Budapesti Műszaki és Gazdaságtudományi Egyetem
Virtualizációs technológiák és alkalmazásaik (VIMIAV89)

Azure alkalmazásfejlesztés

Házi feladat

Fekete Norbert (CO0DA1)
2012. december 16.

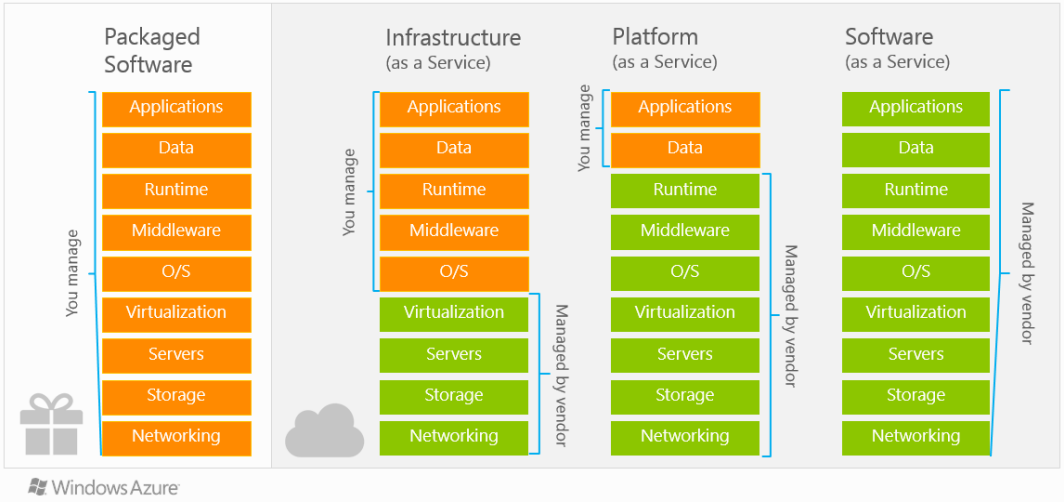
1 Bevezető

1.1 Cloud computing

Napjainkban nagyon divatosak a felhőalapú megoldások. Az is egyre elterjedtebb, hogy egy-egy nagy cég nem csak a végfelhasználók közvetlen kiszolgálására használja saját infrastruktúráját, hanem "bérbe adja" – kisebb vállalkozásoknak, akik szintén szeretnének felhőalapú szolgáltatásokat nyújtani a felhasználóknak, de az ehhez szükséges infrastruktúra kiépítése nekik túl nagy beruházást jelentene. Sőt, már az is lehetséges, hogy akár csak egy-két magánember futtasson valamilyen saját programot a felhőben.

Ha ehhez a módszerhez folyamodunk, és szoftverünk futtatásához valamely nagyobb cég infrastruktúráját vesszük igénybe (hardver, OS vagy akár futtatókörnyezet), annak megvan az az előnye, hogy annak karbantartásával, menedzselésével nem kell foglalkoznunk – végül is fizetünk azért, hogy ezt hatékonyan elvégezzék helyettünk.

Aszerint, hogy a szolgáltatást nyújtó cég mennyit vállal át a menedzselési feladatokból, a következő három kategóriát szokták megkülönböztetni: IaaS, PaaS, SaaS. Ezek viszonyát mutatja az 1. ábra.



1. ábra: Felhőalapú szolgáltatások szintjei
(kép a WATK [1] első bemutatójából)

A **SaaS (Software as a Service)** tulajdonképpen a kész terméket jelenti, azt a szolgáltatást, amit a végfelhasználó igénybe vesz. Amikor felhőre programozunk, ezt mi állítjuk elő.

A **PaaS (Platform as a Service)** lesz az a szolgáltatási szint, amit mi programozóként igénybe veszünk: kapunk egy futtatókörnyezetet, amelyen például .Net alapú kódot futtathatunk, és egy sereg olyan kiegészítő funkciót, melyek könnyebbé teszik a fejlesztést egy ilyen elosztott jellegű rendszerben. Az OS karbantartása nem a mi dolgunk, nyugodtan koncentrálhatunk a saját programunkra.

Az **IaaS (Infrastructure as a Service)** esetében csak a hardver infrastruktúrát kapjuk készen – általában egy virtuális gépet, valamilyen előtelepített operációs rendszerrel – melyen minden futó alkalmazást nekünk magunknak kell menedzselnünk.

Windows Azure a Microsoft (éppen megújulóban lévő) felhőszolgáltatása. Míg korábban főleg PaaS szintű szolgáltatásokat nyújtottak, és valószínűleg még mindig ez a fő profiljuk, mára már minden típust lefedtek. A PaaS kategórián belül pedig már igazán színes a kép: bármire legyen is igényünk – egy egyszerű weblap hosztolásától kezdve a mobil app-ok háttérszolgáltatásainak futtatásán keresztül egészen a nagy számításigényű, párhuzamosítható feladatok végrehajtásáig – szinte biztos, hogy itt találunk rá megfelelő megoldást, feltéve, hogy van elég pénzünk rá…

1.2 A házi feladat célja

E házi feladat célja az Azure PaaS szolgáltatásai által nyújtott lehetőségek megismerése volt.

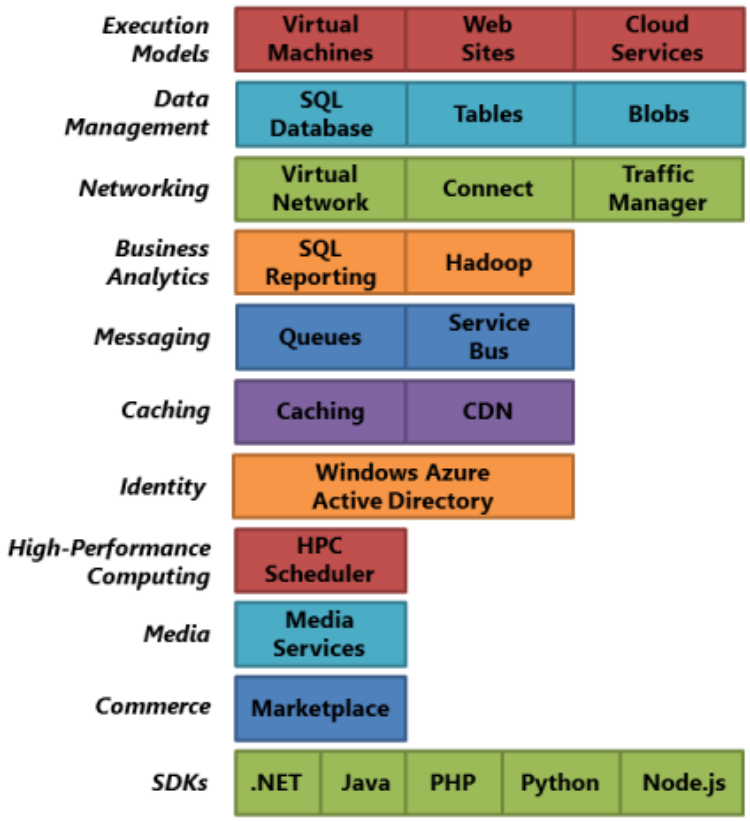
Mivel ezen belül is több különböző modell közül választhatunk (lásd következő szakasz), én eggyel, a „Cloud Services”-zel foglalkoztam a gyakorlatban is.

2 Azure PaaS

2.1 Ismerkedés az Azure platformmal

A Windows Azure egy eléggé komplex, sok egymásra épülő komponensből álló rendszer. Általában nem is egy komponensét használjuk, hanem egy kisebb készletet, összeillő elemekből. (Ezt az Azure magától is így csinálja: pl. ha létrehozunk egy sima virtuális gépet (IaaS), annak a háttértárja valójában egy (hamarosan megismert) BLOB storage-ra képződik le, mely az operációs rendszer felé normál NTFS interfészt mutat.) De gyakorlatilag szinte mindent mindennel összekapcsolhatunk, ha akarunk: ha egy virtuális gépre magunk telepítünk bármilyen http képes programot, az kapcsolatba léphet az Azure szolgáltatásaival, például valamilyen tárhellyel. Sőt, megfelelő beállítások mellett ezt saját, fizikai („on-premises”) gépünkről is megtehetjük.

A komponensek egy lehetséges csoportosítását mutatja a 2. ábra.

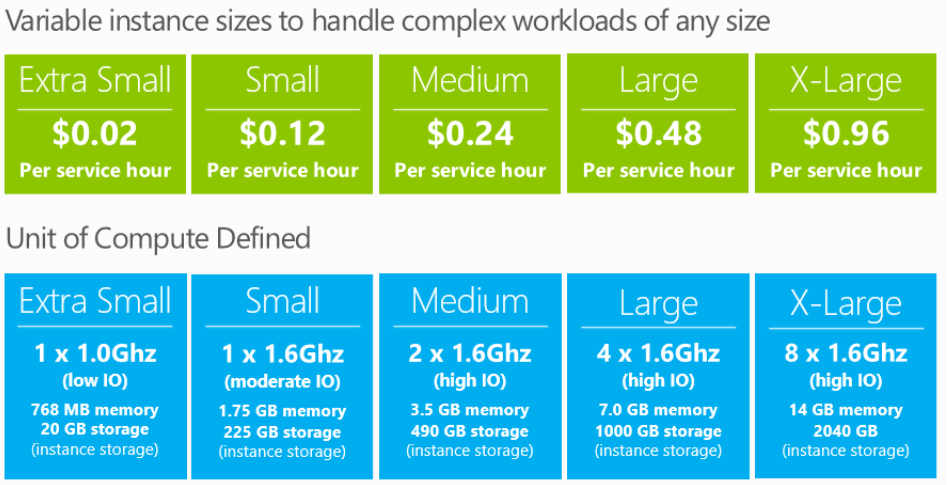


2. ábra: Az Azure komponensei [2]

Ha az Azure-ra szeretnénk fejleszteni, az első dolog, amit el kell döntenünk, hogy az ábra legfelső sorában feltüntetett „futtatási modellek” közül melyiket választjuk. (Természetesen készíthetünk több, különböző futtatási modellre épülő alkalmazást, melyek együttműködnek…) Hogy melyiket érdemes választani, az a feladatunktól, illetve az igényeinktől függ:

- Ha csak egy egyszerűbb weboldalt szeretnénk futtatni, bőven elég a **Web Sites** szolgáltatás. Ez pontosan annyit tud, amennyit a neve is sugall: kapunk egy IIS szervert, melyre weboldalakat telepíthetünk. Képes az ASP.Net, PHP és Node.js nyelvű szerveroldali logika futtatására, illetve létrehozhatunk mellé egy relációs adatbázist (MS SQL vagy MySQL) vagy valamilyen Azure storage-ot, amelyet elér. Összességében tehát már itt is elég nagy szabadságot kapunk.
- Ha ennél mégis többre vágyunk, főleg ha egy komolyabb, strukturáltabb rendszert szeretnénk összerakni, amiben több csomópontunk van, több különböző szerepkörrel, akkor a **Cloud Services**-t kell választanunk. Az Azure itt alapvetően két csomóponttípust különböztet meg: **Web Role**-okat (webes felülettel (IIS) rendelkező, „front-end jellegű” csomópontok) és **Worker Role**-okat (webes felület nélküli, háttérfolyamat jellegű csomópontok). Mindkét típusból tetszőleges számút létrehozhatunk, és az így létrehozott szerepkörökből tetszőleges számút példányosíthatunk a felhőben.
- Ha nem elég, hogy tetszőleges kódot futtathatunk egy előre konfigurált futtatókörnyezetben, marad a **Virtual Machines** (ez az IaaS-hez tartozik): azaz egyszerű virtuális gépeket kérünk, melyekre minden szükséges eszközt, futtatókörnyezetet mi telepítünk, és tetszőlegesen konfigurálhatunk.

A csomópontok méretét mindhárom modellben azonos opciók közül választhatjuk ki „Extra Small”-tól „Extra Large”-ig (lásd: 3. ábra). Ezzel kapcsolatban még annyit érdemes megjegyezni, hogy az Extra Small mérethez megosztott CPU tartozik, tehát a fizikai gép a miénkkel párhuzamosan más virtuális gépeket is futtat, míg a Small mérettől kezdve már saját, dedikált CPU-t kapunk.



3. ábra: Node méretek az Azure-ban [1]

Marketing oldalról három fő érv szokott felbukkanni az Azure mellett, szinte minden anyagban:

- *Nyílt:* sok különböző (nem csak Microsoftos) technológiát támogat, a .Net mellett számos más nyelvhez készítettek API-t, melyek forráskódja ráadásul nyíltan elérhető. Maga az Azure egyébként a HTTP fölött egy RESTful protokollt használ, amely szintén nyílt, így nincs akadálya annak, hogy egy vállalat saját igényeinek megfelelően újabb nyelv támogatásával egészítse ki a platformot. Ráadásul ezen a protokollon keresztül az Azure kommunikálhat on-premises gépekkel, vagy akár egy másik, konkurens felhőszolgáltató virtuális gépeivel („no lock-in”).
  A tényeket tekintve, szerintem ez az állítás meglepő módon igaz (legalábbis első ránézésre).

- *Rugalmas:* ezt a sok különböző komponens egymásra épülése és kombinálhatósága szerintem lehetővé teszi.
- *Pay-as-you-go* modell: állandó havidíj helyett lehetőség van egy olyan számlázási módra, amiben folyamatosan mérik, hogy az egyes erőforrásokból mennyit fogyasztottunk, majd a hónap végén ennek megfelelően kell fizetni, mint pl. egy hagyományos közmű esetében. Így az ember csak azért fizet, amit ténylegesen használ – az más kérdés, hogy mennyit… (Arra viszont nem árt figyelni, hogy a „csak leállított", és nem törölt virtuális gépért ugyanúgy fizetni kell, mint a 100% CPU terheltséggel futóért! A fizikai gép egész addig allokálva van nekünk, amíg a virtuális gépet ki nem töröljük – ami viszont nem jelent nagy problémát, mivel a háttértár ettől még megmarad.)
  Az mindenképpen előnye ennek a megoldásnak, hogy az ügyfél nincs előzetes megállapodáshoz kötve, az indulás után ráér kikísérletezni, hogy pontosan mennyi erőforrásra van szüksége. Az Azure portálon találunk egy kalkulátort is, ami segít megbecsülni a költségeket [6].

A következőkben a Cloud Services-zel fogunk foglalkozni, illetve az adattárolási lehetőségekkel, amelyek elérhetők a többi futtatási modellből is.

2.2 Cloud Services

Ahogy az már elhangzott, ebben a futtatási módban kész „platformot" kapunk, azaz speciális, előre felkonfigurált virtuális gépeket, melyek a mi kódunkat futtatják. Ezek egyébként majdnem „normális", Windows Servert futtató virtuális gépek, pl. egy .Net futtatókörnyezettel; ha engedélyezzük, be is léphetünk rájuk Remote Desktoppal, de egy dolgot nem szabad elfelejteni: ezeknek a virtuális gépeknek egy célja van, a mi kódunk futtatása. A háttértárjuk nem perzisztens! Azaz: írhatunk és olvashatunk fájlokat a merevlemezen, de ha valamilyen okból az a fizikai gép, ami a kódunkat futtatja, kiesik, az Azure gondolkodás nélkül allokál nekünk egy másikat, amelyre ugyan azt a sablont (OS, .Net stb.) húzza, mint amelyikkel kezdtünk, rátölti a kódunkat, és elindítja – pótolva a kiesett gépünket, de elfelejtve minden adatot, amit annak a háttértárjára írtunk.

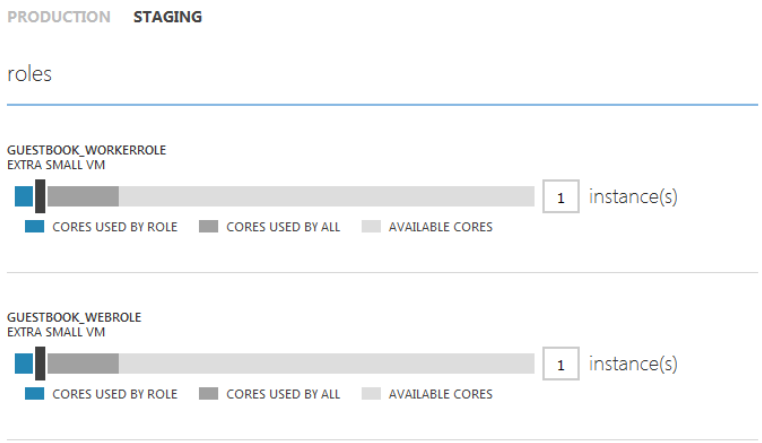
A kódunknak tehát olyan értelemben „állapotmentesnek" kell lennie, hogy semmilyen fontos adatot nem tárolhat a merevlemezen. Míg hagyományos programozás esetén tisztában vagyunk azzal, hogy a RAM tartalma „elszállhat", ezért a fontos adatokat a diszkre írjuk, itt, amikor felhőre programozunk, tisztában kell lennünk azzal, hogy a diszk tartalma is mulandó, a fontos adatokat az Azure által nyújtott valamely adattároló szolgáltatásnak kell elküldenünk, hogy az rögzítse. Innentől kezdve az Azure felelőssége az adataink biztonságos tárolása.

Ahogy már említettük, a Cloud Services modellben lehetőségünk van több szerepkör létrehozására – azaz többféle végrehajtandó kódot írunk, különböző feladatokra, majd mindegyiket tetszőleges számban „példányosítjuk", vagyis tetszőleges számú virtuális gépet bízunk meg a futtatásukkal.

Ez egy újfajta skálázási lehetőséget jelent: amellett, hogy a különböző teljesítményű virtuális gépek közül történő választással „vertikálisan" tudjuk skálázni az alkalmazásunkat, így, a példányszámok növelésével „horizontálisan" is skálázhatjuk azt.

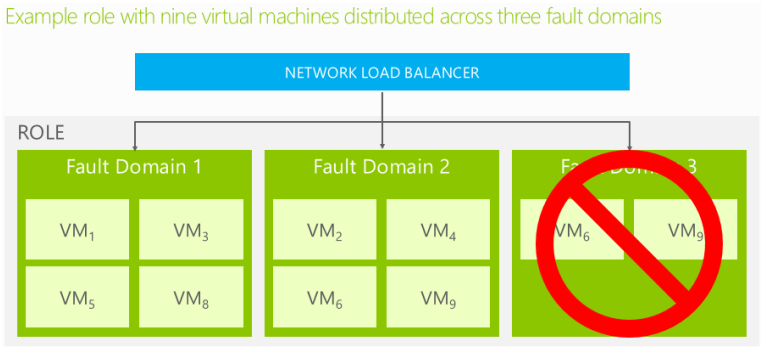
Ehhez persze ügyesen kell megírnunk a programunkat: innentől kezdve nem elég, hogy a kódunkat az előbbi értelemben állapotmentesre írjuk; úgy kell megírnunk, hogy ha egy szerepből több példányunk fut, mindegy legyen, hogy egy külső kérés melyikhez fut be, a rendszer hatékonyan tudjon rá reagálni. (A bejövő kérések elosztásáról ugyanis az Azure load balancer-e gondoskodik. A felhasználó csak egy logikai címet lát, amiről a load balancer átirányítja valamelyik virtuális gépre, melyen a mi programunk egy példánya fut. Akár az is megtörténhet, hogy ugyanattól a felhasználótól az egymás utáni kéréseket különböző példányok kapják meg. – Bár gondolom, ez a probléma nem csak cloud környezetben merül fel, én akkor szembesültem vele, amikor elkezdtem végiggondolni az első Azure-os programom felépítését…)

Ha ezt sikerül megoldani, akkor az alkalmazásunk felfelé skálázása a terhelés esetleges megnövekedése esetén már gyerekjáték: csak belépünk az Azure menedzsment portálra, és feljebb toljuk a csúszkát, ami a példányok számát szabályozza (4. ábra). (Az egyes gépek teljesítménye szintén a portálról állítható. Annyi megkötés van, hogy az egy szerephez tartozó valamennyi példány ugyanolyan teljesítményű kell, hogy legyen.) Az ábrán még egy kényelmes funkció látszik: az éles rendszer mellett lehetőségünk van egy tesztrendszert („staging") is működtetni, majd ha úgy gondoljuk, hogy az új verzió készen áll, gombnyomásra kicserélhetjük az éles rendszert („VIP swap" – Virtual IP address swap, oda-vissza működik).



4. ábra: Role példányszámok beállítása a management portalon

Ha minden szerepkört ilyen elosztott módon futtatunk, az a hibatűrést is növeli. Mindenből legalább 2 példány futtatása esetén a Microsoft 99.95%-os rendelkezésre állást garantál. Ezt úgy próbálják elérni, hogy a különböző példányokat különböző fault domain-ekbe helyezik (5. ábra).

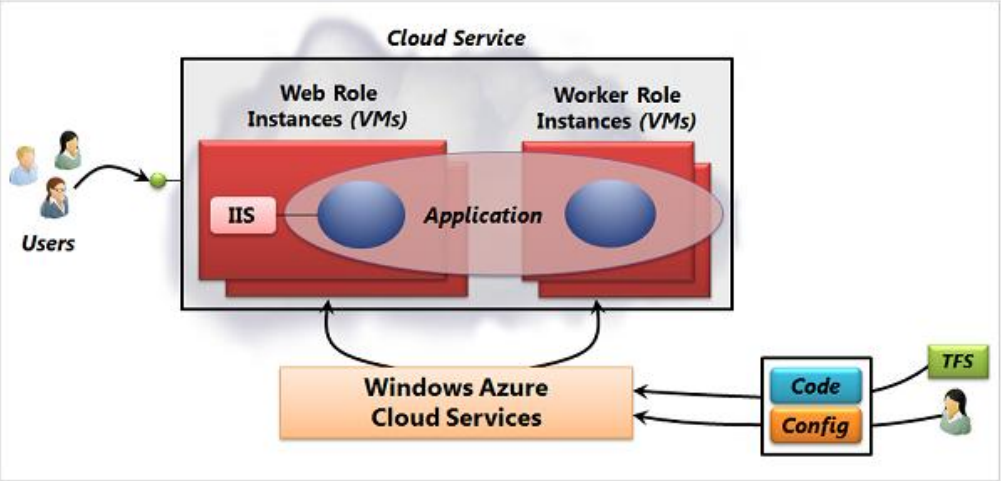


5. ábra: Hibatűrés és terheléselosztás [1]

Az általunk létrehozható szerepköröknek az Azure-ban két fő típusa van:

- **Web Role:**
  Olyan csomópont, ami Microsoft IIS-t futtat. Ezen általában az alkalmazásunk felülete található, egy webhely, amit a felhasználó egy böngészővel megtekinthet. (Például egy ASP.Net weboldal, de WebService-eket is futtathatunk rajta.)
- **Worker Role:**
  Háttérműveletek végrehajtására való csomópont. Nincs rajta IIS és ASP.Net, funkcióját a Windowson futó service-ekhez szokták hasonlítani. Olyan hosszabb futási idejű feladatok végrehajtására való, amikkel nem szeretnénk a webes felületet futtató csomópontokat terhelni. (Programozni is egyszerűbb, pl. nem kell az ASP.Net threadpooljával és az osztályaink többszöri példányosításával törődnünk – van egy belépési pontja a programnak, az egyszer meghívódik, onnantól mi parancsolunk.) Gyakori megvalósítása egy végtelen ciklus, ami egy Azure-os üzenetsort polloz, feladatokra várva.

A Cloud Services felépítéséről mutat egy áttekintő képet a 6. ábra.



6. ábra: Cloud Services felépítése [3]

2.3 Adattárolás

A felhőben az adattárolás megbízható megoldása különösen fontos, mivel számításokat végző csomópontjaink önmagukban nem rendelkeznek megbízható tárhellyel. Ugyanakkor sok van belőlük, és általában nagy mennyiségű adattal dolgoznak, ezért fontos a többszörös hozzáférés és a konzisztencia kezelés hatékony megvalósítása is.

Az Azure-ban a megbízhatóságról redundáns tárolás gondoskodik: a rendszer minden adatunkat három példányban menti, úgy, hogy az egyes példányok különböző fault domain-ekbe kerüljenek (egy datastore-on belül). Ez mindig adott, ezt a funkciót ki se tudjuk kapcsolni. Ha az elsődleges példány kiesik, automatikusan valamelyik replika kerül a helyébe, és közben megkezdődik a kiesett példány pótlása.

A helyi redundáns tároláson kívül kérhetünk ún. Geo Redundant Storage-ot (GRS) is. Ilyenkor az adatainkat egy másik datacenterbe is replikálni fogják, amely azonos régióban van ugyan az elsővel, de attól garantáltan legalább 500 mérföld távolságra.

2.3.1 BLOB Storage

A BLOB (Binary Large OBject) az adattárolás legegyszerűbb módja: egy egyszerű bitfolyam, különösebb segédstruktúrák nélkül. Egy fájlhoz hasonlóan kezelhetjük; vagy valami nagyméretű adatfolyamot mentünk bele (pl. képfájl, videó), vagy mi magunk alakítjuk ki a struktúráját, és ily módon tároljuk benne kisebb adatok sorozatát.

A BLOB-ok ún. Container-ekbe rendeződnek. Ezek csak logikai névterek, amikkel rendszerezhetjük őket. Létrehozás után a BLOB-okra az URL-ükkel hivatkozhatunk, melynek szerkezete a következő:

```
http://<StorageAccount>.blob.core.windows.net/<Container>/<BlobName>
```

A <StorageAccount> egy hozzáférési azonosító az Azure tárhely szolgáltatásaihoz. Segítségével nem csak a BLOB-okat kezelhetjük, hanem a Táblákat és Üzenetsorokat is (lásd következő szakaszok). Ez határozza meg, hogy konkrétan melyik datacenterben kívánjuk tárolni az adatainkat. (Célszerű a felhasználókhoz minél közelebbit választani.)

A BLOB-oknak két fajtája van:

- **Block BLOB:**
  Blokkokra van osztva, és hasonlóan a hagyományos, láncolt szerkezetű fájlokhoz, az elejétől a végéig haladva tudjuk írni/olvasni. Azonban ha egy blokk átvitele sikertelen, nem kell elölről kezdenünk a teljes műveletet, elég a sérült blokktól újraindítani. Max. mérete 200 GB.
- **Page BLOB:**
  Lapokra van osztva, és véletlenszerű hozzáférésre van tervezve, tehát tetszőleges pontján írható/olvasható. Érdekesség, hogy az Azure VM-ek, NTFS-re képezve, ezt használják háttértárnak. Max. mérete 1 TB.

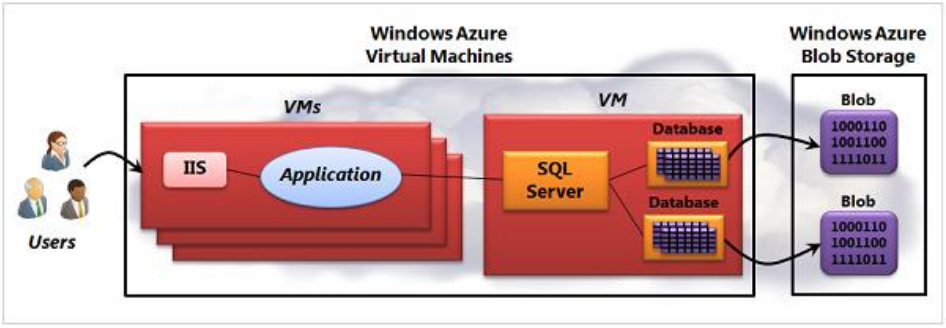
A BLOB-okra alkalmazható a Content Delivery Network (CDN) szolgáltatás is, ami a gyorsabb elérés érdekében képes a statikus adatokat a világ több pontján cache-elni. (Pl. Youtube-hoz hasonló alkalmazásokhoz.)

2.3.2 Relációs adatbázisok (SQL)

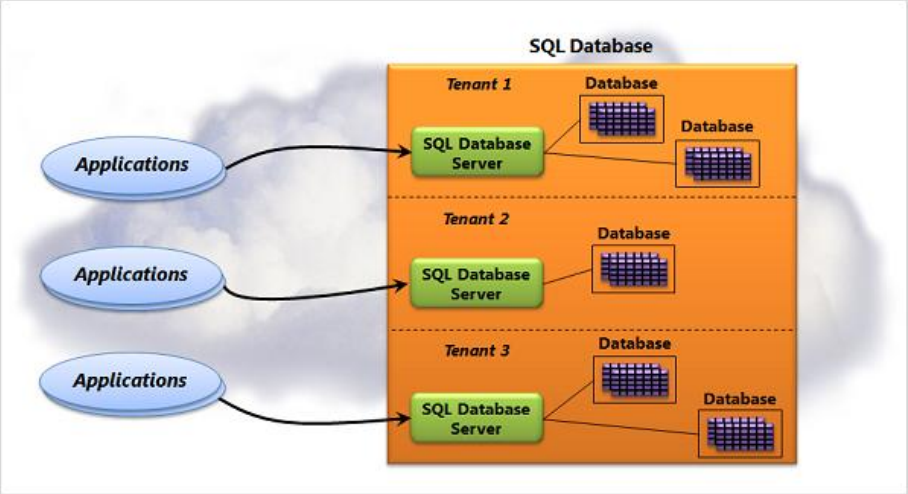
Ha strukturáltan szeretnénk adatokat tárolni, és azon kifinomult lekérdezéseket futtatni, fordulhatunk a már jól bevált SQL adatbázisokhoz.

Erre is több lehetőségünk van:

- Futtathatunk egy saját SQL Servert (ill. bármilyen adatbázis-kezelőt) egy sima IaaS alapú virtuális gépen (7. ábra). Erre van is egy kész VM sablon, így a telepítés néhány lépését megspórolhatjuk, de gyakran még így sem optimális ez a megoldás. (Akkor hasznos, ha az SQL szerver minden egyes paraméterét magunk szeretnénk finomhangolni.)
- Használhatjuk az **SQL Database** nevű szolgáltatást, ami egy (Azure által menedzselt) megosztott környezet, melyen belül minden felhasználó saját logikai adatbázis-kezelőt kap (8. ábra).
  Egy ilyen logikai adatbázis legfeljebb 150 GB adatot tartalmazhat. Ha ezt szeretnénk túllépni, egy *Federation* nevű opció áll rendelkezésünkre, ami több logikai adatbázis együttműködését jelenti. Ezenkívül a hibatűrés növelése érdekében lehetőség van az adatbázisok szinkronizálására *(SQL Data Sync)* a különböző datacenterek között, vagy akár a vállalati on-premises SQL Server és az SQL Database között. Különböző jelentések készítéséhez szintén támogatást nyújt a rendszer.
  Amit hátrányként kiemelnek, az a megosztott környezetből adódó ingadozó teljesítmény. Ha ez probléma, az előző megoldást kell választani.



7. ábra: Saját SQL Server futtatása Azure VM-en [4]



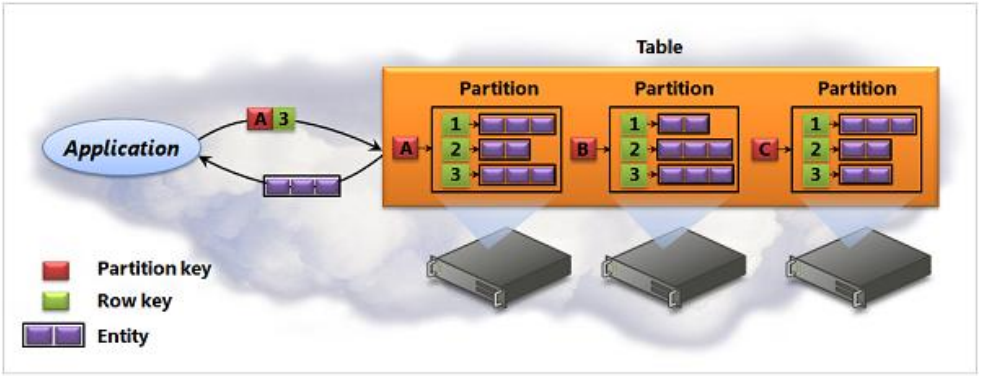
8. ábra: SQL Database - megosztott környezet [4]

2.3.3 Table Storage

Az Azure Table Storage neve elsőre megtévesztő, mert itt nem a relációs adatbázisok tábláiról van szó. Ez egy NoSQL megközelítés, ami gyakori a felhőszolgáltatásoknál.

Működése a kulcs-érték párok elvén alapszik, néhány kiegészítéssel:

- Egy-egy ilyen adattábla ún. partíciókra *(Partition)* van osztva. A partíciók különböző gépeken is lehetnek, ezzel segítve a terheléselosztást. Az egyes partíciókat a **Partition Key** azonosítja.
- A partíciókon belül tetszőleges számú entitás található, melyeket a **Row Key** azonosít (az adott partíción belül). A partíció az elemeket Row Key szerint rendezve tárolja.
- A Partition Key és a Row Key együtt alkotják a Primary Key-t, ami egyértelműen azonosít egy entitást a táblán belül.
- Egy entitáshoz legfeljebb 255 típusos tulajdonság *(Property)* tartozhat. (Ebből 3 rendszer számára fenntartott, tehát 252 marad nekünk.) Együttes méretük legfeljebb 1 MB lehet.
- Nincs fix séma: az entitások akár egy partíción belül is rendelkezhetnek különböző számú és típusú tulajdonsággal.



9. ábra: A Table Storage szerkezete [4]

Ez a szerkezet lehetővé teszi, hogy egy tábla hatalmas (akár 100 TB-os) legyen, mégis hatékonyan érhessük el. Ugyanakkor jár némi megkötéssel:

- Nincs lehetőség változatos lekérdezésekre, az entitásokat csak a kulcsaikon keresztül érhetjük el.
- Egy tranzakció csak egy-egy partíción belül lehet atomi.
- Nincs támogatás a táblák közti join művelethez.

Tehát a táblák olyan esetekben hasznosak, amikor nincs szükség bonyolult relációs lekérdezésekre, egyszerűen csak szeretnénk elérni vagy frissíteni egy-egy adatot. – Az egyszerű, Primary Key-en keresztül történő lekérdezéshez képest azért még van egy kis játékterünk: Igaz, a Partition Key-t minden lekérdezésben egyértelműen meg kell adni, de mivel a partíción belül az entitások Row Key szerint rendezve vannak, kijelölhetünk Row Key tartományokat, amelyeken belül az összes entitást lekérjük. (Ezenkívül, ha ily módon, egy partíciónak több elemét kérjük le – akár az összeset –, a lekérdezés eredménye mindig rendezett lesz.)

Fórumokon kritikaként olvastam, hogy szemben az Amazon EC2-vel, az Azure táblák még nem támogatják a másodlagos indexeket, de állítólag már ez is tervben van…

A táblákban az adatok konzisztenciáját egy optimista időbélyeges módszerrel oldották meg: minden entitásnak van egy alapértelmezett tulajdonsága: a TimeStamp. Ez mindig az utolsó commitolt írás időpontját tárolja. Csak olyan frissítést lehet commitolni, amelyben a TimeStamp értéke nem régebbi a táblában lévőnél. [7]

A táblákat egy a BLOB-okéhoz hasonló URL-lel lehet elérni, szintén a Storage Account-on keresztül:

```
http://<StorageAccount>.table.core.windows.net/<TableName>
```

2.3.4 Message Queue

Bár ez inkább a kommunikációhoz tartozik, ez is egyfajta tárhely. A Message Queue egy egyszerű FIFO rendszerű üzenetsor, melyen keresztül string-eket küldhetünk. Leggyakrabban egy Web Role által előállított feladatok tárolására használják, amik arra várnak, hogy egy Worker Role feldolgozza őket.

Üzenetküldésre azért fontos ezt használnunk, mert perzisztens – szemben egy olyan várakozó sorral, amit esetleg a Worker Role-on belül tárolnánk.

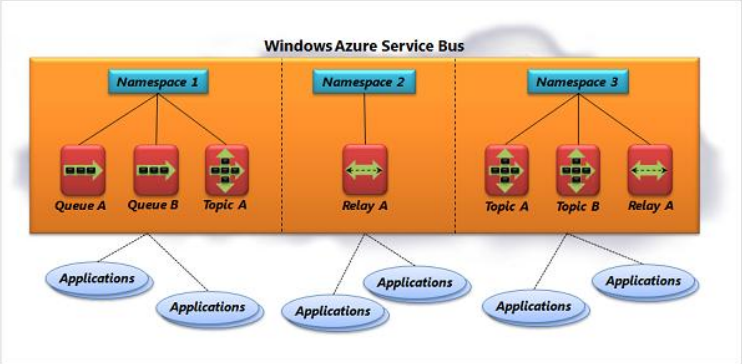
Szintén a Storage Account-on keresztül érhetjük el:

```
http://<storage account>.queue.core.windows.net/<queue>
```

2.4 Service Bus

A hatékony kommunikációt az egyes node-ok között nem mindig lehet megvalósítani az egyszerű üzenetsorokkal. Ezért röviden megemlítem, hogy egy sokkal kifinomultabb megoldást kínál a Service Bus (10. ábra). Három fő üzemmódja van:

- Queue: Az üzenetsorokhoz hasonló, egyirányú kommunikáció, perzisztens tárral.
- Topic: Publisher-Subscriber mintájú kommunikáció, perzisztens tárral. A subscriberek szűrési feltételeket is kiköthetnek az üzenetekre.
- Relay: Kétirányú kapcsolat, perzisztens tár nélkül.



10. ábra: Sevice Bus [5]

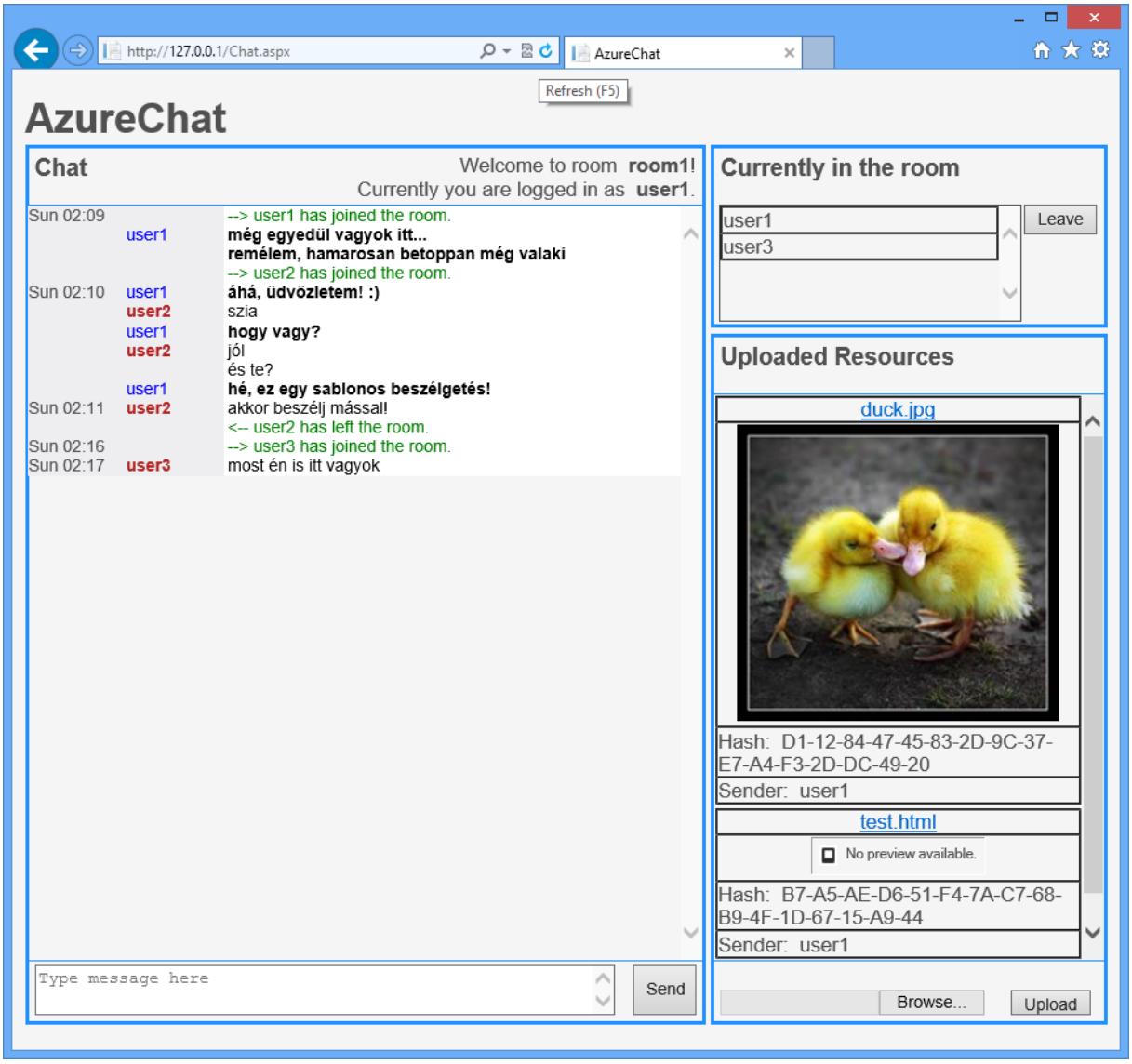
3 Első Azure programom

3.1 A kitűzött cél – az elért eredmény

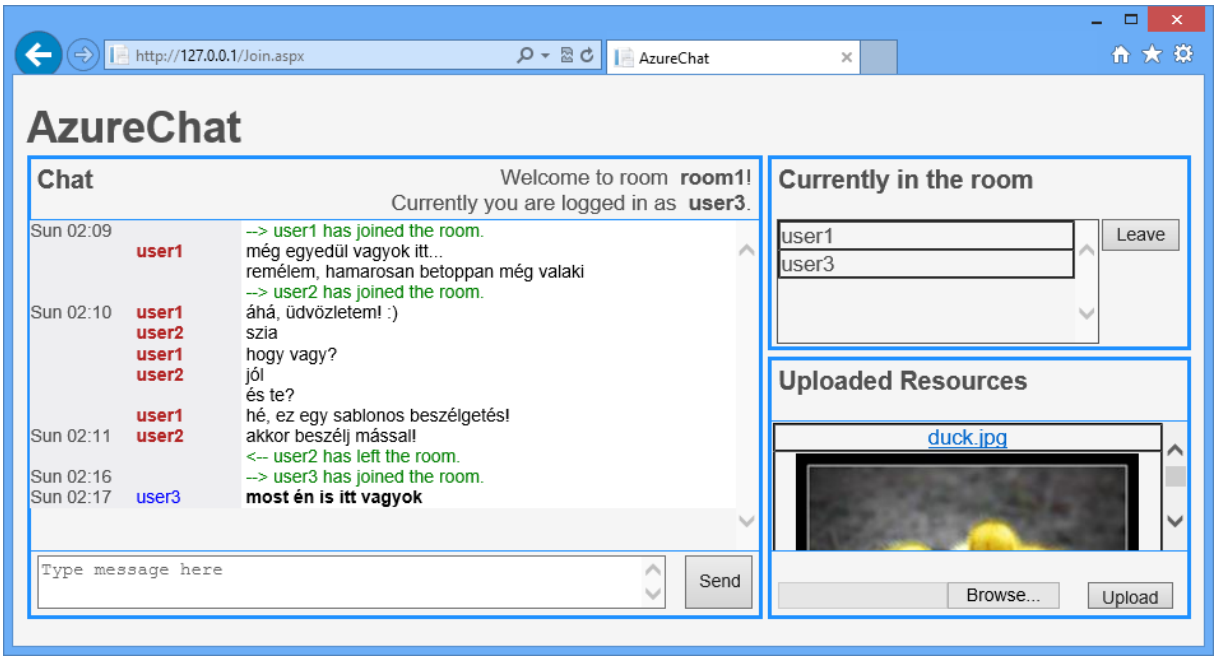
Miután megnéztem a WATK első Cloud Services használatát bemutató Hands-on Lab-ját (GuestBook-os példa), azt találtam ki, hogy egy webes chat programot fogok készíteni, valami egyszerű fájlmegosztó rendszerrel – mivel ez ránézésre eléggé hasonlít a WATK-es példához.

Sajnos rá kellett jönnöm, hogy ezt a feladatot csak azért gondoltam egyszerűnek, mert nem voltam tisztában a web fejlesztés nehézségeivel… Így a program nagyjából működik ugyan, de egyelőre kicsit próbálgatás jellegű, vannak benne kevéssé szép megoldások, és még messze nem hibamentes… De ismerkedésnek szerintem egész jó.

Hogy hogyan is képzeltem el a felületet, most egy screenshottal mutatom be (11. ábra, 12. ábra).



11. ábra: Első Azure programom - user1



12. ábra: Első Azure programom - user3

A program tehát azt tudja, hogy különböző neveken különböző szobákba jelentkezhetünk be, ahol csetelhetünk egymással, és feltölthetünk tetszőleges fájlokat a „közösbe”, melyeket a szobán belül mindenki elér. A fájlokat és az üzeneteket is az Azure tárolja, így amikor valaki utólag bejelentkezik, minden korábbi üzenetet megkap, mondjuk egy hétre visszamenőleg. (Egyszerűen kiegészíthető egy felületelemmel, ami megmondja, pontosan milyen régről szeretnénk látni az üzeneteket.) Azt is látjuk, jelenleg kik vannak a szobában.

A feltöltött fájlokról – ha képek – készítünk egy thumbnailt, ami megjelenik, illetve minden fájl esetében generálunk egy MD5 hashkódot, hogy fontos fájlok esetén ellenőrizhessük, sikeres volt-e az átvitel.

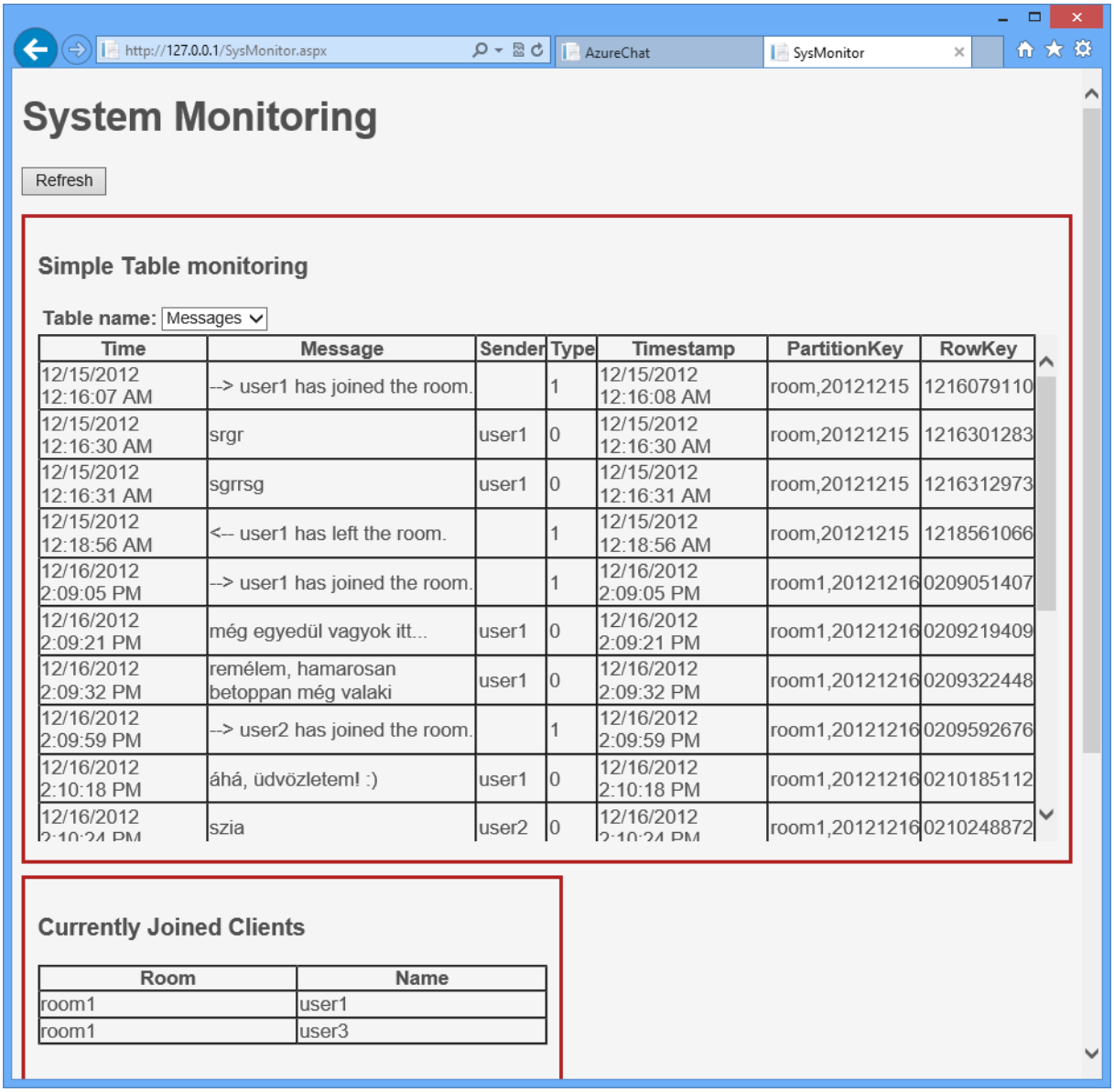
A felhasználókat a nevük és a szobájuk együttesen azonosítja.

Aktuális hiányosságok:

- Jelenleg csak a „Leave” gombbal lehet kilépni. Ha valaki szimplán bezárja az ablakot, a felhasználója bent ragad.
- A feltöltött fájlok listája egyelőre nem frissül automatikusan. Ezért a lista alján van egy „Refresh” gomb.
- Ha valaki a teljes oldalt frissíti a böngészőből, a rendszer kidobja, és a felhasználója bent ragad.
- Még nem implementáltam semmilyen tisztító algoritmust, ami mondjuk egy hét után törölné a felgyűlt üzeneteket – bár ez nem lenne nehéz.
- Jelen formájában csak akkor működik, ha egy darab Web Role példány van.

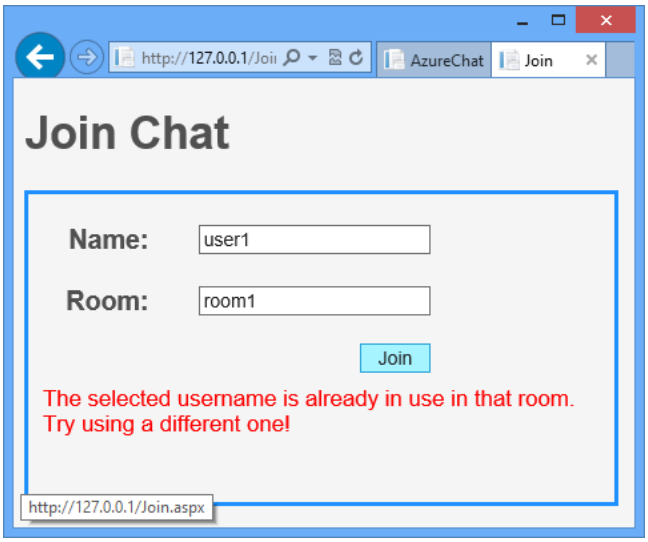
Továbbá valószínűleg „ágyúval lövök verébre”, amikor az üzeneteket Table Storage-ban tárolom, de hát végül is ez csak egy tesztprogram…

A csetes felülethez még két kiegészítő oldal tartozik: egy nagyon egyszerű bejelentkeztető ablak (15. ábra), és egy monitorozó (félkész), amivel a táblák aktuális tartalmát figyelhetjük (11. ábra, 12. ábra).



13. ábra: System Monitoring 1

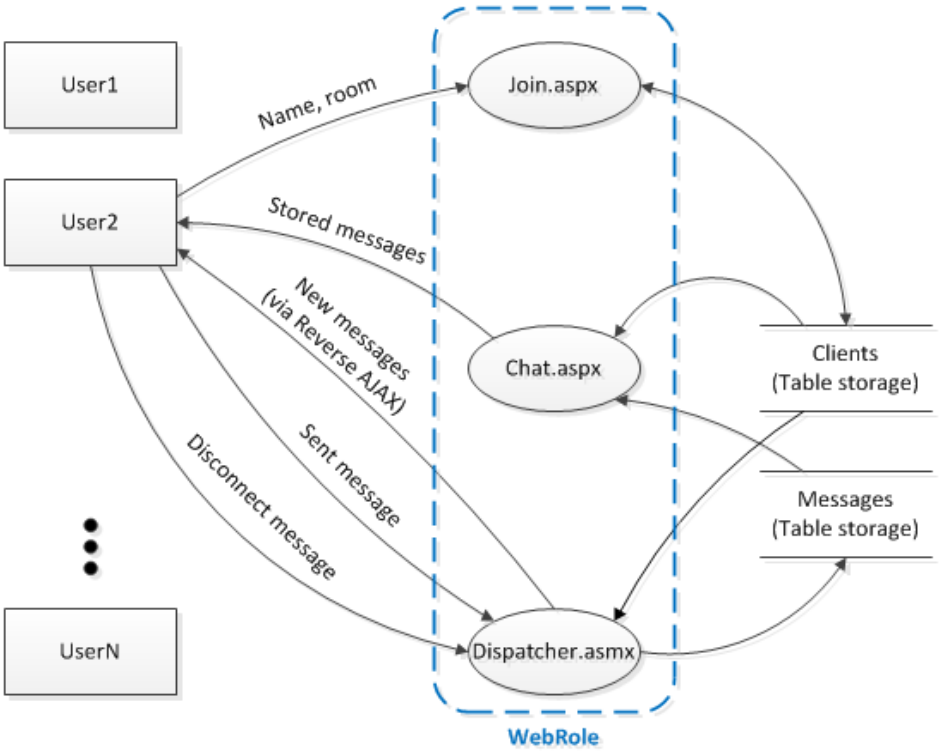
14. ábra: System Monitoring 2



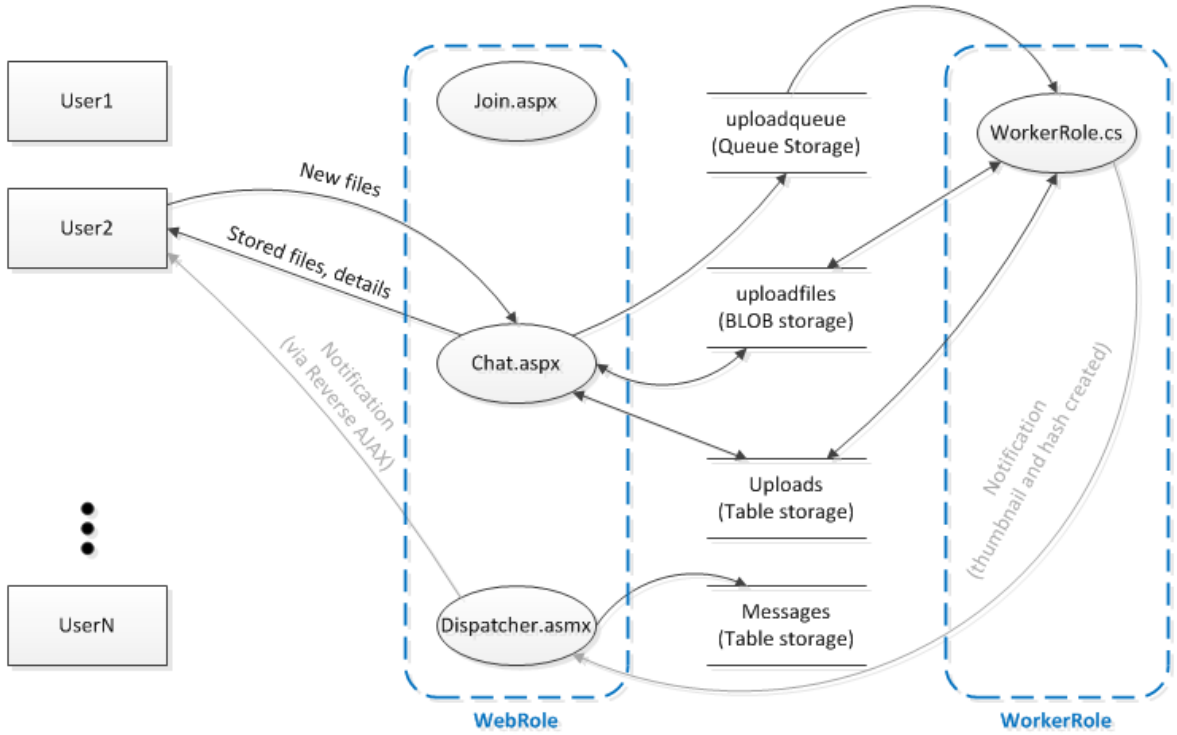
15. ábra: Bejelentkező képernyő

3.2 A program komponensei

A program szerkezetét két (elnagyolt) adatfolyam ábrával próbálom szemléltetni. Az egyik az üzenetküldés működését mutatja (16. ábra), a másik a fájlmegosztás működését (17. ábra). A szürkével jelölt részek csak tervezettek.



16. ábra: AzureChat üzenetküldés DFD



17. ábra: AzureChat fájlmegosztás DFD

Bejelentkezésnél a felhasználó a *Join.aspx*-re navigál, ahol beírja a választott nevét és a szobát, ahova be kíván lépni. A *Join.aspx* ellenőrzi a *Clients* táblában, hogy van-e azonos nevű felhasználó a szobában, és ha nincs, belépteti: bejegyzi az adatait a *Clients* táblába, majd (szerver oldalon) átadja a vezérlést a *Chat.aspx*-nek. Ekkor a *Chat.aspx* lekérdezi az adott szobában lévő, korábbi üzeneteket a *Messages* táblából, a feltöltött fájlokat az *Uploads* táblából, és a szobában lévő további felhasználók nevét a *Clients* táblából. Ezekből az adatokból előállítja a csetelős weblapot az eddig elhangzott üzenetekkel, és elküldi a felhasználónak. (Továbbá készít egy bejegyzést a *Messages* táblába arról, hogy a felhasználó belépett.)

Miután a lapot betöltötte a böngésző, a weblapon található JavaScript kód elindul, és meghívja a *Dispatcher.asmx* nevű ASP Web Service egy metódusát, ami a Reverse AJAX technikát [8] megvalósítja. A metódus addig nem tér vissza, amíg új üzenet nem jött a szobába. Ha ez megtörténik, a metódus visszatér az üzenettel, a JavaScript hozzáadja a weblap tartalmához, majd újra meghívja azt a bizonyos metódust.

Ez így megy egészen addig, amíg a felhasználó rá nem klikkel a „Leave” feliratú gombra, ami üzen a *Dispatcher.asmx*-nek, hogy lecsatlakozunk. Ekkor a *Dispatcher.asmx* törli a felhasználót a *Clients* táblából (és feljegyzi a *Messages* táblába, hogy elhagyta a szobát).

A rendszerüzenetek külön típust képeznek. Ezért amikor valaki be- vagy kilép a szobából, a JavaScript látja, hogy valami történt, ezért kezdeményez egy frissítést a felhasználólistán (ami egy ASP UpdatePanel-ben van benne, így az oldal többi részétől elkülönülve tud frissülni).

(Hasonló mechanizmussal lehetne automatikusan frissíteni a fájlok listáját is.)

Fájlfeltöltésnél a *Chat.aspx* fogadja az új fájlt. Kér egy új BLOB tárhelyet, ahova feltölti. Bejegyzi a fájl alapvető adatait az *Uploads* táblába, majd küld egy üzenetet (az *Uploads*-béli bejegyzés kulcsát) a *Worker.cs*-nek, mégpedig az *uploadqueue*-n keresztül.

A kapott adatok alapján a *Worker.cs* generál egy thumbnail képet (ha tud), amit szintén feltölt a BLOB-ba, generál egy MD hashkódot is, majd, ha mindezzel kész van, frissíti a fájl adatait az *Uploads* táblában. (És itt sajnos a fogadó felek kézi frissítésre szorulnak, de akkor megkapják az új adatokat.)

3.3 Megvalósítás részletei

3.3.1 Visual Studio – A segédeszköz

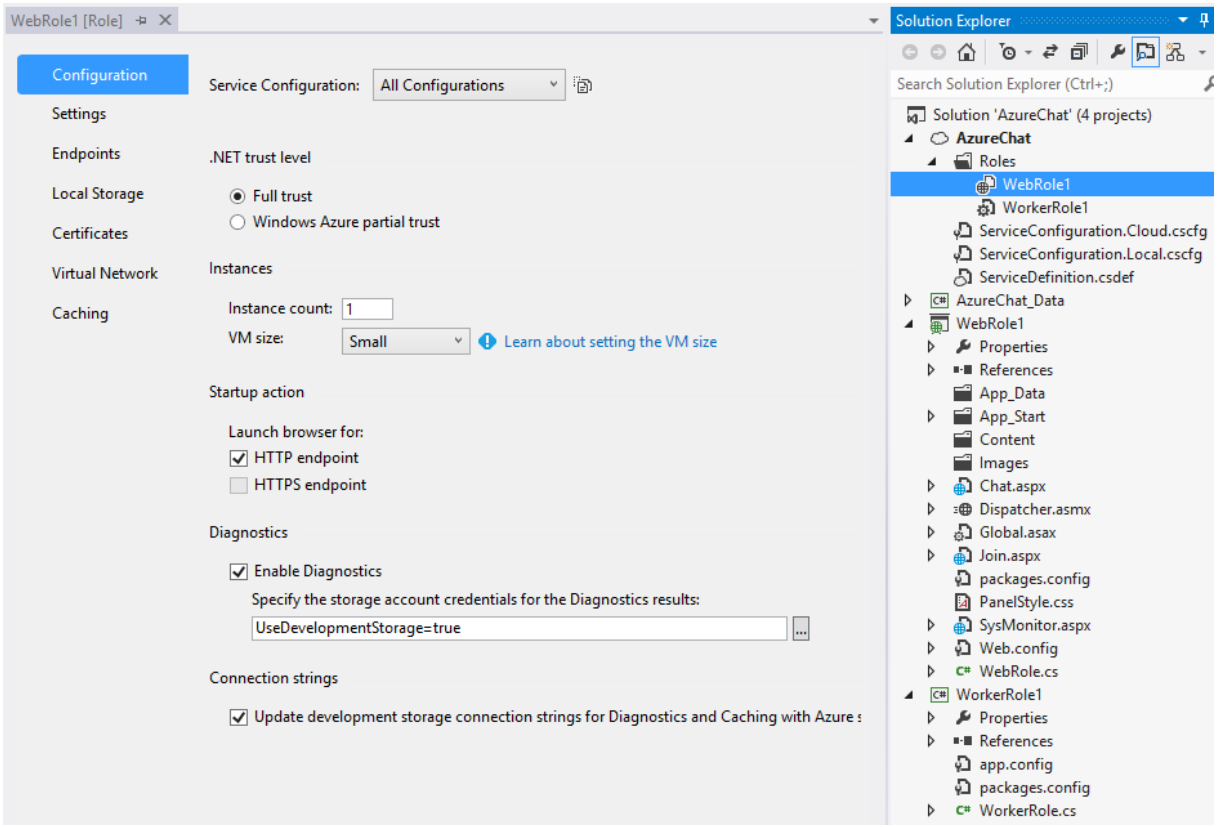
Szerencsére a fejlesztéshez elég sok segítséget nyújt a Visual Studio. (Én a legújabb, 2012-es változatot használtam.) Szinte egy teljes kis Azure-t kapunk hozzá, két program: a *Storage Emulator* és a *Compute Emulátor* formájában. Mindkettőnek van egy kis grafikus felülete. Az előbbié leginkább arra jó, hogy töröljük a teleszemetelt adattárainkat, míg az utóbbién a rendszer üzeneteit (Event Log-ot) követhetjük nyomon, futó virtuálisgép-példányonként egy-egy konzolon.

Az tehát kényelmes dolog, hogy amikor az ember leüti az F5-öt, a program szépen lefordul, rátöltődik a *Compute Emulator*-ra, majd szépen megnyílik egy böngészőablak, betöltve pl. a Web Role-unkhoz tartozó egyik weboldalt. (Viszont ezt senki ne próbálja ki például egy netbookon, ez a folyamat kb. fél percig tart az én elfogadható teljesítményű Core 2-es gépemen… Ez néha bosszantó tud lenni, amikor az ember próbál kijavítani egy hibát, és többször kell nekifutni. – No de hol van ez attól, hogyha mindig fel kéne töltenünk valamit a felhőbe, és ott tesztelni…)

Továbbá szerintem a WATK-et [1] is elég jól megcsinálták, a Hands-on lab-ok betöltik a code-snippetjeiket a Visual Studio-ba, így könnyebben követhetjük a tananyagot. Csak azt hiányoltam, hogy néhány megkötésre nem hívták fel a figyelmet.

A .Net-es API is egész jó, csak sok a furcsa megkötés, amitől néha dőlnek az Exception-ök, és nem túl értelmes üzeneteket adnak… Gondolom, miután megtanulja az ember, már kényelmes.

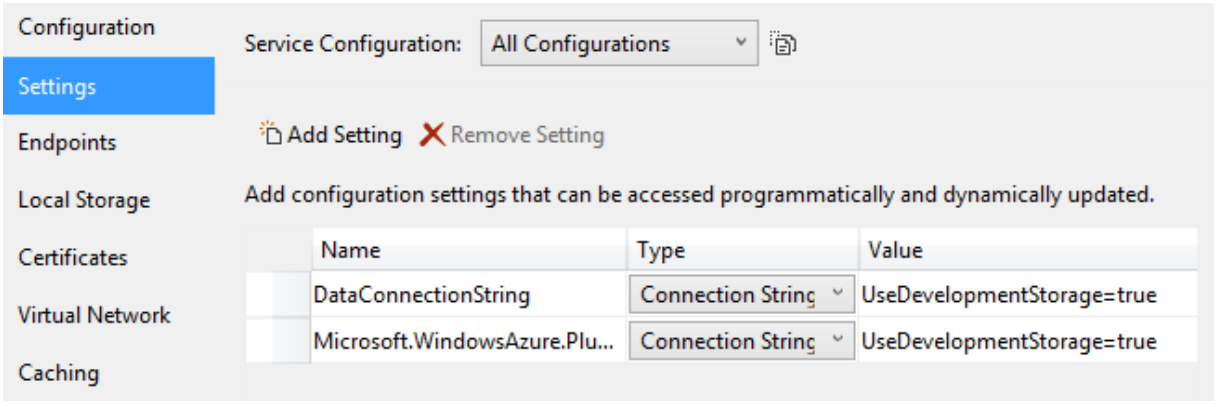
Jó pont, hogy a konfigurációs fájlokhoz (melyek XML alapúak) a Visual Studio kényelmes grafikus szerkesztőfelületet ad (18. ábra).



18. ábra: WebRole beállításai VS2012-ben

3.3.2 Táblák kezelése

Ahhoz, hogy táblákat, BLOB-okat, vagy üzenetsorokat kezeljünk, mindig szükség van egy Storage Account-ra. Az ehhez tartozó Connection String-et pedig a Visual Studio grafikus felületének segítségével állíthatjuk be. Fejlesztés közben elég bejelölni, hogy a Development Storage-ot szeretnénk használni (19. ábra).



19. ábra: Connection String beállítása VS2012-ben

A tárhelykezeléshez használható osztályokat az AzureChat_Data nevű projektbe (dll) tettem (a WATK-et követve). Így mindkét Role eléri őket.

A táblák kezeléséhez először létre kell hoznunk egy entitásokat reprezentáló osztályt. A következő példa a *Clients* táblához tartozik:

```
namespace AzureChat_Data
{
    public sealed class ClientEntity
        : Microsoft.WindowsAzure.StorageClient.TableServiceEntity
    {
        public DateTime LastRecieved { get; set; }

        public ClientEntity(string name, string room):base(room, name)
        {
            LastRecieved = new DateTime(1962, 1, 1);
        }

        public ClientEntity()
        {
            LastRecieved = new DateTime(1962, 1, 1);
        }
    }
//…
```

Látszik, hogy a kulcsokon kívül csak egy idő típusú mezőt tartalmaz (a kulcsok az ősosztályban vannak deklarálva). A Tábla szerkezete a következő:

- PartitionKey: a szoba neve
- RowKey: a felhasználó neve
- LastRecieved: az utolsó üzenet időpontja (egyértelműen azonosítja a szobán belül), melyet a felhasználó megkapott.

Ez például azzal jár, hogy ha kijelölünk egy szobát, ki tudjuk listázni az összes benne lévő felhasználót, méghozzá név szerint ABC sorrendben.

De most leginkább azokat a részleteket próbálom kiemelni, melyekre a WATK nem tér ki:

- A '#', '/', '\', '?' tiltott karakterek a kulcsmezőkben.
- A DateTime típus engedélyezett legkisebb értéke nem a DateTime.MinValue(), hanem 1601.01.01. (A kódban szereplő 1962-t korábban elnézhettem…)
- Ha referenciatípust tárolunk (string), értéket kell adnunk neki, mielőtt a táblába mentjük. Nem maradhat null.
- Kötelező egy default konstruktort megadni – ugyanis a táblából való olvasás után az API azzal hozza majd létre az osztályunkat, majd ezt követően állítja be a tulajdonságait a kiolvasott értékekre.
- Célszerű autoproperty-ket használni: kiolvasás után a setterek lefutnak, ami zavaró lehet. (Én pl. úgy jártam, hogy az üzenetek küldési idejét egy setterben próbáltam beállítani, majd csodálkoztam, hogy a kiolvasott üzenetek küldési időpontja mindig az aktuális idő…)
  Ha pedig valamihez csak gettert írunk (readonly property), az API annak az értékét is menteni fogja a táblába, de kiolvasáskor nem fogja tudni tudni beírni az újonnan létrehozott objektumba.
- A tulajdonságoknak nem lehet akármilyen típusa. Csak a következők támogatottak: byte[], bool, Guid, DateTime, string, int, long, double. [9]

A fenti megkötések azért bosszantóak, mert nem derülnek ki fordítás közben. Csak arra leszünk figyelmesek, hogy a program dobálja az Exception-öket, melyeknek nem minden esetben logikus a szövegük.

Ezzel tulajdonképpen definiáltunk egy sémát. Ebben a programban fix sémát használtam, de lehetne kötetlent is.

Következő lépésként le kell származtatnunk a TableServiceContext osztályból. Ez az osztály egy kapcsolatot reprezentál a táblához. Segítségével írhatjuk/olvashatjuk a tábla tartalmát. Azok a műveletek, melyeket a SaveChanges() metódus hívása előtt végrehajtunk, tulajdonképpen egy tranzakciót jelentenek, egyszerre hiúsulnak meg, ha valami baj történik.

```
class ClientDataContext
        : Microsoft.WindowsAzure.StorageClient.TableServiceContext
{
        public const string TableName = "Clients";

        public ClientDataContext(string baseAddress,
            Microsoft.WindowsAzure.StorageCredentials credentials)
            :base(baseAddress, credentials) {}

        //A függvény neve meg kell, hogy egyezzen TableName-mel!!!
        public DataServiceQuery<ClientEntity> Clients
        {
            get { return this.CreateQuery<ClientEntity>(TableName); }
        }

        public void AddClient(ClientEntity client)
        {
            this.AddObject(TableName, client);
        }

        public void RemoveClient(ClientEntity client)
        {
            //remove unconditionally:
            this.AttachTo(TableName, client, "*");
            this.DeleteObject(client);
        }
```

```
        public void UpdateClient(ClientEntity client)
        {
            this.AttachTo(TableName, client, "*");
            this.UpdateObject(client);
        }
    }
```

Itt arra érdemes figyelni, hogy a kommenttel is jelölt helyen lévő függvény, mellyel lekérdezzük majd lekérdezhetjük a táblát, azonos nevű kell, hogy legyen a táblával. Ha ez nem teljesül, Exception-t kapunk.

Az osztály végén található RemoveClient és UpdateClient függvények elvileg arra lennének jók, hogy gondolkodás nélkül töröljünk, vagy felülírjunk adatokat a táblában, nem törődve a Timestamp értékével. Ez annyi előnyt jelentene, hogy nem kéne előre lekérni az adott entitást (feleannyi adatmozgás). Nálam valamiért nem működött, így nem használtam őket. Helyettük az ősosztályban már elérhető RemoveObject és UpdateObject függvényeket használhatjuk. (Az AddObject-et azért volt érdemes wrappalni, mert az előző kettővel ellentétben az paraméterül várja a tábla nevét is, amit ebben a context osztályban rögzítettem.)

Mielőtt használhatnánk a táblát, még egy dolgunk van hátra: a rendszer indításakor beállítani a ConnectionString-et, és létrehozni a táblákat, ha esetleg nem léteznének.

Részletek következnek a TableManager osztályból, amely a táblák kezelését végzi:

```
private static readonly CloudStorageAccount storageAccount;
private readonly ClientDataContext clientContext = null;
static TableManager()
{
    storageAccount =
           CloudStorageAccount.FromConfigurationSetting("DataConnectionString");
    CloudTableClient.CreateTablesFromModel(typeof (ClientDataContext),
                           storageAccount.TableEndpoint.AbsoluteUri,
                           storageAccount.Credentials);
//…
}
public TableManager()
{
    clientContext = new ClientDataContext(
                             storageAccount.TableEndpoint.AbsoluteUri,
                             storageAccount.Credentials);
    clientContext.RetryPolicy = RetryPolicies.Retry(3,
                                            TimeSpan.FromSeconds(1));
    clientContext.IgnoreResourceNotFoundException = true;
//…
}
```

Ez eddig a *Clients* táblához tartozó inicializáló rész. Az IgnoreResourceNotFoundException igazra állítása azért fontos, mert így, ha egy nem létező kulcspárhoz tartozó kliens adatait kérdezzük le, nem egy ronda Exception-t kapunk, hanem helyette a lekérdezés egy üres enumerációval fog visszatérni. Így ellenőrizhetjük, hogy egy adott néven létezik-e már felhasználó. Az ezt megvalósító függvény a következő:

```
public bool ClientExists(string room, string name)
{
    var query = from c in clientContext.Clients
                where c.PartitionKey == room && c.RowKey == name
                select c;
    return query.ToList().Count > 0; //.Any() is not supported... :(
}
```

Amint az látható, a tábla adatait a korábban bemutatott context osztályon keresztül érhetjük el, legkönnyebben egy LINQ lekérdezés segítségével. Pozitívum, hogy a feltételeinket kényelmesen, SQL-szerűen fogalmazhatjuk meg, viszont ismét előjön a probléma: rengeteg megkötés van, és minden hiba csak futásidőben derül ki.

Itt például azt láthatjuk, hogy az Any függvény nem támogatott. Helyette úgy tudjuk az „ürességet" ellenőrizni, hogy meghívjuk a ToList függvényt, ami kiértékeli, majd listává alakítja a lekérdezést. Ha ennek az elemszáma 0, a lekérdezésnek nem volt eredménye.

A következő példa az üzenetek (Messages tábla) lekérdezéséről szól. A függvény visszaadja egy adott szoba üzeneteit, adott számú napra visszamenőleg:

```
public IEnumerable<MessageEntity> GetMessages(string room, int daysback)
{
    daysback = Math.Min(daysback, MAX_DAYS);
    DateTime start = DateTime.UtcNow.AddDays(-daysback).Date;
    IEnumerable<MessageEntity> result = new List<MessageEntity>(100);
    for (int i = 0; i <= daysback; i++)
    {
        var date = start.AddDays(i);
        var datequery = from m in messageContext.Messages
                        where m.PartitionKey == room + "," +
                                                date.ToString("yyyyMMdd")
                        select m;
        result = result.Concat(datequery.AsTableServiceQuery().ToList());
    }
    return result;
}
```

Az üzenetek tárolásánál a következő kulcsokat alkalmaztam:

- PartitionKey: küldés dátuma (pontosabban amikor a rendszer megkapta)
- RowKey: küldés időpontja a napon belül, 0,1 ms pontossággal. (Itt persze, előfordulhat ütközés, de ez nem gond, legfeljebb a kliens újra próbálja az adást. Ezt a JavaScript automatikusan megteszi…)

Láthatjuk, hogy a különböző napokon küldött üzenetek különböző partíciókban vannak. Ezeket csak egyenként lehet lekérdezni, ezért van szükség a *result* nevű listára és a for ciklusra, ami egyenként végigiterál a napokon, lekérdezi őket, majd az eredményüket hozzáfűzi a *result*-hoz.

Még egy kis kódrészletet bemutatok, abból a függvényből, ami egy adott szobában, egy adott időpont utáni első üzenetet adja vissza:

```
var q1 = from m in messageContext.Messages
         where m.PartitionKey == room + "," + last.ToString("yyyyMMdd")
               && m.RowKey.CompareTo(last.ToString("hhmmssffff")) > 0
               && m.RowKey != last.ToString("hhmmssffff")
         select m;
var r1 = q1.Take(2).ToList();
```

A DateTime *last* nevű paraméter mondja meg azt a bizonyos időpontot. Ebben a részletben egyrészt az az érdekes, hogy bár használhatjuk a CompareTo-t arra, hogy feltételeket szabjunk meg a RowKey-re (és csak arra!), olyan furcsán működik, hogy nem képes a határozottan nagyobb relációt megvalósítani. Tehát itt a CompareTo-nak kicsit más szemantikája van: egy minimum- és egy maximumértéket határozhatunk meg vele. További érdekesség, hogy még a második (!=) feltétellel együtt is mindig megkapjuk azt az üzenetet, amit épp a last időpontban küldtek. Ezért a lekérdezés után még ellenőrizni kell a kapott lista tartalmát…

A BLOB és Queue kezelés nem tartogatott ennyi meglepetést, a leírásnak megfelelően működtek.

3.3.3 Fájlfeltöltés

Nem igazán az Azure-hoz tartozik, inkább az ASP.Net-hez, de leírok egy problémát, amire a megoldást több fórumról sikerült nagy nehezen összevadászni. Nevezetesen: ha egy aspx-en belül található egy FileUpload vezértlő és egy UpdatePanel is, a FileUpload nem fog működni.

Többféle magyarázatot találtam, de a megoldás, ami végül bevált, a következő:

- A form tag-hez, melyen belül vannak, hozzá kell adni az `enctype="multipart/form-data"` attribútumot.
- A FileUpload-ot bele kell tenni az UpdatePanel-be. (Látszólag semmi értelme.)
- Azt a gombot, aminek az onclick eseményére történik a feltöltés (és szintén az UpdatePanelen van), hozzá kell adni az UpdatePanel triggerei közé, a következő módon:

```
<Triggers>
    <asp:PostBackTrigger ControlID="UploadButton"/>
</Triggers>
```

Ez nálam megoldotta a problémát.

3.4 Deployment

A program telepítéséhez az alábbi műveleteket kell elvégezni az Azure menedzsment portálon:

- Létre kell hozni egy új Affinity Group-ot. Az ezen belüli elemek egymáshoz fizikailag „közel” kapnak helyet.
- Létre kell hozni egy – már sokat emlegetett – Storage Account-ot is, melyet az előbbi csoporthoz hozzáadunk. Nemsokára ennek a „Primary Access Key”-ére lesz szükségünk, ezt másoljuk ki.
- Szintén létre kell hozni egy Cloud Service-t, melyet szintén hozzá kell adni a csoporthoz.

Ezután a Visual Studio grafikus felületén:

- Be kell jegyeznünk a Storage Account nevét, és Primary Access Key-ét a ServiceConfiguration.Cloud.cscfg fájlba. Ezt direktben be is írhatjuk az XML alapú fájlba, de kényelmesebb kattintgatni. Mindkét Role-hoz külön be kell írni.
- Jöhet a fordítás! Jobb-klikk a „Cloud Project” nevére (a Solution-ben az első elem), majd Package. Az eredmény két fájl: egy .cspkg, és egy .cscfg. Előbbi tartalmazza magát az alkalmazást, utóbbi pedig a külön is frissíthető konfigurációs beállításokat.

Visszatérve a menedzsment portálra:

- Kiválasztjuk az új Cloud Service-ünket, és rákattintunk az „Upload a new production deployment” linkre. Itt feltölthetjük az imént generált két fájlt.
- A sikeres telepítést követően a „Site URL”-en keresztül érhetjük el az alkalmazást.

4 Konklúzió

Az Azure-ra való fejlesztést nyilván meg kell tanulni, de szerintem kifejezetten érdekes, főleg mivel elég komoly infrastruktúrákat tudunk kipróbálni. A 90 napos ingyen próba pedig jó húzás: talán ezzel rá lehet „kapatni” az embereket, de ha nem, legalább lehet próbálgatni… Szerintem érdemes minden érdeklődőnek kipróbálnia!

Hivatkozások

[1] Windows Azure Training Kit (WATK): http://windowsazure-trainingkit.github.com/

[2] Introducing Windows Azure: https://www.windowsazure.com/en-us/manage/services/fundamentals/intro-to-windows-azure/

[3] Windows Azure Execution Models: https://www.windowsazure.com/en-us/develop/net/fundamentals/compute/

[4] Data Management and Business Analytics: https://www.windowsazure.com/en-us/develop/net/fundamentals/cloud-storage/

[5] Service Bus: https://www.windowsazure.com/en-us/develop/net/fundamentals/hybrid-solutions/

[6] Pricing Calculator: https://www.windowsazure.com/en-us/pricing/calculator/

[7] Robbin Cremers: Everything you need to know about Windows Azure Table Storage to use a scalable non-relational structured data store http://robbincremers.me/2012/03/01/everything-you-need-to-know-about-windows-azure-table-storage-to-use-a-scalable-non-relational-structured-data-store/

[8] Reverse AJAX technique in ASP.NET (CSASPNETReverseAJAX): http://code.msdn.microsoft.com/CSASPNETReverseAJAX-7a1f0c2b#content

[9] Understanding the Table Service Data Model: http://msdn.microsoft.com/en-us/library/windowsazure/dd179338.aspx