Többplatformos mobilalkalmazások

Microsoft Azure

2016.

# Bevezetés

## Célkitűzés

A labor során a hallgatók egy-egy példán keresztül ismerkednek meg a Mobile App és Notification Hub szolgáltatás konfigurálásával, illetve a szerver és kliens oldali fejlesztéssel a Mobile App SDK-n keresztül.

## Előfeltételek

A labor elvégzéséhez szükséges eszközök:

* Visual Studio 2015, (Web development tools, Windows SDK, Windows Phone SDK)
* Azure előfizetés
* Microsoft Developer Account az értesítések kezelésére. (Imagine-nel ingyen igényelhető, <https://imagine.microsoft.com/en-us/Catalog/WindowsStoreDevAccount>)
* SQLite SDK (visual studio kiterjesztés)
  + SQLite for Universal Windows Platform   
    <https://visualstudiogallery.msdn.microsoft.com/4913e7d5-96c9-4dde-a1a1-69820d615936>
* A fenti komponensek telepítéséhez rendszergazdai jogosultság szükséges, a használatukhoz viszont nem

## Amit érdemes átnézned

* Kapcsolódó előadásanyag
* ASP.NET WebAPI

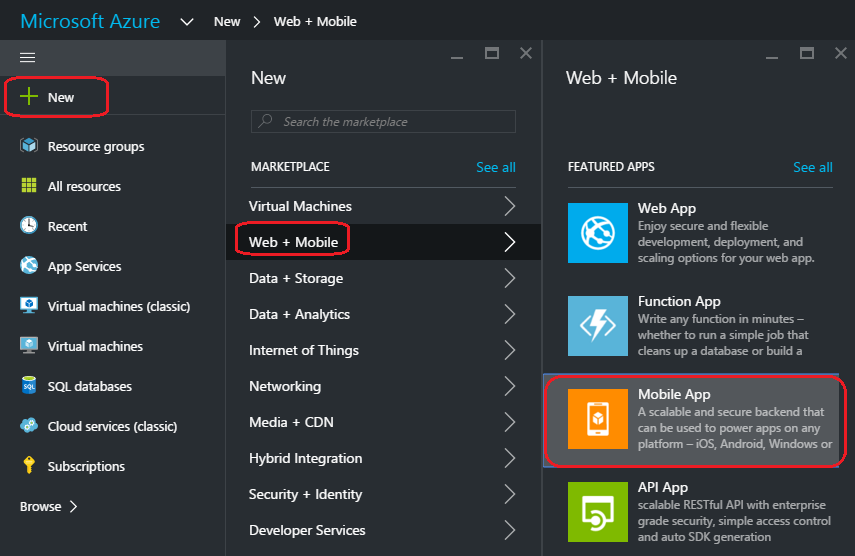
## Oktatói útmutatás

A labor anyag úgy lett összeállítva, hogy teljes szélességében bemutassa a Mobile App és a Notification Hub szolgáltatások gyakorlati használatát. Ebből eredően a teljes labor csak nagyon erőltetett tempóban fér bele 90 percbe, ezért a következő megközelítéseket javasoljuk:

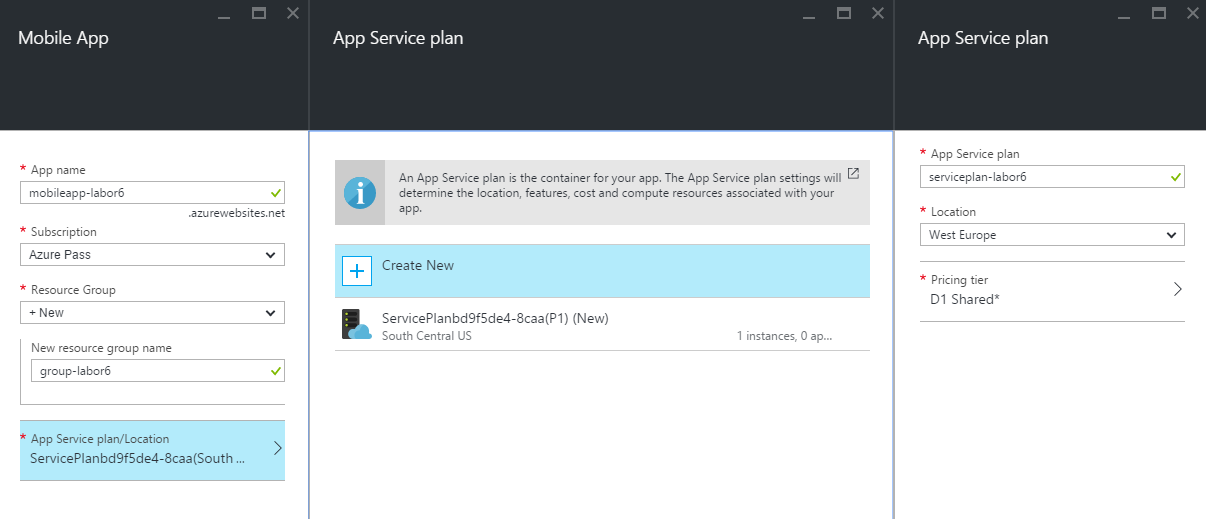
* Vágjunk az anyagból. Hagyjuk ki például a Notification Hub, illetve a Windows Phone platform megvalósítását.
* A begépelendő hosszabb kódrészleteket bocsássuk előre a hallgatók rendelkezésére

# Mobile App létrehozása az Azure portálon

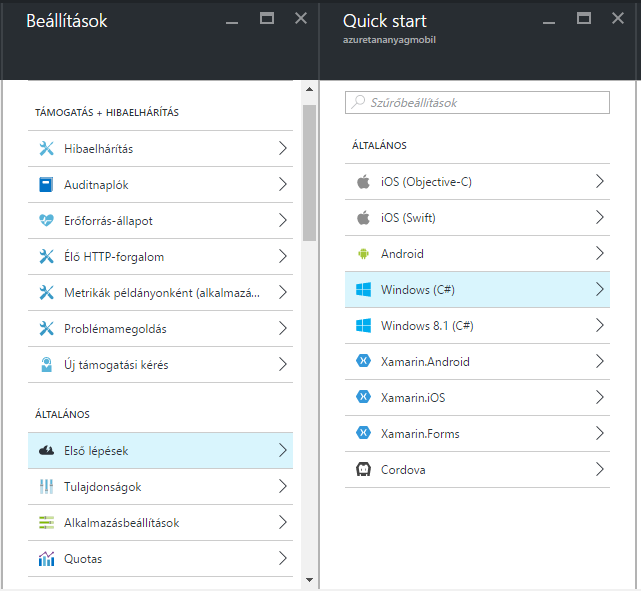
Nyissuk meg a <http://portal.azure.com> adminisztrációs oldalt, majd jelentkezzünk be az előfizetésünkkel összekapcsolt felhasználónkkal. Mielőtt munkához látunk, ne felejtsük el a megfelelő előfizetést kiválasztani (amennyiben több van).

Hozzunk létre egy új erőforrást, ami **Web + Mobile** kategóriában lévő **Mobile App** típusú legyen.

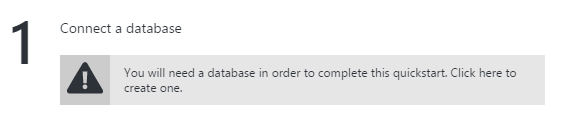
Nevezzük el a szolgáltatásunkat, figyeljünk arra, hogy a neve egyedi legyen. Válasszuk ki melyik erőforráscsoportba (**resource group**) tartozzon (vagy akár újat is létrehozhatunk), és válasszuk ki melyik **App Service plan**-be tartozzon a szolgáltatásunk (vagy készítsünk egy újat), figyeljünk az árazásra is (**pricing tier**)

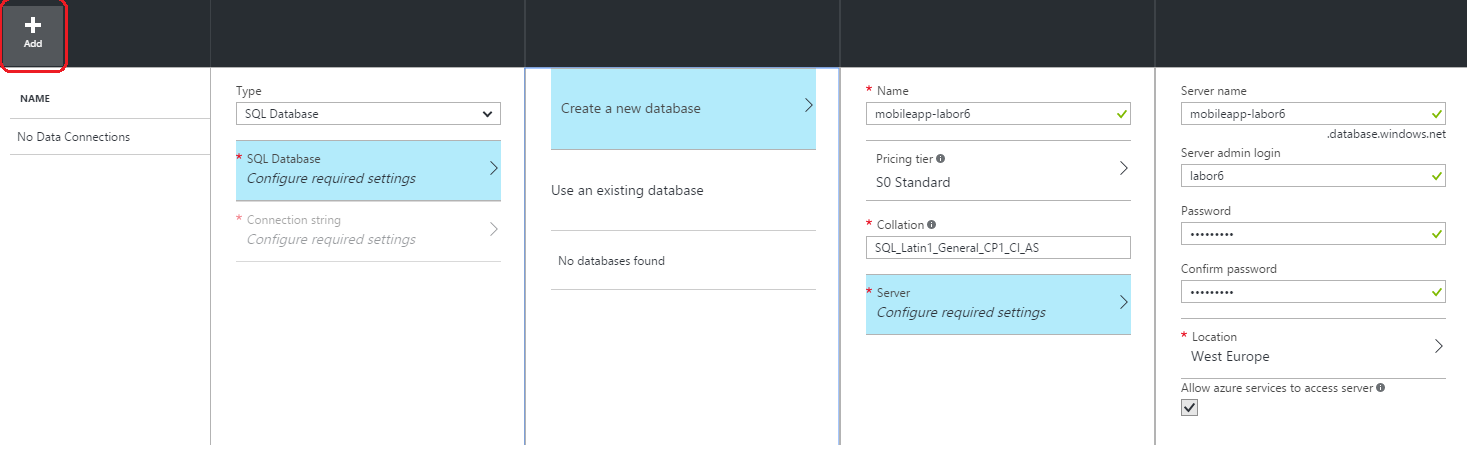


Válasszuk ki az **Első lépések** opciót azon belül pedig a **Windows (C#)** menüpontot, ahol gyorsan bekonfigurálhatunk egy egyszerű projectet, adateléréssel és legenerálhatunk egy szerver oldali és egy kliens oldali Visual Studio projectet. A pontos beállításokat a fenti ábra tartalmazza.

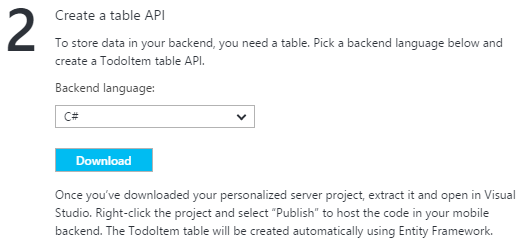


A következő ablakban az 1. lépésként konfiguráljuk be az adatbázis kapcsolatot (**Warning dobozra kattintva**). A **Data Connections** ablakban, készítsünk egy új kapcsolatot a **New** gombra kattintva. Ezen belül pedig használjunk SQL adatbázist, amiből készítsünk egy újat.

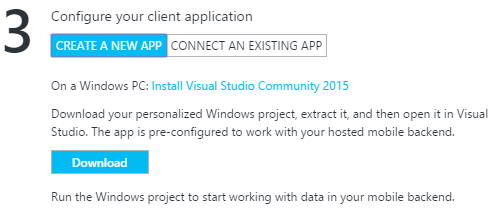




Készítsünk egy szerver oldali példa projectet a 2. pont segítségével, legyen **C#** a kiválasztott nyelv. A **Download** gombra kattintva letölthetjük a generált fájlokat, amiket csomagoljunk is ki egy mappába.



Majd a 3. pontban generáltassuk, majd töltsük le egy új kliens alkalmazás projectjét, és ezt is csomagoljuk ki.



# Szerver alkalmazás áttekintése

A generált alkalmazás szokványos ASP.NET 4.5 MVC 5 platformra épít, nem sok különbség van az eddig megismert Web App-októl. Funkcióját tekintve egy egyszerű teendőket kezelő webalkalmazás, alapvető CRUD műveletekkel.

## Adatelérési Réteg

Az adatokat jelen esetben Azure SQL tárolja, amit az Entity Framework 6.0 keretrendszer segítségével érhetünk el a .NET-es alkalmazásunkból. Az adatbázis sémát Code First módszerrel tartjuk karban.

Az egyetlen entitást a DataObjects mappában találhatjuk a TodoItem osztályban.

public class TodoItem : EntityData

{

public string Text { get; set; }

public bool Complete { get; set; }

}

Észrevehetjük, hogy ez az entitás az EntityData osztályból származik, ami nem Entity Framework specifikus ősosztály, hanem a Mobile App SDK része. Ez az absztrakt ősosztály implementálja az ITableData interfészt, ami definiálja az egységes adatkezeléshez szükséges tulajdonságokat, és a megfelelő metainformációkat (attribútumokat pl.: TableColumn), melyek meghatározzák az adatok megfelelő struktúrába történő perzisztálását.

Az adatbázis kontextust a Models mappában található <appneve>Context osztályban találhatjuk. Ennek az OnModelCreating felüldefiniált metódusában van egy konvenció felvéve, arra, hogy hogyan is kell a TableColumn attribútumot feldolgoznia a keretrendszernek, ami a mező típusát tartalmazza.

protected override void OnModelCreating(DbModelBuilder modelBuilder)

{

modelBuilder.Conventions.Add(

new AttributeToColumnAnnotationConvention<TableColumnAttribute, string>(

"ServiceTableColumn", (property, attributes) => attributes.Single().ColumnType.ToString()));

}

## Inicializáció

A webalkalmazás Mobil App-ként való inicializálása az App\_Start / Startup.MobileApp.cs osztályban található. Itt az ASP.NET pipeline konfigurálása mellett olyan hívásokat is láthatunk, amelyeket a Mobile App SDK-n keresztül hívunk meg.

Alapértelmezett konfigurációk használata, melyek a Web.config állományból kerülnek kiolvasásra.

new MobileAppConfiguration()

.UseDefaultConfiguration()

.ApplyTo(config);

Beállítások elkérése (Web.config) és az Azure App Service környezet inicializációja.

MobileAppSettingsDictionary settings = config.GetMobileAppSettingsProvider().GetMobileAppSettings();

if (string.IsNullOrEmpty(settings.HostName))

{

// This middleware is intended to be used locally for debugging. By default, HostName will

// only have a value when running in an App Service application.

app.UseAppServiceAuthentication(new AppServiceAuthenticationOptions

{

SigningKey = ConfigurationManager.AppSettings["SigningKey"],

ValidAudiences = new[] { ConfigurationManager.AppSettings["ValidAudience"] },

ValidIssuers = new[] { ConfigurationManager.AppSettings["ValidIssuer"] },

TokenHandler = config.GetAppServiceTokenHandler()

});

}

## Controller réteg

Ha a megszokott ApiController leszármazottakat szeretnénk használni a Mobil App szolgáltatásunkban, akkor azokat el kell látni a MobileAppController attribútummal, ami egy filter formájában biztosítja a webes kérések megfelelő integrációját a kliensekkel. Erre láthatunk példát a Controllers / ValuesController.cs osztályban is, ami egy nagyon egyszerű „Helló Világ” funkcionalitást valósít meg.

[MobileAppController]

public class ValuesController : ApiController

{

// GET api/values

public string Get()

{

// ...

}

// POST api/values

public string Post()

{

return "Hello World!";

}

}

A Controllers mappában található másik osztály, a TodoItemController, már bonyolultabb logikát valósít meg: kezeli a TodoItem entitásokkal kapcsolatos CRUD műveleteket. Az SDK itt is segítséget nyújt nekünk úgy, hogy az adatelérés műveleteit egy absztrakciós rétegen keresztül végezhetjük, így nem kell foglalkozzunk azzal, milyen tárolóban is találhatóak a rekordok. A TodoItemController ősosztálya a TableController generikus osztály, ami egy ITableData típusú entitás (jelen esetben TodoItem) kezelését rejti el nekünk.

Mindössze annyi a dolgunk, hogy az Initialize metódus felüldefiniálása során, készíteni kell egy IDomainManager példányt, amin keresztül a TableController tudja a perzisztencia réteget kezelni. Jelen esetben ez egy, az alkalmazásunk adatbázis kontextusával példányosított EntityDomainManager, ami az Entity Framework-öt használja ORM rendszerként.

public class TodoItemController : TableController<TodoItem>

{

protected override void Initialize(HttpControllerContext controllerContext)

{

base.Initialize(controllerContext);

mobileapp\_labor6Context context = new mobileapp\_labor6Context();

DomainManager = new EntityDomainManager<TodoItem>(context, Request);

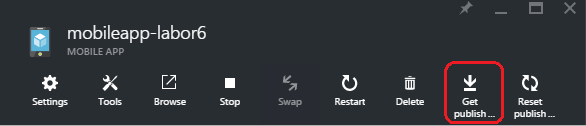
}

}

A lekérdezések, módosítások, új elemek felvétele az ősosztályban található metódusokon keresztül történik (Query, Lookup, Update, Insert, Delete), érdekesség, hogy ezek a műveletek OData szabványokhoz kapcsolódó osztályokat is használnak, mint például a Delta osztály, ami az entitás változásait reprezentálja.

## Publikálás

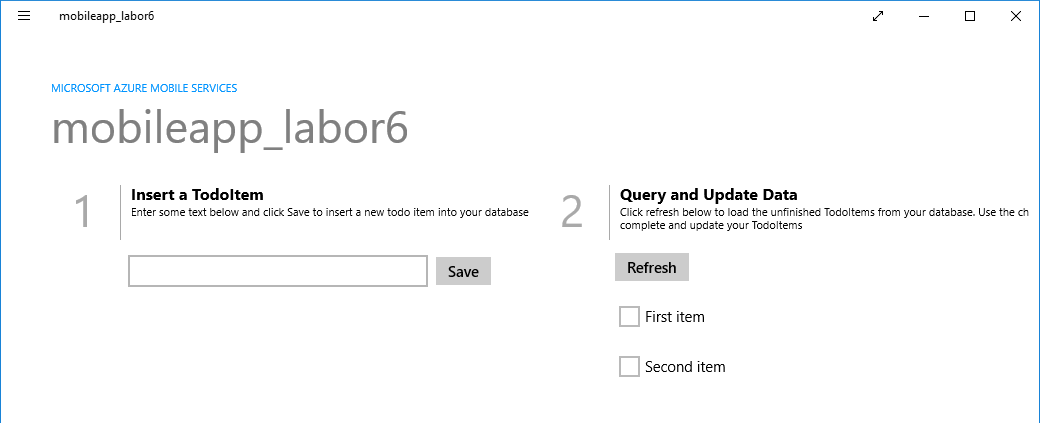
Navigáljunk el az Azure portálon az előzőleg létrehozott mobile app service-ünkhöz, aminek az áttekintő nézetéről töltsük le a **Get publish profile** gomb segítségével a publikáláshoz szükséges beállításokat.



A Visual Studio Solution Explorer nézetében kattintsunk jobb gombbal a projekten, majd a felugró menüből válasszuk a **Publish** menüpontot. A megjelenő ablakban az **Import**-ra kattintva tallózzuk ki az előbb letöltött fájlt. A **Publish** gombra kattintva elindíthatjuk a feltöltést a felhőbe.

# Kliens alkalmazás áttekintése

Az alkalmazás egy képernyőből áll (MainPage.xaml), ahol a Mobile App szolgáltatásunkon keresztül új teendő elemet szúrhatunk be, megtekinthetjük a szerveren meglévő elemeket, és módosíthatjuk azok állapotát (elvégeztük-e már vagy nem).



## Mobile App SDK

Tekintsük át milyen elemek szükségesek a Mobile App szolgáltatással való kommunikáláshoz! Ebben nagy segítséget nyújt a kliens oldali SDK.

Nyissuk meg az App.xaml.cs osztályt, amiben az alkalmazás inicializációja található.

Itt találhatunk egy statikus mezőt, MobileService néven, ami példányosítva is van a szolgáltatásunkra mutató elérési címmel. A típusa MobileServiceClient, és rajta keresztül tudjuk elvégezni az összes lehetséges műveletet, amit a szerver oldalon definiáltunk. (pl.: sima és táblákat kezelő api-k elérése, offline szinkronizáció, stb.)

// This MobileServiceClient has been configured to communicate with the Azure Mobile App.

// You're all set to start working with your Mobile App!

public static MobileServiceClient MobileService = new MobileServiceClient(

"https://mobileapp-labor6.azurewebsites.net"

);

A projekt DataModel könyvtárában találhatjuk az adatosztályokat, jelen esetben a TodoItem POCO osztályt, ami a tulajdonságokat, és a sorosításhoz szükséges metainformációkat tartalmazza.

public class TodoItem

{

public string Id { get; set; }

[JsonProperty(PropertyName = "text")]

public string Text { get; set; }

[JsonProperty(PropertyName = "complete")]

public bool Complete { get; set; }

}

A nézethez tartozó logika a MainPage.xaml.cs forrásfájlban van elhelyezve.

A megjelenítéshez szükséges listát az oldal egy MobileServiceCollection típusú mezőben tárolja (items), ami az ObservableCollection-ből származik a felület frissítéséhez szükséges változásértesítés miatt, illetve megvalósít a szolgáltatáshoz kapcsolódó funkciókat (pl.: inkremetális betöltéshez lapozás).

Egy konkrét táblához tartozó típusos műveleteket egy IMobileServiceTable objektumon keresztül kezelhetjük (todoTable), amit az inicializáció során az App osztály MobileServiceClient objektumától kérünk el.

private MobileServiceCollection<TodoItem, TodoItem> items;

private IMobileServiceTable<TodoItem> todoTable = App.MobileService.GetTable<TodoItem>();

Az adatok lekérése a RefreshTodoItems függvényben történik aszinkron módon az oldal betöltését követően (OnNavigatedTo) és a frissítés gombra kattintva (ButtonRefresh\_Click). A még nem „befejezett” elemek lekérdezés a todoTable (IMobileServiceTable) objektumon keresztül történik, az eredmény pedig az items mezőbe (MobileServiceCollection) kerül, ami pedig majd a lista forrásának van beállítva.

// This code refreshes the entries in the list view by querying the TodoItems table.

// The query excludes completed TodoItems.

items = await todoTable

.Where(todoItem => todoItem.Complete == false)

.ToCollectionAsync();

// ...

ListItems.ItemsSource = items;

Mentés gombra kattintva egy új TodoItem jön létre majd az InsertTodoItem metódus a todoTable (IMobileServiceTable) objektumon keresztül beszúrja azt a szerveren is, illetve a helyi listához is hozzáadásra kerül.

private async void ButtonSave\_Click(object sender, RoutedEventArgs e)

{

var todoItem = new TodoItem { Text = TextInput.Text };

await InsertTodoItem(todoItem);

}

private async Task InsertTodoItem(TodoItem todoItem)

{

// This code inserts a new TodoItem into the database. When the operation completes

// and Mobile App backend has assigned an Id, the item is added to the CollectionView.

await todoTable.InsertAsync(todoItem);

items.Add(todoItem);

}

Az teendő elem melletti check box-ra kattintva az elem állapotának megváltoztatása a szolgáltatás kontextusában a beszúráshoz analóg módon történik.

private async void CheckBoxComplete\_Checked(object sender, RoutedEventArgs e)

{

CheckBox cb = (CheckBox)sender;

TodoItem item = cb.DataContext as TodoItem;

await UpdateCheckedTodoItem(item);

}

private async Task UpdateCheckedTodoItem(TodoItem item)

{

// This code takes a freshly completed TodoItem and updates the database. When the service

// responds, the item is removed from the list.

await todoTable.UpdateAsync(item);

items.Remove(item);

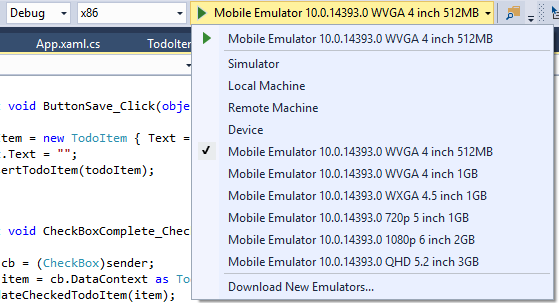
ListItems.Focus(Windows.UI.Xaml.FocusState.Unfocused);

}

## Kipróbálás

**Próbáljuk ki** ennek a tudásnak a fényében az alkalmazást! Vigyázzunk arra, hogy a Solution Platforms-nál ne az ARM legyen kiválasztva!

**Próbáljuk ki** Windows Phone emulátorban is, ehhez módosítsunk az eszközt, amin futtatni szeretnénk az alkalmazást (legyen mondjuk a **Mobile Emulator 10.0.14393.0 WVGA 4 inch 512 MB)**! Az első indulás lassú lehet.

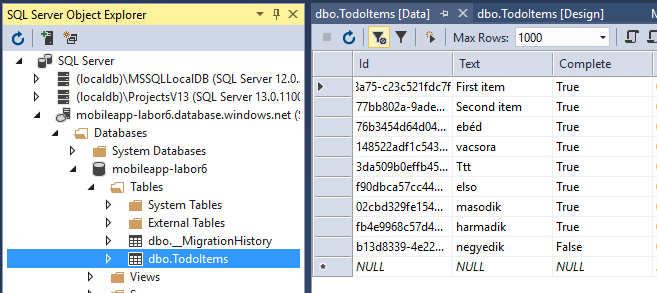


Figyeljük meg, hogy leállítás, majd újbóli indítás után is - nem meglepő módon - megmaradnak a felvett elemek, mivel azokat a felhőben tároljuk és a szolgáltatásunkon keresztül kérjük le!

A Visual Studio-ban az **SQL Server Object Explorer** nézetben egy bejelentkezés után az adatbázisban tárolt adatokat is meg tudjuk tekinteni. ha a kitallózott táblán a jobb gombos menüből a **View Data** menüpontot választjuk.

Előfordulhat, hogy az adatbázis-szerveren a tűzfalszabály érvényre juttatására hibát, kapunk, ilyenkor adjuk ezt hozzá az Azure portálon!

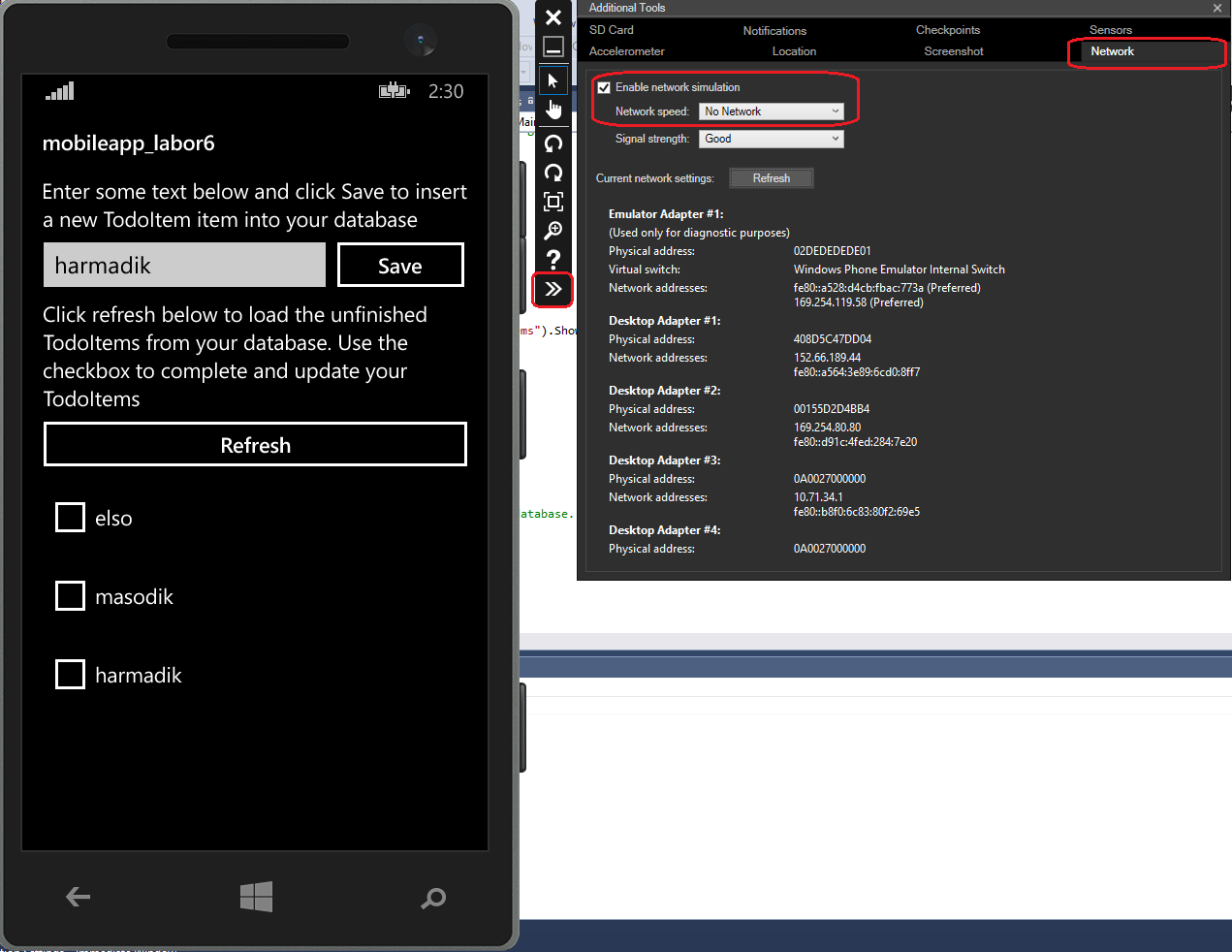
Bejelentkezési információk a portálon a connection string-ben ha elfelejtettük volna



## Offline működés kipróbálása

Próbáljuk ki mi történik, ha nincs az eszközön internet! Ehhez legegyszerűbben az emulátor beállításai között tudjuk szimulálni. (**dupla jobbra nyíl / Network fül / Enable network simulation / No network**)

Tipp: Asztali környezetben a repülő üzemmóddal lehet gyorsan előidézni a kapcsolat nélküli állapotot. A legújabb Windows Phone-os emulátorokban nem lehet No Network-öt szimulálni. Ilyenkor a repülő üzemmód bekapcsolása a hoszt gépen segít.



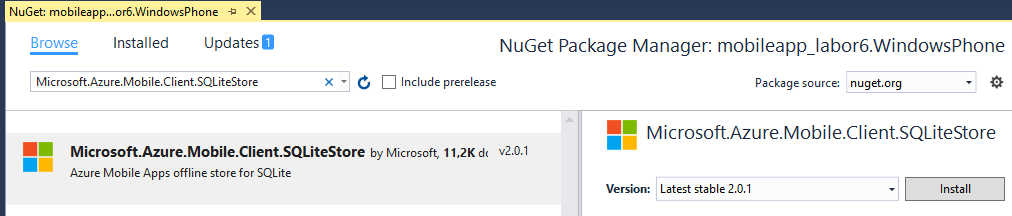
A lista nem tölt be, új elemet sem tudunk felvenni, elszáll az alkalmazásunk, az adatbázisba pedig nem kerülnek bele az elemek.

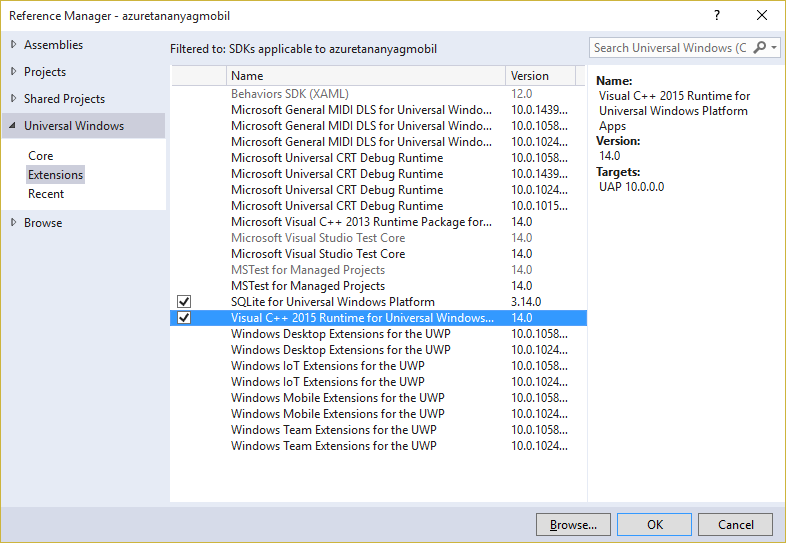
## Offline működés implementáció

Készítsük fel az appot kapcsolat nélküli működésre is. Szerencsére a generált példaalkalmazásunk már tartalmaz olyan kikommentezett kódrészleteket, amelyek ezt a problémát oldják meg az SDK segítségével.

A projekthez adjuk hozzá a kliens oldali perzisztencia réteghez (SQLite) szükséges szerelvényeket, csomagokat.

* Projekt / References jobb gomb / Add reference / Universal Windows fül / Extensions / **SQLite for Universal Windows platform** és **Visual C++ 2015 Runtime for Universal Windows Platform Apps** pipa / OK
* Projekt / References jobb gomb / Manage NuGet packages / Browse / keresés: **Microsoft.Azure.Mobile.Client.SQLiteStore** / Install





Menjünk a MainPage.xaml.cs fájlba és annak a tetején kommentezzük vissza a két névtér hivatkozását. (CTRL + K, CTRL + U)

using Microsoft.WindowsAzure.MobileServices.SQLiteStore; // offline sync

using Microsoft.WindowsAzure.MobileServices.Sync; // offline sync

Az osztály elején az IMobileServiceTable helyett használjuk az IMobileServiceSyncTable osztályt, amit a GetSyncTable metódussal tudunk elkérni a szolgáltatás proxy-tól. (A régi mező már nem kell) Ez az objektum fogja nekünk kezelni a TodoItem-ek kliens oldali perzisztálását, a szerverrel való kommunikációt, és a kettő közötti szinkronizálást.

//private IMobileServiceTable<TodoItem> todoTable = App.MobileService.GetTable<TodoItem>();

private IMobileServiceSyncTable<TodoItem> todoTable = App.MobileService.GetSyncTable<TodoItem>();

A szinkronizálás nem automatikus, nekünk kell inicializálni a tárolót, és kezdeményezni a szinkronizációs folyamatot. Ehhez a fájl alján kommentezzük vissza az InitLocalStoreAsync és a SyncAsync függvényeket. Az első metódus megvizsgálja, hogy inicializálva lett-e már a szinkronizációs kontextus, ha nem akkor létrehozza azt és definiál benne egy táblát a TodoItem elemeknek, majd elindít egy szinkronizációt. A SyncAsync függvény az IMobileServiceSyncTable (todoTable) segítségével küldi fel és kéri le az adatokat. A lekérés során lehetőségünk van szűrést is végezni, így nem fontos az egész táblát leszinkronizálni, ha nem szeretnénk, illetve maga a szinkronizáció spórol a sávszélességgel, mivel idő alapon csak a módosított elemeket mozgatja.

private async Task InitLocalStoreAsync()

{

if (!App.MobileService.SyncContext.IsInitialized)

{

var store = new MobileServiceSQLiteStore("localstore.db");

store.DefineTable<TodoItem>();

await App.MobileService.SyncContext.InitializeAsync(store);

}

await SyncAsync();

}

private async Task SyncAsync()

{

await App.MobileService.SyncContext.PushAsync();

await todoTable.PullAsync("todoItems", todoTable.CreateQuery());

}

Az inicializációt hívjuk meg az oldalra történő navigálás során.

protected override async void OnNavigatedTo(NavigationEventArgs e)

{

**await InitLocalStoreAsync(); // offline sync**

await RefreshTodoItems();

}

A tényleges szinkronizációt hívjuk meg az interakcióink során is.

private async Task InsertTodoItem(TodoItem todoItem)

{

await todoTable.InsertAsync(todoItem);

items.Add(todoItem);

**await SyncAsync(); // offline sync**

}

private async Task UpdateCheckedTodoItem(TodoItem item)

{

await todoTable.UpdateAsync(item);

items.Remove(item);

ListItems.Focus(Windows.UI.Xaml.FocusState.Unfocused);

**await SyncAsync(); // offline sync**

}

private async void ButtonRefresh\_Click(object sender, RoutedEventArgs e)

{

ButtonRefresh.IsEnabled = false;

**await SyncAsync(); // offline sync**

await RefreshTodoItems();

ButtonRefresh.IsEnabled = true;

}

A SyncAsync metódusunkba tegyünk hibakezelést is! A felugró ablakok megjelenítéséhez néhány dolgot még meg kell tennünk. Elsőként vegyünk fel két privát tagváltozót a MainPage-be, ezek pedig legyenek a következők:

private MessageDialog dialog = new MessageDialog("");

private bool isOpen = false;

Az elsőben tároljuk el a dialógusablakunkat, a másodikban pedig azt, hogy éppen van-e nyitva dialógusablak. Erre azért van szükség, mert egyszerre csak egy dialógusablak lehet nyitva. Ha az egyik feldobása előtt egy kicsit több ideig tart az internetkapcsolat „ellenőrzése”, akkor könnyen előidézhetünk olyan helyzetet, hogy akkor szeretné feldobni a programunk a hibajelző dialógusablakot, amikor már egy nyitva van, és UnathorizedAccessExceptiont kapunk.

Írjunk egy aszinkron metódust, amiben egyszerűen eldobjuk a kérést, hogyha nyitva van a dialógus, egyébként pedig feldobjuk a felhasználónak:

private async Task ShowDialogAsync(string content, string title = "Hiba")

{

if (isOpen)

{

return;

}

dialog.Title = title;

dialog.Content = content;

isOpen = true;

await dialog.ShowAsync();

isOpen = false;

}

Módosítsuk a SyncAsync metódust a következőképpen: a két utasítást tegyük Try-Catch blokkba, a megfelelő ágakat pedig töltsük ki!

private async Task SyncAsync()

{

string content;

try

{

await App.MobileService.SyncContext.PushAsync();

await todoTable.PullAsync("todoItems", todoTable.CreateQuery());

}

catch (MobileServicePushFailedException ex)

{

content = "Szinkronizáció nem sikerült, lehet, hogy kapcsolat nélküli állapotban vagy.\n" +

"Üzenet: " + ex.Message + "\nÁllapot: " + ex.PushResult.Status.ToString();

await ShowDialogAsync(content);

}

catch (Exception ex)

{

content = "Szinkronizáció sikertelen " + ex.Message +

"\n\nHa még mindig kapcsolat nélküli állapotban vagy, " +

"próbálj frissíteni a kapcsolat felépítését követően.";

await ShowDialogAsync(content);

}

}

Ezen kívül egy helyen dob fel még a mintaalkalmazás MessageDialogot, ezt a metódust is módosítsuk, hogy a megírt metódusunkat használja (**RefreshTodoItems)**:

private async Task RefreshTodoItems()

{

MobileServiceInvalidOperationException exception = null;

try

{

// This code refreshes the entries in the list view by querying the TodoItems table.

// The query excludes completed TodoItems

items = await todoTable

.Where(todoItem => todoItem.Complete == false)

.ToCollectionAsync();

}

catch (MobileServiceInvalidOperationException e)

{

exception = e;

}

if (exception != null)

{

await ShowDialogAsync(exception.Message, "Error loading items");

}

else

{

ListItems.ItemsSource = items;

this.ButtonSave.IsEnabled = true;

}

}

Offline esetben MobileServicePushFailedException dobódik, ahol a PushResult.Status CancelledByNetworkError értéket vesz fel.

Próbáljuk ki!

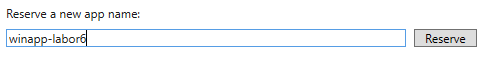
Azt tapasztaljuk, hogy ha az internetkapcsolat él, minden rendben működik, ha pedig megszakítjuk a kapcsolatot, a hibakezelő ablakok feldobódnak, és a szerver oldali adatbázis táblában nincsenek benne az újonnan felvett/módosított adatok. Ha visszakapcsoljuk az internetet és frissítésre nyomunk, vagy újraindítjuk az alkalmazást, a helyileg tárolt adatok feltöltődnek a szerverre és már az adatbázisban található rekordok között is megjelennek.

# Notification Hub

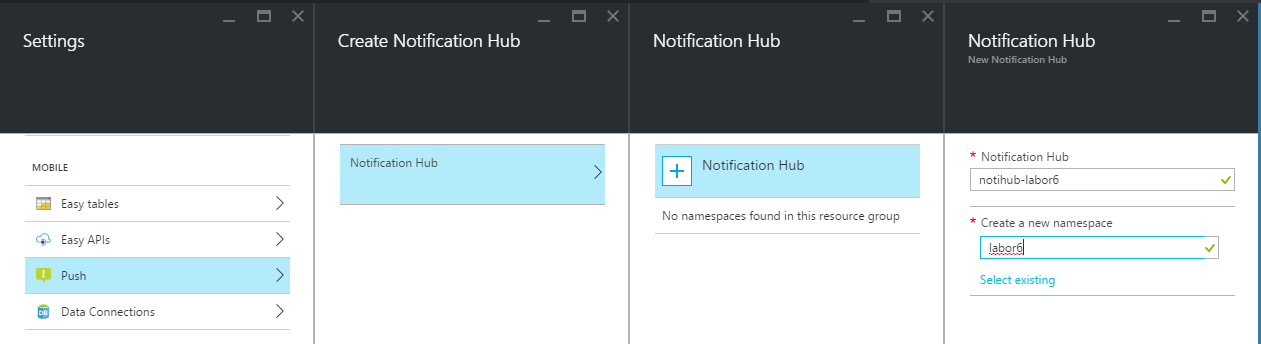
Készítsünk értesítés küldés funkcionalitást az alkalmazásunk számára.

## Adminisztráció

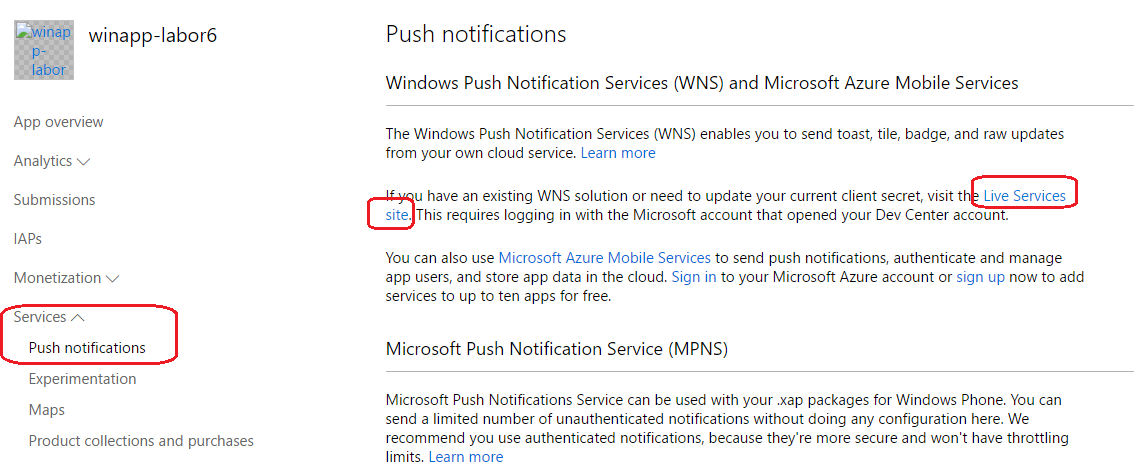
Regisztráljuk be az alkalmazásunkat a Windows Store-ba (nem publikáljuk, csak regisztráljuk, hogy tudjon a rendszer róla): a projekten jobb gomb **Store / Associate App with the Store,** jelentkezzünk be a (Dreamspark-kal igényelt) Microsoft Developer fiókunkkal. Adjunk nevet az alkalmazásunknak, majd a frissülő listából válasszuk ki, és kattintsunk a következő oldalon az **Associate** gombra.



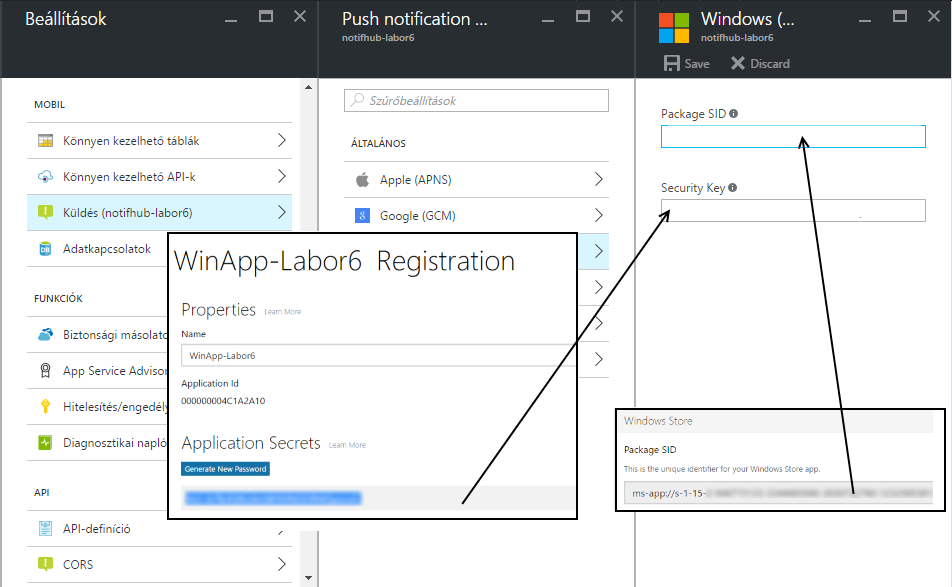
Az Azure portálon a Mobile App beállításai között válasszuk ki a **Küldés** menüpontot, és ott készítsünk egy új **Notification** **Hub**-ot tetszőleges névvel és tetszőleges névtérbe. (A **Create Notification Hub** ablak **Létrehozás** gombjára se felejtsünk el kattintani)



Ha készen van a Hub, adjuk meg a beállításait, amit a beregisztrált appunk adatai között találunk meg a Microsoft developer oldalon. <https://developer.microsoft.com/en-us/dashboard/overview> / app kiválasztása / Services / Push notifications / Live Services link



Ez a link elirányít az app szolgáltatásainak részletes oldalára, ahonnan a Package SID-t és az Application Secret-et kell átmásolni az Azure portálon megnyitott felületen.



## Szerver oldal

Most hozzá kellene adni a szerver alkalmazásunkhoz regisztrációs végpontot, ahova a kliensek a push értesítési csatorna azonosítóikat tudják beregisztrálni. Szerencsére az általunk használt konfiguráció (UseDefaultConfiguration) alapértelmezetten tartalmazza ezt. Ha nem ezt használnánk akkor az AddPushNotifications bővítő metódust kellene meghívni a konfiguráción.

Küldjünk push értesítést a kliensnek, ha elvégezte egy teendőjét. Esetünkben ez a TodoItemController PatchTodoItem metódusában következik be, mivel jelenleg a kliens csak akkor hívja ezt a műveletet, ha elvégzettre állítja az elem állapotát.

Készítsünk egy függvényt, ami a küldést hajtja végre:

private async Task SendNotification(TodoItem item)

{

MobileAppSettingsDictionary settings =

Configuration.GetMobileAppSettingsProvider().GetMobileAppSettings();

string notificationHubName = settings.NotificationHubName;

string notificationHubConnection =

settings.Connections[MobileAppSettingsKeys.NotificationHubConnectionString].ConnectionString;

NotificationHubClient hub = NotificationHubClient.CreateClientFromConnectionString(

notificationHubConnection, notificationHubName);

var windowsToastPayload =

@"<toast><visual><binding template=""ToastText01""><text id=""1"">" +

item.Text + @" Completed :)</text></binding></visual></toast>";

try

{

var result = await hub.SendWindowsNativeNotificationAsync(windowsToastPayload);

Configuration.Services.GetTraceWriter().Info(result.State.ToString());

}

catch (System.Exception ex)

{

Configuration.Services.GetTraceWriter().Error(ex.Message, null, "Push.SendAsync Error");

}

}

A működése egyszerű, a Mobile App beállításai alapján példányosítunk egy NotificationHub-ot (Microsoft.Azure.NotificationHubs nuget ha nem lenne benne), majd a Windows platformra specifikus XML-t elküldjük a hubon keresztül + logolunk egy keveset.

És hívjuk ezt meg a PatchTodoItem a visszatérés előtt!

public async Task<TodoItem> PatchTodoItem(string id, Delta<TodoItem> patch)

{

await SendNotification(patch.GetEntity());

return await UpdateAsync(id, patch);

}

Publikáljuk a megszokott módon a web alkalmazásunkat!

## Kliens oldal

Nyissuk meg az App.xaml.cs osztályt és írjunk egy metódust, ami a csatorna azonosító beregisztrálását végzi, a MobileServiceClient proxyn keresztül.

private async Task InitNotificationsAsync()

{

var channel = await PushNotificationChannelManager.CreatePushNotificationChannelForApplicationAsync();

await MobileService.GetPush().RegisterAsync(channel.Uri, null);

}

Hívjuk meg az App osztály OnLaunched metódusa végén.

protected override async void OnLaunched(LaunchActivatedEventArgs e)

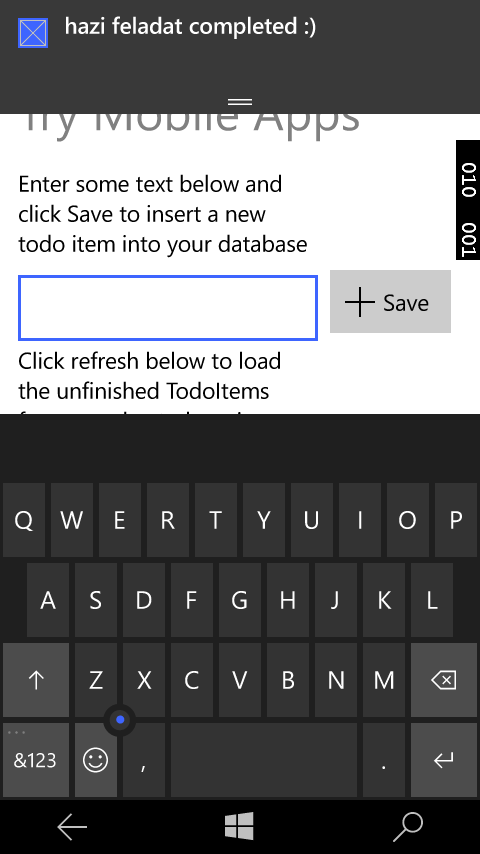
{

// ...

await InitNotificationsAsync();

}

Próbáljuk ki!



Ha csak magában szeretnénk használni a Notification Hub-ot és nem a Mobile App keretei között, akkor más könyvtárakat és API-kat kell használnunk szerver és kliens oldalon is. Ennek többek között az lehet az oka, hogy a Notification Hub teljes funkcionalitását szeretnénk felhasználni (pl.: Tag-ek), amiket a Mobile App SDK nem támogat jelenleg.