Adattárolás a felhőben

Microsoft Azure

2016.

# Bevezetés

## Célkitűzés

A labor célja, hogy a hallgatók megismerkedjenek az Azure Storage működésével:

* Alkalmazásfejlesztés Blobok használatával
* Alkalmazásfejlesztés Table Storage használatával
* Rövid kitekintés a relációs adatbázisok használata felé

## Előfeltételek

A labor elvégzéséhez szükséges eszközök:

* Visual Studio 2015
* Azure előfizetés

## Amit érdemes átnézned

* Kapcsolódó előadásanyag

## Oktatói útmutatás

Az útmutató első két feladata a Blob Storage és a Table Storage használatára mutat egy-egy egyszerű példát. Néhány praktikus tudnivaló:

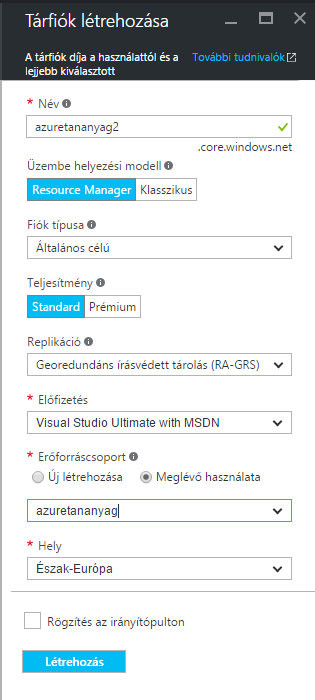
**Azure hozzáférés**: Az első két feladat teljes értékűen megoldható tisztán az Azure emulátor használatával, vagy egy oktató által létrehozott Tárfiókkal (storage account). A hallgatók Imagine (not billable) subscription-je nem alkalmas tárhelyfiók létrehozására, így ők az oktató által létrehozott tárhelyfiókot vagy Storage Emulator-t használhatnak. A Storage Emulator telepítéséhez rendszergazdai jogosultság szükséges.

**Technológia**: Az első példa Windows Forms, míg a második ASP.NET MVC technológiával készült. Mindkét példa könnyen átültethető a másik technológiára, így ha hallgatóink nem ismerik valamelyiket, könnyen igazíthatjuk ehhez a laborgyakorlatot is.

**Időgazdálkodás**: A gyakorlatok 90 perc alatt történő befejezéséhez érdemes a feladatok egyes részeit elhagyni, vagy a nagyobb forráskódrészleteket előre megosztani a hallgatókkal.

### Azure Tárhely-fók létrehozása

Röviden emlékeztessük a hallgatókat tárhely-fiók szerepére az előadás alapján, majd hozzunk létre egy tárhely fiókot! Ehhez az Azure portál (<https://portal.azure.com>) Új 🡺 Adatok és tárolás 🡺 Tárfiók parancsait használjuk. A tárhely fiók beállításához a lenti paramétereket használjuk:



A névnek globálisan egyedinek kell lennie ezért megadásakor legyünk kreatívak.

# Alkalmazásfejlesztés

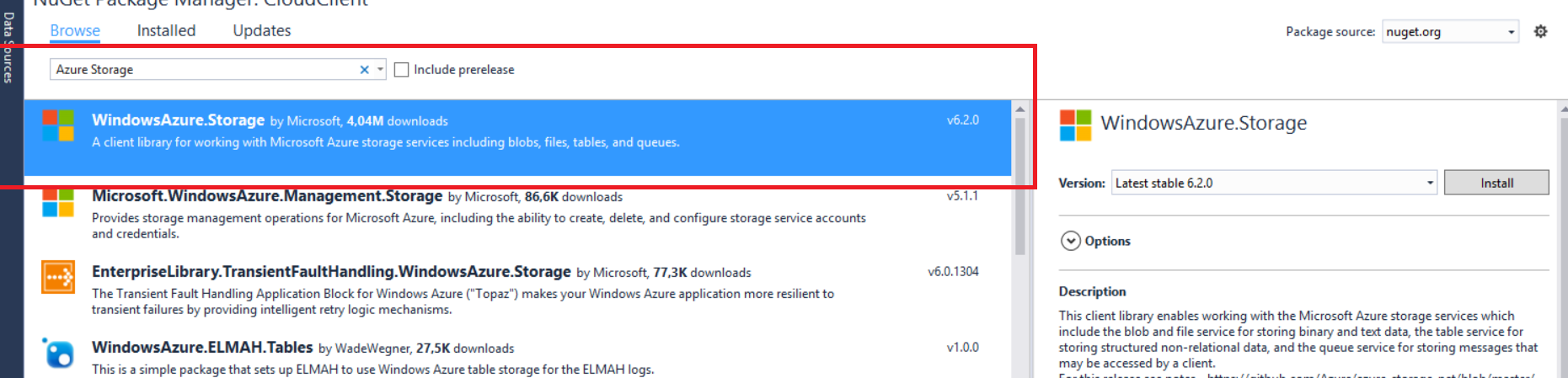
A következő részben két alkalmazást fogunk készíteni. Az egyik Blob-ok feltöltését, böngészését és letöltését teszi lehetővé Windows Forms alapokon, a másik pedig egy Todo alkalmazás MVC alapokon a Table storage képességeinek a szemléltetésére.

## Blob tárhely

### Előkészítés

Hozzunk létre egy új Windows Forms alkalmazást Visual Studioban (File / New Project / Visual C# / Windows / Windows Forms Application)!

Rögtön adjuk is hozzá a WindowsAzure.Storage NuGet csomagot a projekthez (Project / Manage NuGet packages… / Browse):

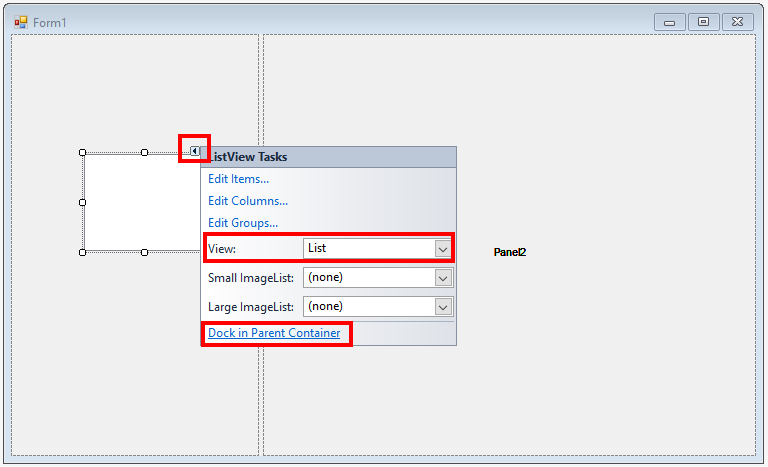


Ezután készítsük el a Windows Forms alkalmazás vázát. Meséljük el, hogy mire lesz képes az alkalmazás:

* két panel, bal oldalon a konténerek, jobb oldalon a konténerben lévő Blobok
* konténer létrehozása a bal oldalon context menüből
* blob feltöltése a jobb oldalon context menüből
* blob letöltése a jobb oldalon context menüből

Először készítsük el az alkalmazás vázát. A Toolboxból húzzunk egy SplitContainer vezérlőt a Formra és győződjünk meg arról, hogy az elválasztó vonal függőlegesen osztja ketté a Formot (alapbeállítás).

Ezután mind a bal oldali, mind a jobb oldali panelba húzzunk bele a Toolboxból egy ListView-t. Mindkét esetben használjuk a ListView smarttagjét és állítsuk be, hogy a ListView nézete „List” értéken legyen, és hogy ki legyen dokkolva a szülő vezérlőben:



Nevezzük el a baloldalit lvContainers-nek, a jobb oldalit lvBlobs-nak. Ezután húzzunk még rá 2 ContextMenuStrip-et a Toolboxról a Formra. Végezzük el a következő műveleteket:

* contextMenuStrip1: Adjunk hozzá egy új menüelemet, Create container néven. Ezután az lvContainer vezérlő tulajdonságai között a ContextMenuStrip tulajdonság értékét állítsuk be erre a vezérlőre.
* contextMenuStrip2: Adjunk hozzá egy új menüelemet, Upload blob néven. Nevezzük át uploadBlob-ra. Ezután az lvBlob vezérlő tulajdonságai között a ContextMenuStrip tulajdonság értékét állítsuk be erre a vezérlőre.

Ha most elindítjuk az alkalmazást, még nem látunk semmit, de a panel két oldalában a context menük már előjönnek.

Ezután vegyünk fel egy új, statikus osztályt a projektbe CloudManager néven. Ide fogjuk majd megírni az üzleti logikát (konténerek és blobok listázása, letöltés, feltöltés). Vegyünk fel két statikus mezőt:

public static class CloudManager

{

private static readonly string connectionString = "";

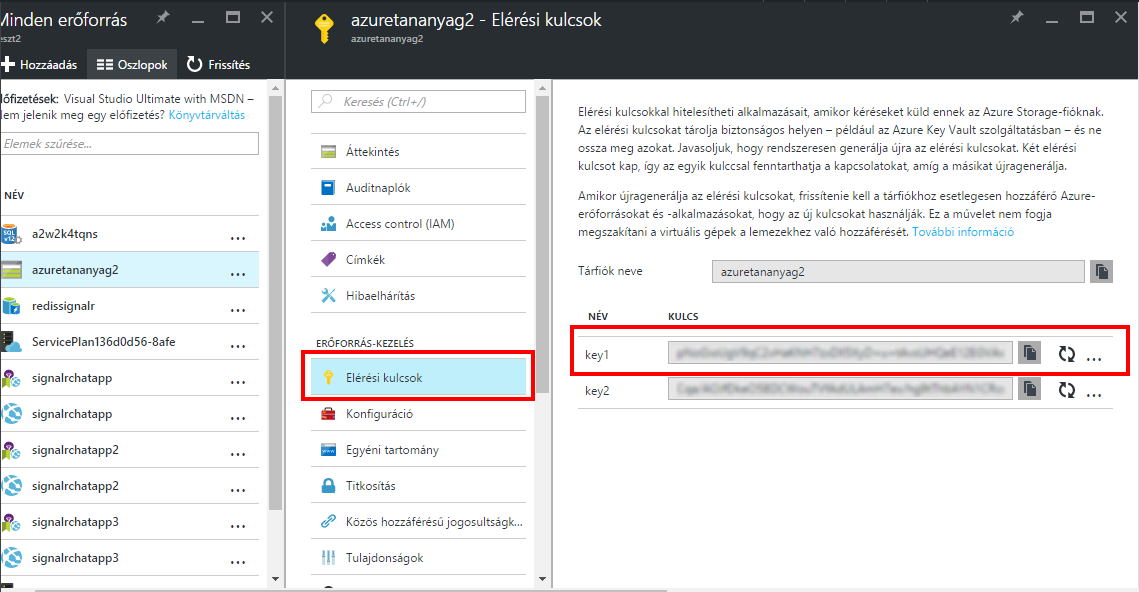
private static CloudStorageAccount account = CloudStorageAccount.Parse( connectionString );

}

* A connectionString értékét a következőképpen kell előállítani:

DefaultEndpointsProtocol=http;AccountName=(…);AccountKey=(…)

Az AccountName helyére kerül a tárfiók neve, amit megadtunk a létrehozáskor (ebben a példában azuretananyag2), az AccountKey-t pedig a portálról tudjuk megkapni, a tárfiók beállításainál az elérési kulcsok közül a key1-et kell használunk.



* Az account nevű adattag fog majd belépési pontot nyújtani a blob tárhely felé. Ezen keresztül tudjuk a tárhellyel a kapcsolatot felvenni, műveleteket végezni.

Ha nem szeretnénk valódi felhős tárhelyhez csatlakozni, az account változó értéke lehet CloudStorageAccount.DevelopmentStorage. Ebben az esetben a lokális emulátorhoz fogunk csatlakozni (ne felejtsük el elindítani előtte!).

### Konténerek megjelenítése

Első feladatként listázzuk ki a konténereket. Ehhez vegyünk fel egy új műveletet a CloudManager osztályba:

public static List<CloudBlobContainer> GetContainers()

{

var client = account.CreateCloudBlobClient();

return client.ListContainers().ToList();

}

A CloudStorageAccount osztály nem csak a blobok, hanem az összes többi tárhelyszolgáltatás felé is belépési pont. Ezért először egy CloudBlobClient példányt kell létrehoznunk, ami a már a Blob specifikus műveleteket fogja elérhetővé tenni. Egy ilyen művelet a ListContainers(), ami az összes konténert kilistázza.

Kössük be ezt a műveletet a Formra! Elsőként a Form.cs code-behind állományába vegyük fel az alábbi műveletet:

private void PopulateContainers()

{

lvContainers.Items.Clear();

List<CloudBlobContainer> containers = CloudManager.GetContainers();

foreach (var container in containers)

{

lvContainers.Items.Add(new ListViewItem {Text = container.Name, Tag = container});

}

}

Először kitöröljük a konténerlistában lévő összes elemet. Ezután pedig az előbb megírt metódussal lekérdezzük az összes konténert, majd mindegyikhez létrehozunk egy ListViewItem objektumpéldányt. Ezeknek a Text tulajdonsága (ami megjelenik a felületen) a konténer neve lesz, a Tag tulajdonsága (egy olyan tulajdonság, ami a vezérlőhöz tetszőleges értéket el tud tárolni későbbi használatra), maga a konténer.

Végül iratkozzunk fel a Form Loaded eseményére és az eseménykezelőt írjuk meg az alábbi módon:

private void Form1\_Load( object sender, EventArgs e )

{

PopulateContainers();

}

Ha most elindítjuk az alkalmazást, akkor egyelőre még nem látunk semmit, hiszen egyelőre nincsenek konténereink. Létrehozhatunk egyet a Visual Studio Cloud Explorerben, hogy kipróbáljuk, valóban jól működik-e a kód.

Írjuk meg a CloudManager-be a konténer létrehozását:

public static void CreateContainer( string containerName )

{

var client = account.CreateCloudBlobClient();

var container = client.GetContainerReference( containerName );

container.CreateIfNotExists();

container.SetPermissions( new BlobContainerPermissions {

PublicAccess = BlobContainerPublicAccessType.Off } );

}

Újra létrehozzuk a CloudBlobClient-et, majd ezen keresztül a GetContainerReference() metódussal a konténerre mutató referenciát. Ez a referencia az adott konténerrel végezhető műveleteket teszi elérhetővé. Egy ilyen művelet a konténer létrehozása, ha még nem létezik, ezt végzi el a CreateIfNotExists() metódus. Egy másik ilyen lehetséges művelet a jogosultságok beállítása, ezt végzi a SetPermissions művelet. Itt, ha időnk engedi, nézzük meg a többi opciót is.

Végül kössük be ezt is a felületre! A contextMenuStrip1 Create container menüelemének Click eseményére iratkozzunk fel. Az eseménykezelőben használjuk a Prompt osztályt arra, hogy bekérjük a felhasználótól a konténer nevét (ennek forráskódját célszerű előre a hallgatókhoz eljuttatni, ld. Függelék), majd ezzel a névvel hozzuk létre a konténert az előbb megírt metódus segítségével (végül töltsük újra a konténerek listáját):

private void createContainerToolStripMenuItem\_Click( object sender, EventArgs e )

{

string containerName = Prompt.ShowDialog( "Container name", "Create container" );

CloudManager.CreateContainer( containerName );

PopulateContainers();

}

Ha most elindítjuk az alkalmazást, már tudnunk kell konténereket felvenni a bal oldali nézetben. Ha a tároló nevébe ékezetes betűt vagy szóközt írunk, akkor hibát fogunk kapni.

### Blobok feltöltése

A következő lépés a blobok feltöltésének megírása. Ehhez az alábbi műveletet vegyük fel a CloudManagerbe:

public static void UploadBlob(string file, CloudBlobContainer container)

{

var blob = container.GetBlockBlobReference(Path.GetFileName(file));

blob.UploadFromFile(file);

}

Először a megadott fájlnak meghatározzuk a nevét (Path.GetFileName()), majd a paraméterként kapott konténeren belül megszerezzük erre a hivatkozást (GetBlockBlobReference()) (itt emlékeztethetünk arra, hogy mi az a blokk blob és miért ezt használjuk). A megadott fájl elérési útjának megadásával közvetlenül feltölthető a fájl.

Ezután már csak be kell kötnünk a context menübe a feltöltést. Ehhez az uploadBlob ContextMenuStrip egyetlen menüelemének Click eseményére iratkozzunk fel és az eseménykezelőt az alábbi módon írjuk meg:

private void uploadBlobToolStripMenuItem\_Click( object sender, EventArgs e )

{

if( lvContainers.SelectedItems.Count != 1 )

return;

OpenFileDialog getFileDialog = new OpenFileDialog();

if( getFileDialog.ShowDialog() == DialogResult.OK )

{

CloudManager.UploadBlob( getFileDialog.FileName,

(CloudBlobContainer)lvContainers.SelectedItems[0].Tag );

}

}

Először meggyőződünk róla, hogy pontosan 1 elem van kijelölve a baloldalon, majd az OpenFileDialog dialógusablak segítségével kitallóztatjuk a fájlt. Végül az előbb megírt műveletnek átadjuk paraméterül a fájlnevet és a baloldalon kiválasztott listaelem Tag tulajdonságában lévő CloudBlobContainer objektmot (ezért tároltuk el korábban, itt emlékeztessünk erre).

Ezután kipróbálhatjuk a feltöltést; a Cloud Explorerben tudjuk ellenőrizni az eredményt.

### Blobok listázása

Listázzuk ki a feltöltött blobokat, ha a felhasználó rákattint egy konténerre a bal oldalon! Ehhez először a CloudManagerbe vegyünk fel egy új műveletet:

public static List<IListBlobItem> GetBlobs( CloudBlobContainer container )

{

return container.ListBlobs().ToList();

}

Ez egyszerűen csak a paraméterként kapott konténer blobjait adja vissza.

Ezután vegyünk fel egy PopulateBlobs() nevű metódust a Form code-behind részébe:

private void PopulateBlobs()

{

lvBlobs.Items.Clear();

if (lvContainers.SelectedItems.Count != 0)

{

var blobs = CloudManager.GetBlobs((CloudBlobContainer) lvContainers.SelectedItems[0].Tag);

foreach (var blob in blobs)

{

lvBlobs.Items.Add(new ListViewItem {Text = blob.Uri.ToString(), Tag = blob.Uri});

}

}

}

Először kitöröljük az elemeket, majd ha pontosan egy elem van a bal oldali részen kijelölve, akkor a kijelölt elem Tag tulajdonságában lévő CloudBlobContainer-t felhasználva lekérjük a konténer blobjait az előzőleg megírt metódussal. Ahogy a konténerek esetén, itt is mindegyikhez létrehozunk egy listaelemet, ahol a szöveg a blob uri-ja, a Tag tulajdonság pedig az Uri objektum.

Végül iratkozzunk fel a bal oldali listview (lvContainers) SelectedIndexChanged eseményére, és ebben hívjuk meg ezt a PopulateBlobs() műveletet:

private void lvContainers\_SelectedIndexChanged( object sender, EventArgs e )

{

PopulateBlobs();

}

Szintén érdemes a PopulateBlobs() művelethívást beszúrni a feltöltés végére:

private void uploadBlobToolStripMenuItem\_Click( object sender, EventArgs e )

{

if( lvContainers.SelectedItems.Count != 1 )

{

return;

}

OpenFileDialog getFileDialog = new OpenFileDialog();

if( getFileDialog.ShowDialog() == DialogResult.OK )

{

CloudManager.UploadBlob( getFileDialog.FileName,

(CloudBlobContainer)lvContainers.SelectedItems[0].Tag );

}

PopulateBlobs();

}

Ha most kipróbáljuk az alkalmazást, már listázni is tudjuk a blobokat.

### Blobok letöltése

Az utolsó lépés a letöltés. Ehhez vegyük fel az alábbi metódusot a CloudManagerbe:

public static byte[] DownloadBlob(Uri uri)

{

var client = account.CreateCloudBlobClient();

var blob=client.GetBlobReferenceFromServer( uri );

blob.FetchAttributes();

var result = new byte[blob.Properties.Length];

blob.DownloadToByteArray( result, 0 );

return result;

}

A GetBlobReferenceFromServer() metódussal úgy kaphatunk egy blobra mutató referenciát, hogy a konténerre mutató referenciát nem kell előtte elkérnünk (ez is része az urinak). A blob referencia viszont csak egy üres hivatkozás az urin kívül nem sok minden van benne. Nincs benne például a blob hossza sem, de erre szükségünk van, ha byte tömbként szeretnénk lementeni (alternatívaként rögtön lementhetjük fájlba, vagy akár Stream-re is, ilyenkor nem kell a hosszt előre tudnunk). Először a FetchAttributes() metódust hívjuk meg, ami a blob nagyon sok tulajdonságát lekéri a felhőből, így például a hosszát is. Ezután a Properties.Length tulajdonságon keresztül már tudjuk, milyen hosszú bájttömbre van szükségünk. Végül a DownloadToByteArray() művelettel egyszerűen letöltjük a blob-ot.

Ezután már csak be kell kötnünk a felületre ezt a műveletet is. Ehhez először iratkozzunk fel a jobb oldali ListView (lvBlobs) DoubleClick eseményére, majd töltsük ki az alábbi módon:

private void lvBlobs\_DoubleClick(object sender, EventArgs e)

{

if (lvBlobs.SelectedItems.Count != 1) return;

var selectedItem = lvBlobs.SelectedItems[0];

var uri = (Uri)selectedItem.Tag;

SaveFileDialog saveFile = new SaveFileDialog()

{

FileName = Path.GetFileName(uri.ToString())

};

if (saveFile.ShowDialog() == DialogResult.OK)

{

var contents = CloudManager.DownloadBlob(uri);

File.WriteAllBytes(saveFile.FileName, contents);

}

}

Annyit teszünk, hogyha nincs kijelölve egyetlen blob sem, akkor egyszerűen kilépünk a metódusból (hiszen az is előfordulhat, hogy az „üres” részre kattintunk kétszer, nem az egyik elemre). Itt egyszerűen csak a SaveFileDialog segítségével kitallóztatjuk a mentés helyét, és megadjuk a fájl nevét alapértelmezettként, majd a CloudManager DownloadBlob() művelete segítségével letöltjük az aktuálisan kijelölt blobot. Az url-et a kijelölt listaelem Tag tulajdonságából vesszük ki újra (ezt korábban eltároltuk, emlékeztessünk rá itt).

Próbáljuk ki az alkalmazást! Ezzel megismerkedtünk a blobok alapvető működésével. A következő fejezetben a Table Storage működését fogjuk megismerni.

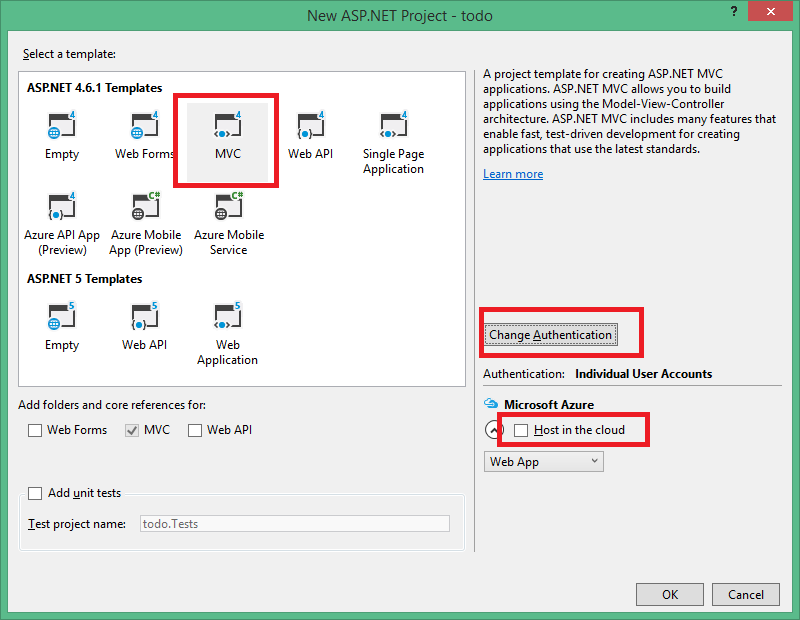
## Table Storage

Ebben a részben a Table Storage működését fogjuk megismerni egy egyszerű Todo alkalmazáson keresztül, amely a feladatokat fogja Table Storage-ba menteni.

### Előkészítés

Visual Studio-ban hozzunk létre egy új ASP.NET Web Application projektet ( File=>New Project=>Visual C#=>Web=>ASP.NET Web Application)! A webalkalmazás létrehozásakor az alábbi beállításokra figyeljünk:

* A projektet MVC-s alkalmazássablon szerint hozzuk létre
* A „Host int he cloud” opciót kapcsoljuk ki
* A Change Authentication opciót használva válasszuk a „No authentication” opciót



Telepítsük a projekthez is az Azure Storage kliensosztály-könyvtárat (ld. előző rész).

### Modell létrehozása

Elsőként vegyük fel a modell osztályunkat! Vegyünk fel a projekt „Models” mappájába egy új osztályt Item néven!

Az Item osztály fogja reprezentálni a teendőket; ez hagyományos adatbázisok esetén megfelel a rekordtípusnak. Az Item osztályban egy név és egy leírás fog még szerepelni, illetve egy flag, amivel jelezzük, hogy a feladat készen van-e már:

public class Item : TableEntity

{

[JsonProperty(PropertyName="name")]

public string Name { get; set; }

[JsonProperty(PropertyName = "desc")]

public string Description { get; set; }

[JsonProperty(PropertyName="isComplete")]

public bool Completed { get; set; }

}

Az osztálynak kötelezően a TableEntity osztályból kell származnia. Ez definiálja a PartitionKey és a RowVersion tulajdonságokat (ezek szerepére az előadás alapján emlékeztessünk), illetve a néhány egyéb, Table Storage specifikus tulajdonságot is (pl.: ETag az update-ekhez). Fontos, hogy minden entitásnak rendelkeznie kell egy paraméter nélküli konstruktorral is!

Mutassuk meg a TableEntity osztály szerkezetét a hallgatóknak és értelmezzük az egyes tulajdonságok szerepét! (jobb klikk 🡺 go to definition, vagy F12)

### Controller létrehozása

Hozzuk létre a controllert, amely tudja kezelni a felhasználói interakciókat! A Controllers mappához adjunk hozzá egy új controllert, használjuk az MVC5 controller – Empty sablont! A Controller neve legyen ItemController!

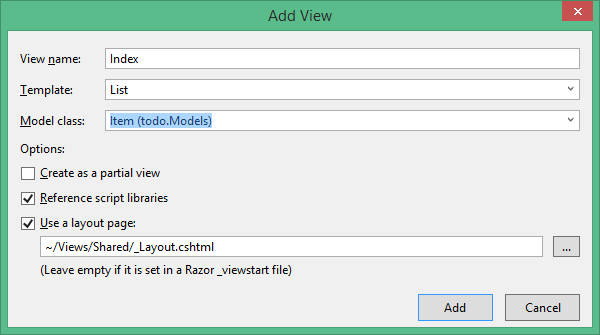
### Nézetek létrehozása

Vegyük fel a nézeteket is! 3 funkciót fogunk támogatni, amelyekhez nézetek felvétele szükséges:

* feladatok listája
* új feladat felvétele
* feladat szerkesztése

#### Listázó nézet

A Views mappában található Item mappára jobb gombbal kattintva tudunk felvenni új nézeteket az ItemsController controllerhez. Minden nézet létrehozása után töröljük ki azokat a propertyket a listából vagy az adatbekérő formról, amik a TableEntityből származnak (tehát csak a Name, Description és az IsCompleted maradjon meg)! Elsőként vegyük fel a listázó nézetet:



A listázó nézet végén módosítsuk a kódot úgy, hogy a foreach ciklus alján csak Edit opció legyen (Delete és Details nem kell), és az Edit-nek az alábbi módon generálódjon a link:

@foreach( var item in Model )

{

<tr>

<td>

@Html.DisplayFor( modelItem => item.Name )

</td>

<td>

@Html.DisplayFor( modelItem => item.Description )

</td>

<td>

@Html.DisplayFor( modelItem => item.Completed )

</td>

<td>

@Html.ActionLink( "Edit", "Edit", new { id = item.RowKey } )

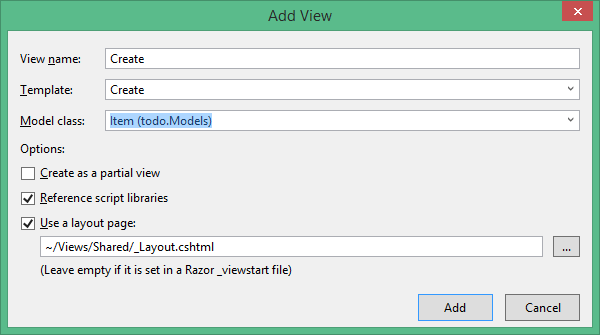
</td>

</tr>

}

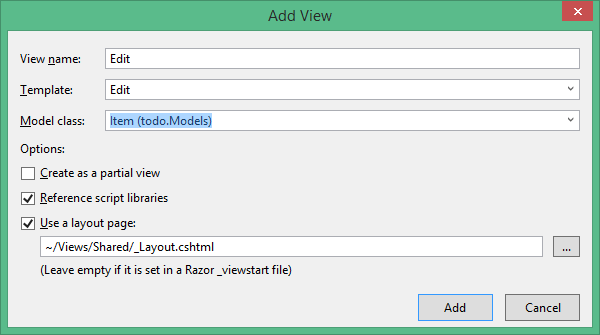
#### Új elem nézet

Az előzőhöz hasonló módon vegyünk fel még egy View-t, ezt paraméterezzük az alábbi módon:



#### Szerkesztőnézet

Végül vegyük fel a szerkesztőnézetet is:



A szerkesztőnézetet módosítsuk úgy, hogy az „Id” helyett a RowKey és a PartitionKey utazzon a rejtett mezőkben:

<h4>Item</h4>

<hr />

@Html.ValidationSummary( true, "", new { @class = "text-danger" } )

@Html.HiddenFor( model => model.PartitionKey );

@Html.HiddenFor( model => model.RowKey );

## Üzleti logika programozása

A következő részben az egyelőre üres nézetek és controller mögé bekötjük a Table Storage-os működést. Vegyünk fel a projektbe egy TableRepository osztályt! Ezen keresztül fogjuk a kommunikációt megvalósítani a Table Storage felé.

Ezután pedig az alábbi kódrészletet szúrjuk be a fájlba:

public static class TableRepository

{

private static readonly string connectionString = "";

private static CloudStorageAccount account = CloudStorageAccount.Parse( connectionString );

private static CloudTable todoTable;

private static CloudTable ToDoTable

{

get

{

if( todoTable == null )

{

var client = account.CreateCloudTableClient();

todoTable = client.GetTableReference( "TodoItems" );

todoTable.CreateIfNotExists();

}

return todoTable;

}

}

}

Magyarázat:

* A kód első sora néhány statikus mezőt tartalmaz, amiben különböző adminisztrációs adatokat kell megadnunk. Ezeket a blobos részhez hasonlóan használjuk.
* TodoTable tulajdonság: Ezen keresztül fogjuk elérni a táblát, amelyben az adatok vannak. Hasonlóan a blobos fejlesztéshez, először itt is egy kell egy kliens, ami most CloudTableClient. Ettől tudjuk elkérni a táblára mutató hivatkozást, majd létrehozni a táblát, ha még nem létezik.

Ezzel az adminisztrációs feladatokhoz szükséges metódusaink megvannak. A TodoTable tulajdonság első meghívásakor inicializálja a táblát, ha még nem létezik.

Ezután írjuk meg az alábbi metódust, amivel le tudjuk kérdezni a még be nem fejezett elemeket a gyűjteményből (szükség lesz Item osztály névterének importálására):

public static IEnumerable<Item> GetIncompleteItems()

{

TableQuery<Item> query = new TableQuery<Item>().Where

(

TableQuery.GenerateFilterConditionForBool( "Completed", QueryComparisons.Equal, false )

);

return ToDoTable.ExecuteQuery( query ).ToList();

}

A TableQuery objektumon keresztül tudunk lekérdezéseket létrehozni. A Where feltételbe szűrőket adhatunk meg, itt most a GenerateFilterConditionForBool() metódust használjuk, mert egy boolean típusú mezőre szűrünk. Ha több feltételünk is van, akkor a TableOperations enumot tudjuk használni a logikai műveletek reprezentálására.

Hozzuk létre a metódust, amellyel be tudunk szúrni egy új elemet az adatbázisba:

public static Item CreateItem( Item item )

{

item.PartitionKey = DateTime.Today.ToString();

item.RowKey = Guid.NewGuid().ToString();

TableOperation insertOperation = TableOperation.Insert( item );

ToDoTable.Execute( insertOperation );

return item;

}

Itt a TableOperation.Insert() factory metódussal egy olyan TableOperation példányt hozunk létre, ami beszúrást reprezentál. Ezt kell lefuttatnunk és paraméterül megadni a beszúrandó elemet. A beszúrás előtt beállítjuk a PartitonKey és a RowVersion tulajdonságokat. Itt emlékeztessünk ezek szerepére. Bonyolultabb esetben érdemes a PartitionKey értékét jól átgondolni (pl.: ha lenne kategória a feladatokhoz, használhatnánk azt), hiszen ennek hatása van mind a tárolásra, mind a lekérdezésre (erre is emlékeztessünk az előadás alapján).

Ezek után egészítsük ki a controllert úgy, hogy hívják meg a listázás és a létrehozás műveleteket. A korábban létrehozott ItemController kódjában az Index() metódust cseréljük le erre:

public ActionResult Index()

{

var items = TableRepository.GetIncompleteItems();

return View( items );

}

Illetve egészítsük ki a létrehozáshoz szükséges kódrészletekkel (szükség lehet a megfelelő névterek importálására):

public ActionResult Create()

{

return View();

}

[HttpPost]

public ActionResult Create(Item item)

{

if (ModelState.IsValid)

{

TableRepository.CreateItem(item);

return RedirectToAction("Index");

}

return View(item);

}

Ha most futtatjuk az alkalmazást, akkor a /Item/Index url-en tudjuk listázni az összes feladatot, a /Item/Create url-en pedig létre tudunk hozni újakat. Teszteljük az alkalmazást!

* Először nézzük meg a listát! Üresnek kell Lennie.
* Ezután hozzunk létre egy új elemet úgy, hogy a Completed értéke hamis marad! Ha most listázzuk az elemeket, akkor ezt látjuk a listában.
* Végül hozzunk létre egy olyan új elemet, amelyben a Completed értékét igazra állítjuk! Ez nem fog megjelenni a listában.

Végül vegyük fel a szerkesztéshez szükséges műveleteket! Először a TableRepository osztályba vegyük fel az alábbi két metódust:

public static Item GetItem( string rowKey )

{

TableOperation retrieveOperation = TableOperation.Retrieve<Item>( DateTime.Today.ToString(), rowKey );

var result = ToDoTable.Execute( retrieveOperation ).Result;

return (Item)result;

}

public static Item UpdateItem( Item item )

{

item.ETag = "\*";

TableOperation updateOperation = TableOperation.Replace( item );

ToDoTable.Execute( updateOperation );

return item;

}

A GetItem() metódus egy adott azonosítójú dokumentumot ad vissza. Ezt fogjuk használni arra, hogy a szerkesztőfelületen megjelenítsünk egy adott dokumentumot. Ebben a TableOperation Retrieve factory metódusát használjuk, amely elemek lekérdezésére való. Paraméterként meg kell adni a PartitionKey-t és a RowKey-t. A visszatérési érték nem csak maga az entitás, hanem néhány egyéb kiegészítő információ is; az entitás maga a Result tulajdonságban van.

Az UpdateItem() metódust fogjuk használni arra, hogy egy adott dokumentumot módosítsunk. A módosításhoz a Replace() factory metódussal hozunk létre egy műveletet. A felülíráshoz mindenképpen meg kell adni az entitás ETag-jét (ezt konkurenciakezeléshez használja az Azure). Mivel az nem utazik most rejtett mezőben, ezért ezt még “\*” értékre be kell állítanunk, végül lefuttatjuk a műveletet.

Ezeket a műveleteket tudjuk használni a szerkesztéshez. Ehhez az ItemController osztályba vegyük fel az alábbi két metódust:

public ActionResult Edit(string id)

{

Item item = TableRepository.GetItem(id);

if (item == null)

{

return HttpNotFound();

}

return View(item);

}

[HttpPost]

public ActionResult Edit(Item item)

{

if (ModelState.IsValid)

{

TableRepository.UpdateItem(item);

return RedirectToAction("Index");

}

return View(item);

}

Ezután már szerkeszteni is tudjuk az elemeket, ha az Index nézetben az Edit linkre kattintunk.

# Függelék

public static class Prompt

{

public static string ShowDialog( string text, string caption )

{

Form prompt = new Form()

{

Width = 500,

Height = 150,

FormBorderStyle = FormBorderStyle.FixedDialog,

Text = caption,

StartPosition = FormStartPosition.CenterScreen

};

Label textLabel = new Label() { Left = 50, Top = 20, Text = text };

TextBox textBox = new TextBox() { Left = 50, Top = 50, Width = 400 };

Button confirmation =

new Button() { Text = "Ok", Left = 350, Width = 100, Top = 70, DialogResult = DialogResult.OK };

confirmation.Click += ( sender, e ) => { prompt.Close(); };

prompt.Controls.Add( textBox );

prompt.Controls.Add( confirmation );

prompt.Controls.Add( textLabel );

prompt.AcceptButton = confirmation;

return prompt.ShowDialog() == DialogResult.OK ? textBox.Text : "";

}

}