing += new EventHandler<ClosingEve
ntArgs>(this.Current_Closing);
```

Vannak azonban olyan ViewModel-ek is amelyek értékeit csak az alkalmazás futása alatt szeretnénk megőrizni, tehát az újbóli futásnál nem, mint esetünkben az új rendszám létrehozására szolgáló NewRegPlateViewModel. Ezen ViewModel-ek értékeit csupán a Deactivated eseménynél kellene perzisztálni.

Vannak továbbá olyan ViewModel-ek is amelyeknek értékeit egyáltalán nem szeretnénk, vagy nem szükséges elmenteni, ezért olyan megoldást választunk, hogy ezen ViewModelek között különbséget tudjon tenni a program, ez pedig az öröklődés.

ViewModelBase

Az MVVM Light-ban a ViewModel-eink tehát egy ViewModelBase osztályból származnak. Ahhoz hogy az automatikus perzisztálásunk működjön, ebből az osztályból származtatunk egy TemporaryViewModelBase-t, melyből örökölt ViewModel osztályainkat csak a Deactivated eseményben perzisztáljuk.

A TemporaryViewModelBase osztályból származtatunk továbbá egy PersistentViewModelBase osztályt, melyből örökölt ViewModel osztályaink értékeit a Closing eseménynél is szeretnénk eltárolni, és az alkalmazás újraindítását követően újra betölteni.

Ebben az osztályban definiálunk egy IsFirstRun nevű propertyt, melnyek segítségével az IsDesignMode propertyhez hasonlóan, meg tudunk adni olyan kódrészleteket, melyek az alkalmazás telepítése utáni első futtatáskor futnak le.

BindeableApplicationBar

A windows phone 7 alkalmazásokban található Application Bar sajnálatos módon nem egy Silverlight vezérlő, hanem az operációs rendszerhez köthető natív vezérlő.

XAML-ben nem is a fő vezérlő része, hanem a shell névtérben található és külön megadható.

Sok hátránya van, elsők között hogy nem lehet code-behind-ból rá hivatkozni, csak ha az egészet code-behind-ból építjük meg, másodsorban nem lehet neki Command objektumot, vagy EventToCommand Behaviour-t definiálni, tehát az eseménykezelőt csak és kizárólag code-behind-ban írhatnánk meg, és nem volna rá lehetőség, hogy MVVM módon a ViewModelben definiáljuk a lefuttatandó kódrészletet.

Továbbá mivel nem lehet hozzá adatkötni semmit, így a lokalizáció sem megoldható XAML-ből. A megoldáshoz az egész Application Bar-t code-behind-ból kellene megépíteni, de ez viszont elég kényelmetlen.

Létezik egy megoldás, ami tulajdonképpen az eredeti ApplicationBar vezérlő wrappelése egy silverlight vezérlőbe, s melynek neve BindableApplicationBar.

Ehhez a következőkre van szükség. Kell egy osztály, ami implementálja az IApplicationBar interfészt, és annak minden metódusában tulajdonképpen egy eredeti ApplicationBar

objektumot kezel, módosít, frissít. Ezáltal például a Text property esetén, annak módosulásakor át tudjuk adni az eredeti ApplicationBar menu vagy button vezérlőnek az új text-et, és ehhez nem kell írnunk az alkalmazásban egy sor kódot sem.

Az egyetlen hátulütője a megoldásnak, hogy elveszítjük a design time támogatást, azaz saját kezűleg kell beállítanunk a megfelelő ikonokat.

Ezzel a megoldással viszont kiegészíthetjük funkcionalitással a már meglévő application bar-t, például adhatunk neki egy IsItemsEnabled tulajdonságot, melyet adatköthetünk, és például egy listában egy bizonyos elemet kiválasztva engedélyezhetjük vagy letilthatjuk az egész application bar-t nem csak egyesével az elemeit.

Implementálhatunk olyan gombot is melynek van egy köthető NavigateUri propertyje, és amennyiben rákattintunk, azonnal navigál minket a kívánt oldalra.

Megjegyzés: Ez a funkció NavigateCommand behaviour segítségével is elérhető.

Amennyiben ezt a megoldást választjuk, akkor az ApplicationBar-t most már a vizuális fa részeként kell megadnunk, lehetőleg nulla mérettel, hogy ne takarjon el semmit, valamint a vizuális fa végére érdemes definiálni.

Alkalmazásfelépítés

ViewModellek

MainViewModel

A MainViewModel-ünk az alkalmazás lelke, ő egy perzisztens ViewModel, tehát adatait kilépés után is tároljuk. Tartalmazza a rendszámok listáját, csak úgy mint a kedvenc zónák listáját is.

Itt van ezentúl még a választott rendszám indexe, az esetleges beállított fix idő amennyiben fix időre parkolunk, a telefonszolgáltató kódja, az aktuális zónakód.

Megjelenik továbbá két Command objektum is, az egyik a start, a másik stop parancs. Ezeket a parancsokat majd a MainPage oldalunk fogja adatkötni az alkalmazás menüjében. Mindkét parancs egy – egy metódushívást takar.

A RegPlates és a FavoirteZones egy – egy ObservableCollection. Mivel ezek változó kollekciók, így azáltal hogy ez a típusuk, a változások automatikusan továbbítódnak a megfelelő megjelenítő vezérlők számára is.

A ViewModel-ek közötti messaging szolgáltatás egy üzenetére iratkozunk fel a RegisterForMessages metódusban, mégpedig arra az esetre, hogy ha egy NewRegPlateViewModel-től egy új rendszámot kapnánk:

```
this.MessengerInstance.Register<GenericMessage<RegPlateItem>>(this, (message) => t
his.AddRegPlate(message.Content));
```

NewRegPlateViewModel

A rendszámok létrehozásához egy külön ViewModel-t hozunk létre.

Mivel ezen műveletsor egyszeri, így nincs szükség arra hogy adatait tároljuk az alkalmazásból történő kilépés után is, viszont mivel több műveletből áll egy rendszám létrehozása, esetenként több lapon át, így az értékeket mindenképp tálolnunk kell az alkalmazás tálcára történő letétele során. Ebből az okból ez a ViewModel a TemporaryViewModelBase osztályból származik.

Tartalmaz egy rendszám, egy országkód és egy parkolási típus propertyt, melyeket a megfelelő vezérlőkhöz kötünk majd a megjelenítéskor.

Tartalmaz továbbá három Command objektumot, a hozzáadást a rendszámokhoz, az országkód kiválasztását, valamint a parkolási típus kiválasztását.

Amikor a rendszám létrehozásának művelete véget ér, értesítenünk kell a MainViewModel-t arról, hogy van egy új rendszám is a rendszerben, és a megfelelő vezérlőket frissítse.

Ezt a messenger szolgáltatással tesszük meg a következő módon.

```
this.MessengerInstance.Send<GenericMessage<RegPlateItem>, MainViewModel>(new Gener
icMessage<RegPlateItem>(new RegPlateItem(this.RegPlate, this.CountryCode, this.Par
kingType)));
```

A MainViewModel esetében erre a típusú üzenetre iratkoztunk fel.

SettingsViewModel

A beállításokhoz tartozó ViewModel egy perzisztens ViewModel. Ebben tároljuk a rendelkezésre álló nyelveket, valamint azt is hogy ezek közül jelenleg melyik nyelv van kiválasztva.

Egy listában eltároljuk az elérhető szolgáltatókat névvel, és szolgáltató kóddal együtt és itt tároljuk továbbá a választott mobilszolgáltató kódját is.

Erre azért van szükség, mert a platform jelenleg nem támogatja ennek az adatnak a lekérdezését, és a telefonszámból pedig a számhordozás miatt nem derül ki egyértelműen.

Az adatra szükség van az sms küldésekor, mivel minden zónának három telefonszáma van, minden szolgáltatóhoz egy-egy. Amennyiben a felhasználó nem saját szolgáltatóján keresztül küldi be az sms-t, akkor az nem jut el a megfelelő csatornára, és nem lesz érvényes parkolása.

Lokalizáció

Az cég alapvető elvárása, hogy alkalmazásai minél egyszerűbben lokalizálhatóak legyenek. A .NET framework-ben erre a célra a Resource (resx) fájlok állnak rendelkezésre, melyekben kulcs-érték párokat adhatunk meg, ahol a kulcs egy azonosító, az érték pedig tetszőleges string. Ezen kívül lehetőség van még a fordítóknak szóló megjegyzés beírására is az egyes értékekhez.

Ezekből a fájlokból a fejlesztő környezet egy osztályt generál, melyben az azonosítók nevével statikus property-ket hoz létre, mely az aktuális szál aktuális nyelvének megfelelő fájlból adja vissza az értéket, vagy a default fájlból, ha nem található a nyelvhez saját resource fájl.

Silverlight-ban XAML-ben csak és kizárólag publikus propertyket tudunk adatkötni. A Visual Studio resx generátorában van azonban egy apró hiba (vagy szándékosan így van) miszerint is amennyiben egy resx fájl láthatóságát publikusra állítjuk, az osztály konstruktora továbbra

is internal marad, ezáltal nem tudjuk XAML-ből példányosítani. Bár lehetőség van rá, hogy ezt kézzel átírjuk, ám az osztály minden egyes módosítás alkalmával újragenerálódik, és ez fejlesztés közben rendkívül bosszantó.

A megoldás az hogy körüljárjuk a problémát más szemszögből. Létrehozunk egy külön osztályt, melynek neve esetünkben LocalizedStrings lesz, és melynek egy darab statikus propertyje van, melynek neve Resources. Az osztály példányosításakor a Resources értéke a Resources fájlunk (amelyben a sztringeket tároljuk) generált osztályának egy példánya lesz.

Ezekután már képesek leszünk hozzáadni az App.xaml fájlhoz globális resource-ként a LocalizedStrings osztályt, és ezt hivatkozva tudjuk a megfelelő lokalizált értékeket adatkötni.

Az adatkötés tehát így néz ki:

```
<TextBlock Text="{Binding Resources.newParking, Source={StaticResource LS}}"/>
```

Ha minden egyes alkalommal átírnánk a generált osztály konstruktorát publikusra, akkor így nézne ki:

```
<TextBlock Text="{Binding newParking, Source={StaticResource Resources}}"/>
```

Az alkalmazásunk tehát innentől könnyedén lokalizálható. Amilyen nyelven használja a felhasználó az operációs rendszert, azon a nyelven fog megjelenni az alkalmazásunk.

Számunkra ez viszont nem kielégítő, mivel a windows phone 7 operációs rendszer egyenlőre csupán 5 nyelven elérhető, mi pedig a magyar közönséget célozzuk meg elsősorban alkalmazásunkkal, valamint előfordulhat, hogy valaki ugyan angolul használja a telefonját, alkalmazásait szeretné magyar köntösben látni.

Erre az esetre szükségünk van arra, hogy a nyelvet megváltoztassuk. Ez relatíve könnyen megy, hiszen csupán át kell állítanunk az aktuális szál kultúráját magyar kultúrára. Ezzel azonban a szövegeink nem fognak frissülni.

Ha az aktuális kultúrát megjegyezzük, és alkalmazásindításkor azt állítjuk be, akkor a megfelelő nyelven fognak megjelenni a szövegek, azonban ehhez újra kellene indítani az alkalmazást.

Van egy egyszerű és kézenfekvő megoldás. A szövegeink adatkötve vannak, méghozzá a Resources property egy más propertyjéhez. Ha azt mondjuk az alkalmazásnak, hogy megváltoztak, akkor újra fogja kötni minden egyes UI vezérlő. Ráadásul nincs szükség arra sem hogy egyesével küldjönk egy PropertyChanged eseményt, elég csupán azt mondani hogy a Resources property megváltozott. Ehhez létrehozunk egy metódust:

```
public void ResetResources()
{
    if (this.PropertyChanged != null)
    {
        this.PropertyChanged(this, new PropertyChangedEventArgs("Resources"));
    }
}
```

Ha a beállításokban meghívjuk ezt a metódust, akkor az összes oldal újra be fogja tölteni a megfelelő szöveget a vezérlőibe.

```
Thread.CurrentThread.CurrentUICulture = new System.Globalization.CultureInfo(((App
licationLanguage)this.Languages[this.SelectedLanguageIndex]).Culture);

((LocalizedStrings)App.Current.Resources["LS"]).ResetResources();
```

CountryViewModel

A CountryViewModel a CountryCodeSelectorPage-hez tartozó ViewModel.

Az oldalon megjeleítjük az összes elérhető ország nevét és kódját, melyek közül választhatjuk ki a rendszám országkódját.

Ezeket a neveket és kódokat, csak úgy, mint a lokalizáció esetén, egy resource fájlban tároljuk. A példányosításkor ezt a resource fájlt megnyitjuk, és felolvassuk a tartalmát, majd az Items kollekcióban eltároljuk.

Mivel ezek az adatok statikusak, és mindegy hogy ebből a fájlból, vagy az alkalmazáshoz rendelt IsolatedStorage-ből olvassuk ki, így helytakarékosságból az előző megoldást választjuk, hiszen utóbbi esetben kétszer tárolnánk ugyanazt az adathalmazt.

Így nincs szükségünk az automatizált perzisztálási mechanizmusunkra sem, így a ViewModelünket közvetlenül a ViewModelBase-ből származtatjuk.

View-k

LoopingListSelector

A Silverlight Toolkit for Windows Phone –ban található egy primitív vezérlő, amely az operációs rendszerben is található datepicker és timepicker alapja. Lényege tulajdonképpen egy téglalap alapú vezérlő, mely egy potenciálisan végtelen körkörös lista. Alapvetően nincs definiálva hozzá egyszerű adatforrás, csupán egy ILoopingListDataSource interfészt kapunk, melynek két metódusa és egy propertyje van, sorra a GetPrevious(object relativeTo), GetNext(object relativeTo) és a SelectedItem.

A lista tulajdonképpen a cirkuláris lista vizuális megfelelője. Meg kell adnunk a méretét, valamint az elemek méretét. Ezek után kiszámolja azt, hogy a lista elemszámából és az elemek méretéből miként tudja megalkotni a vizuális listát. Ha 6 elem magas és 3 elemből áll, akkor kétszer helyezi el a listát a vizuális fában és görgetésnél az elemeket újrapozicionálja, ezáltal körkörös érzést biztosítva.

Ehhez a vezérlőhöz van szükségünk egy olyan adatforrásra, melyben int-eket lehet átadni, erre szükség van mind a „Zóna”, mind a „Fix idő” vezérlők esetében. A későbbi újrafelhasználás érdekében egy generikus LoopingListDataSource<T> -t implementálunk.

A megoldáshoz felhasználjuk a. Net beépített LinkedListNode és List<T> osztályait.

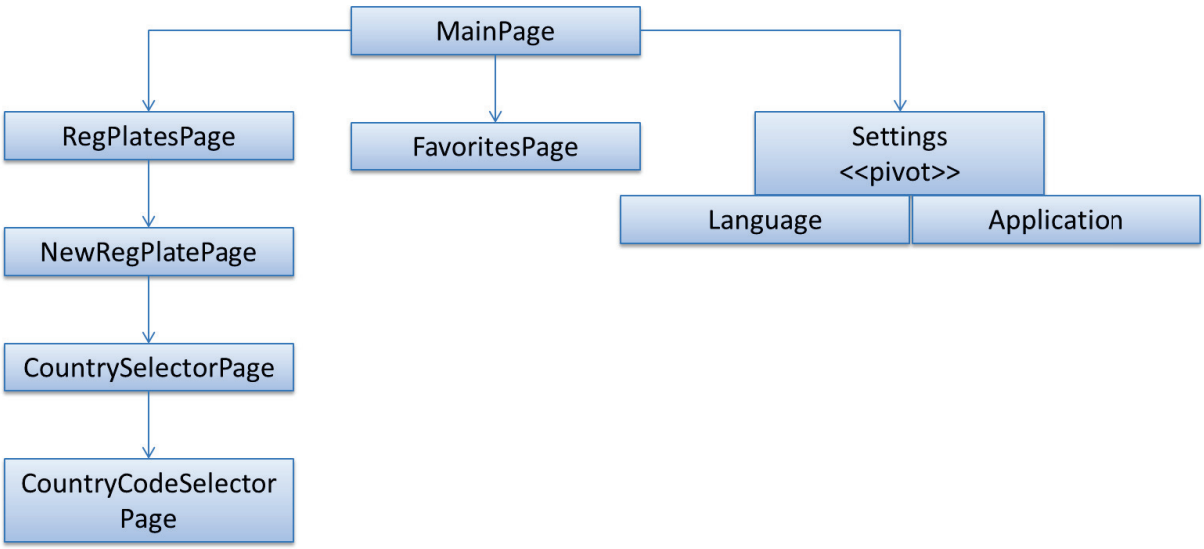
Ahhoz hogy értelme legyen a listának, rendezettnek kell lennie. Az alapvető osztályokhoz biztosítva van egy default Comparer osztály, amivel a rendezést megvalósítja, viszont saját osztályaink nem biztos, hogy implementálják az IComparable interfészt. Használhatunk egy Comparer-t, de minden rendezéshez létrehozni egy új Comparer osztályt elég kényelmetlen. Erre találták ki még .Net 2.0 –ban a Comparison-t, ami egy delegate. Újabban egy ennél is jobb megoldást fogunk használni, mégpedig hogy a Comparer egy Func<T,T,int> lesz.

Amennyiben pedig szükségünk van rá hogy a beépített vagy definiált Comparer<T>-t használjuk azt is megtehetjük egyszerűen:

```
comparer = new Func<T, T, int>(Comparer<T>.Default.Compare);
```

Ahhoz hogy ezt a megoldást használhassuk, és ne kelljen törődni a belső LinkedListNode implementációval, az osztályon belül egy privát NodeComparer osztályunk van, amely az összehasonlítást már LinkedListNode<T> objektumokra végzi, és neki adjuk át tulajdonképpen a funkciónkat.

A Func azért praktikus, mert írhatunk a helyébe inline lambda kifejezést, így nem kell metódusokkal, Comparer osztályokkal, vagy anonymus delegate-ekkel bajlódnunk (mint a Comparison esetében)



MainPage

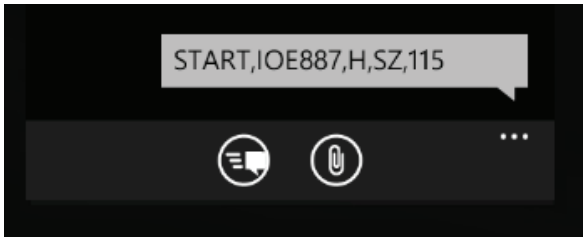
Az alkalmazásunk kezdőoldala a MainPage. Innen tudunk parkolásokat indítani, valamint leállítani, és az összes többi oldalra történő navigáció is innen indul.

Az oldal legfelső vezérlőjén választhatjuk ki az aktuális rendszámot, melyet jobbra balra történő húzással választhatunk.

Az alatta található Zóna kód vezérlő, tulajdonképpen 4 darab LoopingListSelector vezérlő. A mellette található csillag gombbal jutunk el a kedvencek oldalra. Az alatta lévő gombbal tudjuk az aktuálisan kiválasztott zónát hozzáadni a kedvencekhez.

A kedvenc zónáinkhoz nevet is rendelhetünk, mivel könnyebb név alapján beazonosítani egy helyszínt mint szám alapján. pl.: 8834 – Munkahely

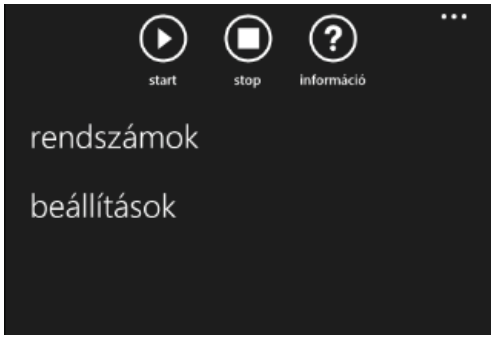
Az Application Bar-on három gomb található, az első a parkolás indítás gomb. Megnyomásakor az alkalmazás a kiválasztott rendszámból, zóna kódból és időből összeállítja a megfelelő parkolási sms-t, amit utána már csak egy gombnyomás elküldeni az alkalmazás által beállított telefonszámra.



A második gomb segítségével a parkolás leállító STOP üzenetű sms-t hozza fel az alkalmazás.

Megjegyzés: A parkolásunkat sms nélkül ingyenesen service kód segítségével is le lehet állítani, azonban a windows phone 7 jelenlegi változatában nincs lehetőség arra, hogy az alkalmazásunkból service kód-ot küldjünk.

Az alkalmazás menüben található továbbá a két navigációs menüelem, melyek segítségével a rendszámok, valamint a beállítások oldalra navigálhatunk.



Az információ gomb segítségével segítséget kaphatunk az adott oldal használatával kapcsolatban.

Amennyiben fix időre szeretnénk parkolni, akkor megnyomjuk a fix idő gombot, és ekkor egy újabb vezérlő jelenik meg a képernyőn.

Ez a vezérlő 3 LoopingListSelector-ból áll.

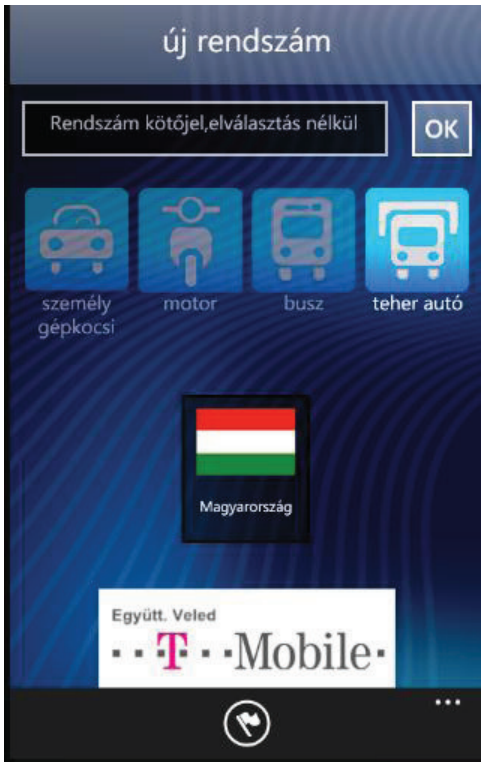
RegPlatesPage

A rendszámokat kezelő oldal egy középre igazított listából áll, mely a kiválasztott elemet mindig középre igazítja. A MainPage oldalon kiválasztott rendszám és az ezen az oldalon kiválasztott rendszám egy és ugyanaz, ha egyik oldalon választunk egyet, a másik oldalon is az lesz a választott.

A kiválasztott rendszámot itt törölhetjük a kuka gombbal, vagy pedig létrehozhatunk új rendszámot a plusz gomb segítségével, melyet megnyomva a NewRegPlatePage-re jutunk.

NewRegPlatePage

Az új rendszám létrehozására szolgáló oldalon megadhatunk egy rendszámot, kiválaszthatjuk a jármű típusát. A motor is megjelenik, mint járműtípus, bár egyelőre motorral parkolni ingyenes. Ez a négy kép tulajdonképpen négy radiobutton egy csoportba foglalva, mind ugyanahhoz a forráshoz adatkötve.



A rendszám beírására szolgáló textbox egy speciális saját fejlesztésű textbox, mely több különböző tulajdonsággal rendelkezik, többek között megadható számára egy vízjel. Van egy property-je mely megadja hogy a beírt szöveget automatikusan kis vagy nagybetűsre konvertálja e.

Van egy propertyje melynek segítségével átadhatunk neki egy karakter halmazt, és egy másik melynek segítségével azt adjuk meg hogy ez a karakterhalmaz legyen az engedélyezett karakterek halmaza, avagy ezen karaktereket nem engedélyezzük a textboxba történő beírás közben.

A zászlóval ellátott vezérlő valamint az alkalmazás menüben található zászlós gomb ugyanoda mutat, mégpedig az országkódválasztó oldalra.

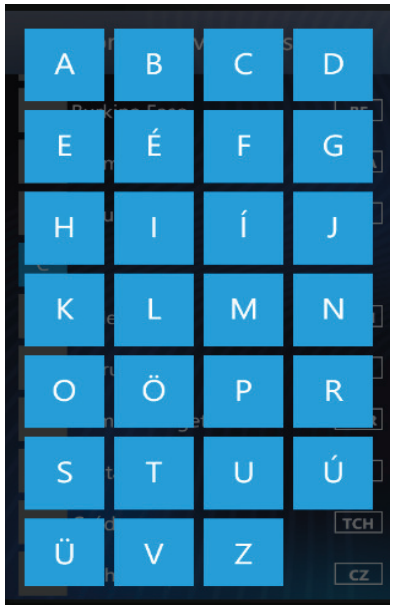
CountrySelectorPage

Az alkalmazás felhasználóinak rendelkeznie kell egy magyar szolgáltatónál lévő telefonszámmal, mivel csak magyar telefonszámról lehet parkolást indítani egyenlőre. A rendszám létrehozás gyorsítását célzandó jött létre ez az oldal, melyen a leggyakrabban előforduló Budapesten tartózkodó járművek felségjelzéseit tartalmazza, a gyorsabb választás érdekében.

Alul az aktuális országkód látható.

Amennyiben más ország felségjelzésével ellátott járművünk van, akkor az alkalmazás menüben a zászló gomb megnyomásával további országok közül választhatunk.

CountryCodeSelectorPage



Ez az oldal egy LongListSelector vezérlőt tartalmaz, mely tipikusan hosszú listákhoz lett kitalálva. A vezérlő teljesen olyan, mint az operációs rendszerben a személyek listájánál található vezérlő. Segítségével gyorsan ugorhatunk egyik csoportról, jelen esetünkben betűről a másikra. A menü segítségével rendezhetjük név illetve országkód sorrendbe is a listánkat.

Transitioning

Ez a vezérlő bár a Silverlight Toolkit for Windows Phone 7 része, ahhoz hogy eredeti hatást nyújtson, szükségünk van egy animációra amikoris egy csoportgomra rányomunk és kinyílik a csoportlista. Az operációs rendszerben ez az animáció egy 3 dimenziós forgatás, és ezt a hatást szeretnénk mi is elérni.

Ehhez rendelkezésünkre áll a GroupViewOpened és a GroupViewClosing esemény, melynél paraméterként megkapjuk az elemekhez tartozó generátort, ami egy ItemContainerGenerator példány.

Ez az osztály arra szolgál, hogy a lista típusú vezérlőknek adatforrásként átadott listák elemeiből megjeleníthető listaelemvezérlőt készísen.

Ebből az osztályból ki lehet nyerni így aztán a listaelemekhez tartozó konténer vezérlőt, és nekünk pontosan erre van szükségünk, mivel ezt szeretnénk animálni.

Erre a célra létezik az úgynevezett Transitioning. Ez azt jelenti, hogy az egyik állapotból a másik állapotba történő eljutást egy Storyboard objektummal adhatjuk meg. Az ITransiton interfésznek van egy GetTransition függvénye, amely egy UIElement vezérlőt vár, és visszaad egy StoryBoard-ot, amely az aktuális vezérlőre alkalmazandó animációt tartalmazza.

Szükségünk van tehát egy ITransition-ra a saját animációnkkal. A jövőbeli újrahasznosítás érdekében egy általánosabb megoldást választunk, mégpedig azt, hogy készítünk egy osztályt, amely bármilyen átadott Storyboard-ből képes transition objektumot csinálni.

Ehhez létrehozzuk a StoryboardTransition nevű osztályt, mely a TransitionElement osztályból származik. Ez az osztály már implementálja az ITransition interfészt, nekünk csupán felül kell írnunk a GetTransition metódust.

Egy Storyboard-ot csupán egy vezérlőre tudunk alkalmazni, ezért nekünk minden vezérlőhöz külön Storyboard-ot kell készítenünk, ami nem triviális feladat, ezért létrehozunk egy IStoryboardCreator interfészt, melynek egy metódusa van méghozzá a CreateStoryboard.

A StoryboardTransition osztályunkhoz hozzáadunk egy csak olvasható IStoryboardCreator propertyt, valamint a konstruktorban lehetőséget adunk rá, hogy megadjunk egy ilyen példányt.

Ezek után az osztály így néz ki:

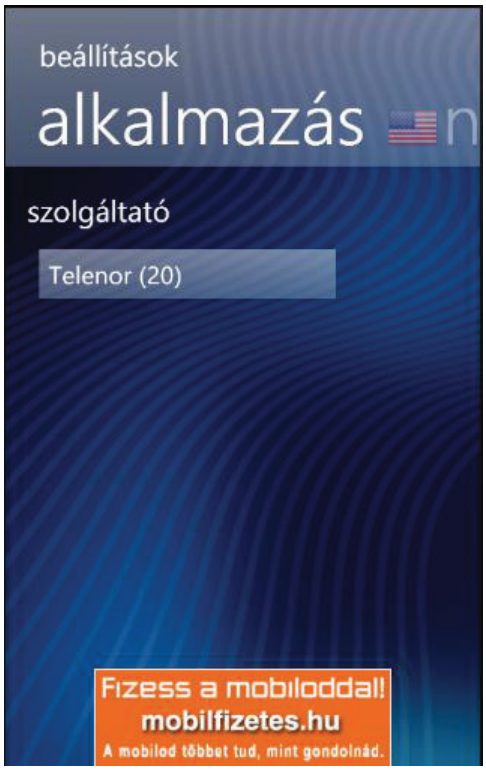
Ezekután szükségünk van egy aktuális példányra az IStoryboardCreator interfészből. Ehhez létrehozzuk az animáció osztályát melnyek neve TursnTileInStoryBoardCreator lesz, mivel az animáció olyan mintha egy csempét fordítanánk magunk fele.

Megírjuk az animációt kódból, vagy ha egyszerűbben megy XAML-ben, akkor megtehetjük ezt is, csupán írjuk meg a kódot XAML-ben, majd XamlReader segítségével felolvastatjuk és átkonvertáljuk Storyboard-á.

Végül nem marad más dolgunk, mint meghívni az eseményben az alábbi kódrészletet:

```
StoryboardTransition mytransition = new StoryboardTransition(new TursnTileInStoryB
oardCreator());
for (int i = 0; i < itemCount; i++)
{
    UIElement element = itemContainerGenerator.ContainerFromIndex(i) as UIElement;
    ITransition animation = mytransition.GetTransition(element);
    animation.Begin();
}
```

SettingsPage



A beállítások oldalunk egy Pivot típusú oldal, mely azt jelenti, hogy oldalirányú oldalakból áll. Ez tipikusan a beállítási oldalakra jellemző oldaltípus, az asztali alkalmazások fül típusú vezérlőjének felel meg.

Az első beállítási oldalon az alkalmazás – specifikus beállításokat adhatjuk meg, jelen esetünkben a felhasználó mobilszolgáltatóját.

A második beállítási oldalunk a nyelvvel kapcsolatos.

CellAd

Ahhoz hogy az ingyenes alkalmazásból a parkolás kényelmi díján kívül egy kis bevétel legyen, reklámokat szeretnénk megjeleníteni az alkalmazásunkban.

Jövőtervezés szempontjából fontosnak találták azt is, hogy amennyiben a reklámozás beválik saját alkalmazásainkban, akkor más fejlesztők számára is elérhetővé kell tenni saját alkalmazásaikban.

A megoldás tehát egy alkalmazástól független vezérlő kifejlesztése. A vezérlő neve CellAd lett. A feladata nem bonyolult, egy web serverhez intéz http kéréseket és ezek alapján reklámokat jelenít meg az alkalmazás felületén, 300x100 pixel méretű képekben.

Az első verzióban a vezérlő minden alkalommal lekérte a szervertől a megjelenítendő reklám ID-t és a reklám távoli elérési útját, majd letöltötte és megjelenítette azt. Ezzel az volt a probléma, hogy minden egyes alkalommal adatforgalmat generált, valamint megakasztotta az alkalmazás futását egy pillanatra.

A második verzióban a vezérlő indításkor lekérte a reklámok listáját, majd a még nem letöltötteket letöltötte és utána már a mentett képeket jelenítette meg. Ezzel az volt a probléma, hogy az alkalmazás indítás idejét növelte meg, különösen az első online indítását.

A harmadik verzióban a vezérlő indításkor lekérte aszinkron módon az a reklámok listáját, majd a még nem letöltötteket letöltötte és utána már a mentett képeket jelenítette meg. Ezzel az volt a probléma, hogy csupán az összes reklám letöltése után jelent meg a legelső reklám az alkalmazásban. Újratervezés után a vezérlő:

Adott egy reklám objektum melnyek propertyjei:

- ID
- URL
- IsSaved
- FileUri
- Version
- NavigateUri

Működés:

1. Lekéri a szervertől a legutolsó reklámadatbázis-frissítés időpontját.
2. Ha ez újabb, mint a tárolt érték akkor
    a. Lekéri a reklámok listáját, mely tartalmazza az ID-ket az URL-eket és a verziószámot.
    b. Ha egy régebbi reklám már nem szerepel akkor törli azt, és a hozzátartozó képet amennyiben már letöltötte.
    c. Ha egy új reklám van akkor hozzáadja a listához és az IsSaved property-t false-ra állítja.
    d. Ha egy reklám verziója újabb, akkor az IsSaved property-t false-ra állítja.
3. Ha nem újabb, vagy megtörtént a frissítés, akkor
    a. Lekéri az aktuálisan megjelenítendő reklám ID-t.
    b. Amennyiben a reklám már mentve van, akkor kiveszi a képet as IsolatedStorage-ból, és megjeleníti azt.
    c. Amennyiben nincs letöltve, akkor aszinkron letölti, elmenti a képet az IsolatedStorage-ba, majd a reklám IsSaved propertyjét true-ra állítja, és végül megjeleníti a képet, így legközelebb már nem kell majd letöltenie.

Ezzel a megoldással amennyiben az első reklám képe még nem mentett, akkor is csupán 1-2 másodperc után megjelenik, amennyiben pedig mentve volt akkor szinte azonnal. Minden egyes reklám akkor töltődik le amikor először szükség van rá.

A letöltés aszinkron módon történik, a letöltési idő alatt az előző reklám képe látszik még 1-2 másodpercig, az alkalmazás reszponzív marad a letöltés alatt is.

Mivel alkalmazásunk több oldalból is áll vagy állhat, és ezek bármelyikén vagy mindegyikén lehet vagy szeretnénk reklámfelületet elhelyezni, ezért szükséges egy központi reklámkezelő létrehozása.

Ezt a példányt az alkalmazás tárolójában is érdemes, sőt szükséges eltárolni, hiszen nem akarjuk hogy minden egyes indításkor újra letöltse az összes reklámot.

A központi reklámkezelő neve CellAdManager. Ahhoz hogy ebből csupán egy legyen, egy Singleton szál-biztos osztályt hozunk létre, azaz egyszerre csak egy példány létezhet belőle.

Ahhoz hogy ne kelljen plusz kódot írnia más fejlesztőknek akik esetleg használják majd a vezérlőt, és elég legyen csupán a vizuális fába behelyezni a vezérlőt a vizuális fába szükség van arra, szükség van arra hogy valahogy el lehessen érni ezt a manager példányt. A megoldás az, hogy az alkalmazás erőforrásai közzé tesszük.

Amikor egy vezérlő lekéri a példányt, akkor az osztály megvizsgálja először is, hogy az erőforrások között megtalálható e már a példány. Ha nem akkor megvizsgálja, hogy az alkalmazás tárolójába el van e mentve a példány. Ha nem akkor létrehoz egy új példányt, elmenti a tárolóba, hozzáadja az erőforrásokhoz és végül visszaadja az új példányt.

Az alkalmazás nem minden oldalán van szükség reklámfelületre, ezért kell az, hogy a reklámkezelő csak akkor kérdezze le a reklámokat, ha van rá „igény”, azaz van olyan CellAd vezérlő, amely látható.

A reklámvezérlők fel kell hogy iratkozzanak nála, és csak abban az esetben kell hogy fusson amennyiben a feliratkozások száma nagyobb mint 0.

A reklámvezérlők továbbá csak akkor jelenítsenek meg reklámot, ha már a kezelő lekérte az első megjelenő reklámot, ezért fel kell iratkozniuk az Initialized eseményre, amely akkor fut le, amikor a dátum és a reklámlista lekérése végbement.

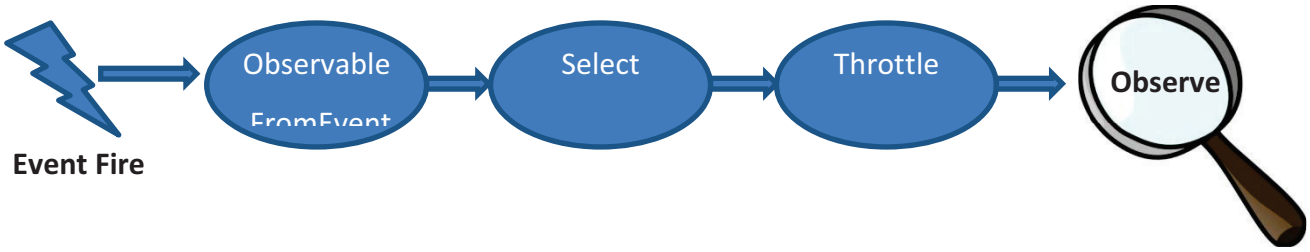
Miután ez megtörtént azonnal le is kell iratkozniuk az eseményről, mert visszalapozásnál már kétszer vagy többször is fel lennének iratkozva az eseményre és többször futna le a kód aminek csak egyszer lenne szabad.

A sok aszinkron kérés és eseménykezelés miatt keletkező bonyolult kód helyett fogjuk használni a Reactive Extensions for .Net framework -öt

Reactive Extensions

A Reactive Extensions a .Net eseménykezelését hívatott újszerű megközelítésben kezelni, és ez a megközelítés az esemény-alapú programozás (Event-driven programming). Az események helyett Observable kollekciókkal dolgozunk, és a kollekciókon végrehajtott

műveletek is Observable kollekciókat adnak vissza, így téve lehetővé hogy egy- egy eseményen több különböző művelet végrehajtását kössük sorba kössük, mintegy csővezetékszerűen végrehajtva műveleteket rajta.



Az Observable.FromEvent segítségével lehetőségünk van egy eseményből egy Observable kollekció létrehozására. A kollekción különböző transzformációs és szelekciós, vagy csoportosító műveleteket hajthatunk végre, mielőtt „megfigyelésre kerül”. Van egy külön Do metódus, ami nem csinál a kollekcióval semmit, csupán mellékhatás végrehajtására használható.

A kollekcióra meghívunk egy Observert. Az observer paramétere egy IObservable<T>, valamint egy Action, mely egy T értéket vár. Az Action-ben definiált függvény fog lefutni a kollekció minden értékére, méghozzá aszinkron módon. Mivel egy esemény aszinkron, a megfigyelése is aszinkron.

Például egy TextBox TextChanged eseménye felhasználói interakciótól függően fog bekövetkezni, ezt nem is lehetne szinkron módon kezelni.

Van olyan függvény is az Rx-ben amely szinkron fut le.

Amire nekünk szükségünk van az Rx-ből, az a tulajdonsága, hogy megszabhatjuk neki, mennyi eseményt szeretnénk látni a kollekcióban, méghozzá a Take() függvénnyel.

Ezáltal leegyszerűsíthetjük az Initialized eseményre történő fel és leiratkozást:

Normál esetben szükségünk lenne egy osztály szintű eseménykezelőre, amire feliratkozunk, majd önmagában leiratkozunk saját magáról, és ez rögtön egy plusz metódust jelentene.

Ha nem osztályszintű változó eseményére akarnánk feliratkozni, mint ebben az esetben, akkor a lokális változóból is szükség volna rá hogy osztályszintű változó legyen.

Lamda kifejezéssel nem lehet helyettesíteni az eseménykezelőt, mivel a lamda kifejezés egy anonym metódus így nehéz volna róla leíratkozni.

Kézenfekvő tehát az Rx:

```
var subscribe = Observable.FromEvent<EventArgs>(this.CellAdManager, "Initialized")
;
                    subscribe.ObserveOnDispatcher().Take(1).Subscribe(
                        (evt) =>
                        {
                            this.CellAdManager.Subscribe();
                            this.subscribed = true;
                        });
```

A kódrészlet a következőt teszi. Készít egy observable kollekciót a manager példány Initialized eseményéből. Ez a túlterhelés reflekciót használ, de van olyan túlterhelése is ami nem.

Ezek után feliratkozunk a Subscribe segítségével és a következő értékre végrehatjuk a lambda kifejezés által definiált függvényt.

Az ObserveOnDispatcher függvényre a szálbiztonság miatt van szükség, a Take(1) pedig azt jelenti, hogy egy esemény bekövetkezése után fogjuk magunkat és eldobjuk a feliratkozást, ez meghívja a Dispose metódusát, amelyben leiratkozunk az eseménykezelőről.

A másik hely ahol lehetőség van az Rx használatára, a http hívások, azaz a dátum, a lista vagy az aktuális reklám lekérése.

Létezik egy metódus melynek a neve FromAsyncPattern, és pontosan arra szolgál, hogy a Begin és End típusú aszinkron műveletekből Observable-t legyen képes előállítani. Nekünk pontosan erre van szükségünk az aktuális reklám lekérése során.

Az ObservableTimer egy saját implementáció, de túlmutat ezen dokumentum célján.

```
this.interval = new ObservableTimer(TimeSpan.Zero, TimeSpan.FromSeconds(6));
this.interval.Start();

            var response = this.interval.Select(
                (i) =>
                {
```

```
                    var request = HttpWebRequest.CreateHttp(this.CurrentAdString +
 "&nocacheparam=" + new Random().Next());
                    request.Headers["Cache-Control"] = "no-cache";
                    request.Headers["Pragma"] = "no-cache";
                    var getAd = Observable.FromAsyncPattern<WebResponse>(request.B
eginGetResponse, request.EndGetResponse);
                    return getAd().Select(r => r.GetStringFromResponse()).FirstOrD
efault();
                });

            var running = response.Subscribe(
                responseData =>
                {
```

A fenti kódsor a következőket hajtja végre. Létrehoz egy timer-t, amely bizonyos időnként kivált egy eseményt. Ezzel az eseménnyel a Select függvényben létrehozunk egy új HttpRequest-et, beállítjuk az értékeit, majd az Observable.FromAsyncPattern segítségével meghívjük, majd lekérdezzük belőle a response –t.

A running-ban feliratkozunk az így létrehozott Observable kollekcióra, mely számunkra egy respons-al fog rögtön szolgálni, és itt hatjuk végre a többi feladatot, majd átállítjuk az interval-t a megfelelő értékre.

A dátum lekérésénél és a lista lekérésénél is ezt tesszük, csak rögtön le is iratkozunk mivel csak egyszer futnak le ezek a kérések:

```
var getDate = Observable.FromAsyncPattern<WebResponse>(request.BeginGetResponse, r
equest.EndGetResponse);
            getDate().Take(1).Subscribe(
                (response) =>
```

Összefoglalás

Az alkalmazás implementálás során tapasztaltakból kiderül az, hogy a Windows Phone 7 operációs rendszernek, platformnak a jelenlegi formájában még vannak ugyan súlyos hiányosságai, de ezeket nagyobb nehézségek árán ki lehet küszöbölni, vagy körül lehet járni.

Természetesen a workaround technikák bár elfogadottak, soha nem a legcélravezetőbbek, ezért tehát a Microsoftnak van hová fejlesztenie még legújabb termékét.

Az implementáció során törekedtünk arra, hogy minél hűségesebbek legyünk a operációs rendszer jellegéhez, valamint a Microsoft által kiadott Design Guidelines for Windows Phone 7 dokumentumhoz, mely nagyon sok leírást, tanácsot, ajánlást, elvárást tartalmaz.

Ilyen volt többek között a LoopingListSelector általánosított felhasználása, csakúgy mint a LongListSelector animációjának elkészítése.

Az alkalmazás lapok közötti navigációjához is használunk animációkat, szintén a csempe forgatás jellegűeket, melyek az operációs rendszer alapját képezik.

Mindezek segítségével igazán konzisztens felhasználói élményt sikerült nyújtanunk, a felhasználó tényleg úgy érzi, hogy az alkalmazás igazi Windows Phone 7 alkalmazás.

Az alkalmazásunkban minimálisan használjuk az operációsrendszerben választott téma színt, csupán a textboxban és a MainPage oldalon a vezérlők mögötti háttértéglalapon jelenik meg a háttérszín áttetszően, mivel a háttér képtől nem térhettünk el még platformonként sem.

Az MVVM megvalósításnak köszönhetően bármilyen jövőbeli módosítás az alkalmazáson könnyen véghezvihető, mind dizájn, mind logika szempontjából, sőt mivel a felület és a logika lazán csatolt, a kettőt egymástól függetlenül akár két egymástól független személy a másik rész megértése nélkül megteheti, új dizájnt készíthet.

Irodalomjegyzék

Replacing a timer with an observable

http://social.msdn.microsoft.com/Forums/en-CA/rx/thread/e87d1931-6a13-4761-91d1-c42ce0aa8139

WP7 Performance

http://www.slideshare.net/ChrisKoenig/wp7-performance

Handling WP7 orientation changes via Visual States

http://dotneteers.net/blogs/vbandi/archive/2011/03/08/handling-wp7-orientation-changes-via-visual-states.aspx

Things to consider when using WP7 Application-Bar

http://www.windowsphonegeek.com/articles/Things-to-consider-when-using-WP7-Application-Bar

Creating and Reusing Dynamic Animations in Silverlight

http://www.codeproject.com/KB/silverlight/AgDynAnimations.aspx?display=PrintAll&fid=1133652&df=90&mpp=25&noise=3&sort=Position&view=Quick

Bindable Application Bar Extensions for Windows Phone 7

http://www.maxpaulousky.com/blog/archive/2011/01/10/bindable-application-bar-extensions-for-windows-phone-7.aspx

Reactive Extensions

http://msdn.microsoft.com/en-us/data/gg577609

101 Rx Samples

http://rxwiki.wikidot.com/101samples

Implementing a generic loopinglistdatasource

http://www.windowsphonegeek.com/articles/WP7-LoopingSelector-in-depth--Part2Implementing-a-generic-LoopingSelectorDataSource

MVVM Light

http://www.galasoft.ch/mvvm/

Localizing Silverlight-based Applications

http://msdn.microsoft.com/en-us/library/cc838238(v=vs.95).aspx

Tombstoning

http://create.msdn.com/en-US/education/quickstarts/Running_your_App_in_the_Background_(Tombstoning)