# Bevezetés

# Felhasznált Technológiák

A szakdolgozat elkészítéséhez több keretrendszert is felhasználtam. Kliens oldalon első sorban a Unity3D játékmotort az alkalmazás alapjaként, Vuforia motort tárgyak felismeréséhez, valamint SQLite adatbázist a szerverről lekérhető adatok tárolására gyorsabb adatelérés, illetve offline üzem esetére. A szerver egy ASP .NET keretrendszert felhasználva készült REST API, amely MySQL adatbázist használ. A kliens és a szerver is a Dapper micro ORM-et alkalmazza, hogy az adatbázisból lekért adatokat könnyen kezelhető objektumokká alakítsa. Az API emellett a Dapper egy bővítményét, a Dapper Extensions-t használja fel, mely képes az alapvető SQL műveletekhez kódot generálni típus alapján. JSON objektumok kezeléséhez a kliens, illetve az ASP .NET keretrendszer beépítve használja a JSON .NET keretrendszert. Az alkalmazás két része C# nyelvben íródott. Az alábbiakban ezeket a technológiákat részletesen ismertetem.

## C#

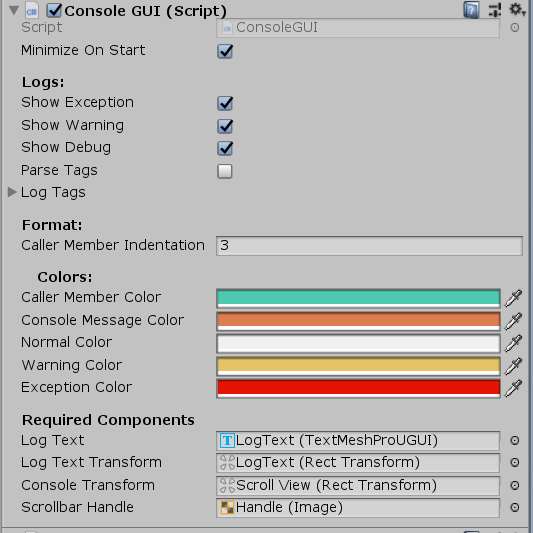
A C# egy erősen, statikusan típusos, objektum orientált nyelv mely leginkább a Java-hoz hasonlítható. A két nyelv eleinte kísértetiesen hasonlított egymásra, ám mostanra már külön vált a fejlődésük. A java a valódi cross-platform programozás nyelve, míg a C# a Windows fejlesztésé lett. A Microsoft az utóbbi években hozzáállást váltott és aktívan törekedik arra, hogy a C# tényleg platform független legyen. Jelenleg az ASP .Net Core Linuxon, Mac-en, valamint Windows-on fut, ezzel közelebb jutva ehhez a célhoz. A programnyelv emellett használható Unity-ben játékfejlesztésre, Xamarin segítségével pedig mobil fejlesztésre. A C# előnye vetélytársával szemben főként az általános osztályokban és a funkcionális programozás lehetővé tételéből származik. Ennek egy kitűnő példája a Linq névtérben található függvények, melyek bármilyen iterálható típusra meghívhatók, és az Action és Predicate elemeknek köszönhetően – ezek olyan változók, amik függvényeket tárolnak, előbbinek nincs visszatérési érteke, utóbbinak egy boolean érték - a szituációra szabhatók.

## Unity3D

A Unity3D az egyik, ha nem legismertebb játék motor napjainkban. Népszerűségét többek között köszönheti árazásának, a szerkesztő művészbarát kialakításának, a támogatott platformoknak, valamint, hogy játékfejlesztés legfőbb technikai kihívásait (fizika, képalkotás stb.) átvállalja a keretrendszer felhasználójától, mégis nagyon flexibilisen használható. A virtuális, kiterjesztett, illetve kevert valóság alkalmazások fejlesztésében is a Unity a legnépszerűbb választás, mivel mobil eszközökre is jól optimalizált és ezeket az alkalmazásokat főként mobilokra fejlesztik.

### GameObject-ek és komponensek

A keretrendszer flexibilisségének és a kódok újrahasználhatóságának oka a kódok rendszerezésére választott megközelítés. Minden entitást a játéktérben egy GameObject reprezentál. Egy GameObject tulajdonképpen csak egy szervezési egység, egy tároló, amely a játéktérben futtatható kódokat a MonoBehaviour-öket vagy másként komponenseket magában foglalja. A GameObject-en keresztül a komponensek egymást is képesek elérni. Alapvető komponens például a Transform, mely az objektum helyzetét, forgását, skálázott méretét határozza meg illetve kezeli a hierarchiában alatta elhelyezkedő Transform-okat és minden objektumon kötelezően szerepel. A GameObject-ekhez tetszőleges számú komponenst adhatunk így könnyedén személyre szabva annak viselkedését. A komponensek publikus vagy „Serializefield” attribútummal ellátott adattagjainak a szerkesztőből adhatunk értéket, így könnyedén az adott helyzetre lehet szabni viselkedésüket. Fontos megjegyezni, hogy nem képes mindent szerializáni, ezeket a mezőket nem jeleníti meg a szerkesztő, ilyenek például a property-k, list-ek, dictionary-k stb.



A képen a ConsoleGUI komponens beállításai láthatóak.

### MonoBehaviour szerkezete

Ahhoz, hogy a kódunkat egy GameObject-hez tudjuk csatlakoztatni annak a MonoBehaviour osztályból kell öröklődnie. Egy MonoBehaviour életciklusa során különböző események következnek be, ezekre reagálva tudjuk a kódunkat futtatni. Ilyen például az Awake, mely közvetlenül a komponens létrejötte után, a Start mely az ezt követő képkocka előtt vagy az Update, amely minden képkocka kirajzolásakor hívódik meg. A Unity az adott eseményekhez tartozó függvények aláírását keresi az osztályunkban, hogy meg tudja hívni azt. Ha egy ilyen nevezetes függvényt használunk, azt a Visual Studio kék kiemeléssel jelzi.



### Coroutine

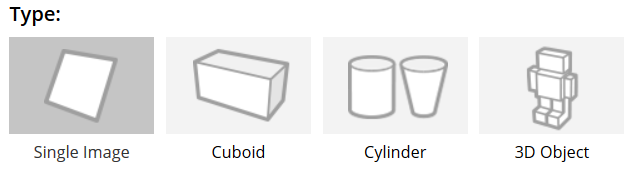
A MonoBehaviour-öket a Unity egy szálon kezeli, szekvenciálisan meghívva a hozzájuk tartozó különleges függvényeket. Ebből következik, hogy ha olyan kódot írunk, amelynek futtatása túl sokáig tart a programunk érezhetően meg fog akadni, így a program várakoztatására sincs lehetőség a hagyományos módon. Ezt a problémát a Coroutine-ok oldják meg, melyek képesek a kód futását félbehagyni, hogy aztán később onnan folytassák azt. A programban többször is elő fognak kerülni, elsősorban API hívások válaszára várva, ugyanis ilyenkor akár több másodperces megakadások jelentkeznének ezek használata nélkül. Szintaxist tekintve, ezt iterátor függvények implementálásával tehetjük meg, melyből a „yield return”, valamint a „yield break” kulcsszavakkal térhetünk vissza. Az így kapott függvényt paraméterült adhatjuk a MonoBehaviour StartCoroutine függvényének.

### AssetBundle

## Vuforia SDK

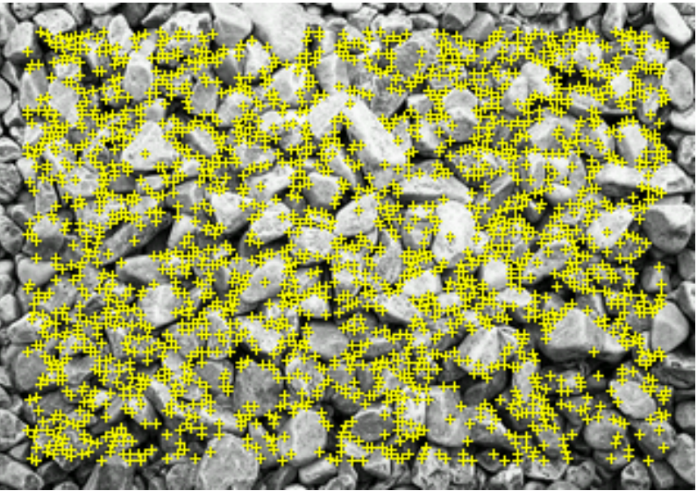
A Vuforia SDK célja, hogy mobilok kameráját és számítógépes látást használva felismerjen 2 és 3 dimenziós objektumokat, kövesse azok pozícióját, dőlését, méretét a kamerához képest. Az objektumokat jellegzetes pontjaik alapján képes felismeri. Ezeket a képességeket felhasználva a programok úgy tudnak elhelyezni 3 dimenziós tartalmakat a követett objektumokhoz képest a kamera képén, hogy annak perspektívájából az hiteles legyen. A Vuforia felhasználható C++, Java és a Unity bővítményén keresztül .Net nyelvekben. A felsorolt technológiák közül a Unity a legnépszerűbb választás, mivel támogatja az Android-ot, iOS-t és az UWP platformokat is, így közös kódbázissal fejleszthetünk több platformra, emellett játékmotor lévén komoly grafikai lehetőségeket nyújt.

### Felismerhető objektum típusok



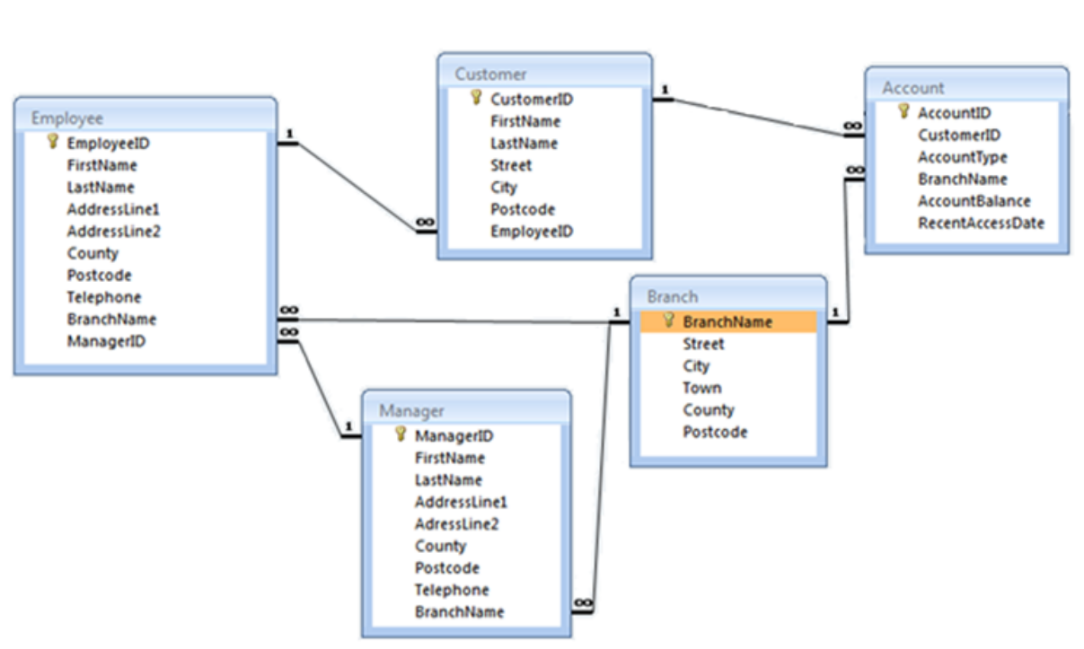
4 féle felismerhető és követhető objektum fajta van, mindegyik picit más alkalmazásra. Az egyszerű képet posztereket, kártyák, újságok és egyéb lapos tárgyakhoz ajánlatos használni. Hasábot olyan testekhez, melynek nem csak egy oldalán, ha minden a hat oldalán jellegzetes tartalom van. Hengert konzervek, poharak egyéb hengeres tárgyak esetében használhatjuk. Ha egy test elég komplex, akkor 3 dimenziós objektumként a legcélszerűbb megközelíteni, a fejlesztők a kicsit, jól körbejárható, részlet gazdag objektumokat ajánlják erre a célra. A Vuforia működését nagyban lassítják az ismétlődő minták, ezek nem ajánlottak, illetve nem képes forgás szimmetrikus képek tanítására.

### Tanítás



Felismerhető objektumok adatait a Vuforia DataSet-nek nevezett, .dat kiterjesztésű fájlokba menti. Ezeknek a tanítása a cég oldalán keresztül történik. Regisztrációt és bejelentkezést követően lehetőségünk van a „target manager” menüpont alatt új adatbázisokat létrehozni, majd ezekhez a fentebb említett objektum típusok közül újakat hozzáadni. A betanított adatbázist ezután letölthetjük és importálhatjuk Unity-be vagy a többi platformhoz tartozó IDE-be.

## Relációs adatbázisok



A relációs adatbázisokban az adatok táblákba vannak rendezve, minden tábla egy entitást ír le például egy felhasználó, termék stb. A táblák sorokból és oszlopokból állnak. Az oszlopok az entitás tulajdonságai, például a felhasználó neve, míg a sorok az entitás egy-egy példányát. A különböző entitások között többféle kapcsolat is lehet, például egy személyhez tartozhat több megvásárolt termék – egy-több kapcsolat -, a szülő anyja – egy-egy kapcsolat –, személyek ismeretsége – több-több kapcsolat. Az oszlopoknak tipikusan van egy típusuk, például egész számok, lebegőpontos számok, szöveg stb. Gyakorlatilag mindegyik relációs adatbázis SQL-t használ az interakció nyelveként. Az SQL az első relációs adatbázisokkal együtt jelent meg az IBM fejlesztésében és azóta ipari standarté vált.

### MySQL

A MySQL egy nyílt-forráskódú, ingyenesen használható szerverként futtatható, tranzakciókat kezelő relációs adatbázis, tárolt eljárásokkal, trigerek-kel, view-kal stb. Olyan felhasználásokra szánták, ahol egyszerre több felhasználót is ki kell szolgálni, például weboldalak és egyéb alkalmazások szerverén futva.

### SQLite

SQLite egy önálló, szerver nélküli, konfigurációt nem igénylő tranzakciókat kezelő relációs adatbázis. Az SQLite teljesen ingyenesen használható, az adatbázis egésze egyetlen fájlban kap helyet, mely tartalmazza a táblákat, indexeket, view-kat, trigger-eket, szabadon másolható 32 és 64 bites rendszerek közt, valamint kis és nagy-endián architektúrák közt. Ezen tulajdonságai révén világelső, ha lokális adattárolásról van szó. Az Android platformon ipari standarté nőte magát az elmúlt években. A fejlesztők szerint az SQLite-ra nem, mint egy másik adatbázis kell gondolni, hanem mint a manuális fájlba írás és fájlból olvasás helyettesítése. Azért választottam az SQLite-ot, mert könnyedén tükrözni tudtam vele a MySQL adatbázis szerkezetét, ami lehetővé tette a probléma mentes gyorsítótárazást.

## ASP .NET

Az ASP .NET a .Net platform bővítésére szolgáló kód könyvtárak és fejlesztői eszközök összessége, melyet kifejezetten a web fejlesztés céljával hoztak létre. Fontosnak tartom megjegyezni, hogy az ASP. Net alatt én (és mások általában) a Windows specifikus verziót értem. Az utóbbi években a Microsoft kifejlesztette az ASP .Net Core-t, ami a keretrendszer platform független, letisztultabb, újratervezett változata. A fejlesztők azt ajánlják, hogy minden új projektet ebben kezdjünk, hiszen már felzárkózott elődjéhez, illetve minden későbbi innovációt a Core verzióban terveznek megvalósítani. A szakdolgozat írásának kezdetekor erről még nem tudtam, ezért a régebbi verzióval dolgoztam. Az új lehetőségek közé tartozik a http kérések kezelése, dinamikus weboldalak fejlesztésére alkalmas Razor szintaxis, authentikáció és még sok más. A számomra a legvonzóbb lehetőség a Web Api projekt, ugyanis ezzel könnyedén, C#-ot használva lehet REST Api írni.

### HTTP

A HyperText Transfer Protocol vagy HTTP az internetes kommunikáció alapja. A protokoll kéréseken és arra adott válaszokon alapul kliens-szerver környezetben, mint amilyen egy böngésző és egy weboldal szerverének az esete. A szerverről érkezet válasz tartalmazza művelet sikerességét és adott esetben a kért adatokat. Különböző metódusok léteznek, ezekből a négy legtöbbet használt a Get, Post, Put, Delete. A Get metódus egy erőforrás reprezentációjának lekérésére szolgál és semmilyen körülmények közt nem szabadna módosítania azt vagy bármilyen más hatásának lennie. A Post ige arra való, hogy a megadott elérési úton lévő gyűjteményhez új elemet adjunk hozzá. A Put arra szolgál, hogy egy már jelenlévő erőforrást módosítsuk, frissítsünk. A Delete a törlésére.

### REST

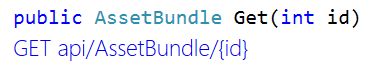
A REST szoftverarchitektúra nagy, hálózat alapú rendszer igényeit észben tartva jött létre, az internetét. Az architektúra az erőforrások fogalma köré épül, kliensekből és szerverekből áll. A kliens olyan kéréseket küld a szervernek, mely egy erőforrásra utal és azt vagy annak állapotának változását kéri (frissíti, törli, feltölti), a szerver feladata ezeket a kéréseket feldolgozni és a megfelelő választ adni. Erőforrás tulajdonképpen bármi lehet, amit értelmes formában címezni lehet például a felhasználók listája, egy adott felhasználó kapcsolatai stb.

Ahhoz, hogy ha RESTfulnak lehessen nevezni a következőknek kell megfelelnie:

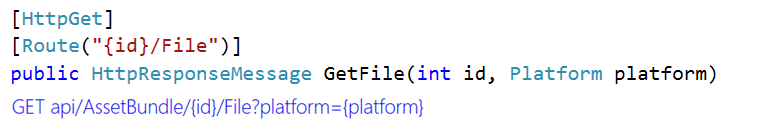
* kliens-szerver alapú
* állapotnélküli, tehát a szerver semmilyen plusz adatot nem tárol a kliens állapotáról, ez kizárólag a kliens feladata
* gyorsítótárazható, a kéréseknek megvannak jelölve, hogy a kapott válasz lementhető-e későbbre vagy sem
* uniform interfész, ahhoz, hogy ez létrejöhessen további négy megkötésnek kell megfelelni: a hivatkozott erőforrás a kérésből egyértelműen kiderül, erőforrás manipulálhatósága reprezentáció segítségével, minden üzenet önmagában elég információt tartalmaz az értelmezéshez, hypermédián keresztül az api összes funkciója bejárható
* rétegezett rendszer, a hierarchiának megfelelően az egyes komponensek nem tudnak, csak a közvetlen arról a rétegről, amellyel interaktálnak
* (opcionális) letölthető kódrészletek, hogy csökkentsék a kliens számára előre implementálandó funkciók számát

### Controlerek és URL kezelés

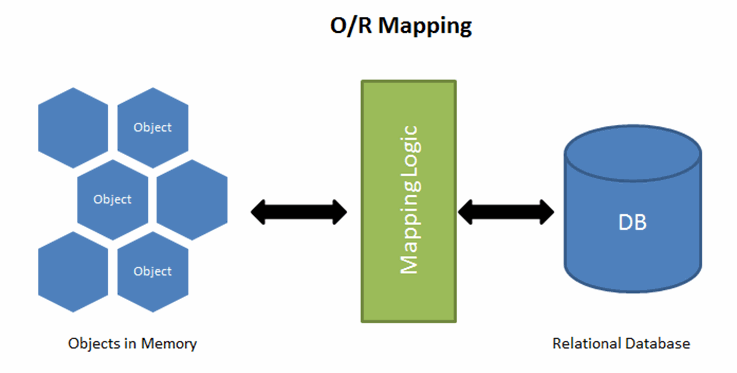
A web Api projekt legfontosabb elemei a Controller-ek, ezek olyan osztályok – tipikusan a „Controller” mappában elhelyezkedve – melyek az ApiController osztályból öröklődnek, és a nevük „Controller”-re végződik. A keretrendszer az osztályok nevét és a bennük lévő publikus függvények aláírását felhasználva automatikusan generálja az elérhető URL-eket, ehhez a függvény nevének meg kell egyeznie a megfelelő http igével.



Bizonyos esetekben azonban előfordulhat, hogy ezt a generált URL-t szeretnénk felülírni, erre is van lehetőségünk. Ezt úgy tehetjük meg, ha a függvény fölé helyezünk egy „Route” attribútumot, mely leírja a hozzá tartozó URL-t, ilyenkor azt is meg kell határoznunk, hogy a kérésnek milyen http igét kell használnia. Megadható egy „RoutePrefix” attribútum is az osztály felett, mellyel megadhatjuk az elérési út elejét, így az ismétlődő részeket nem kell többször leírnunk.



## Dapper



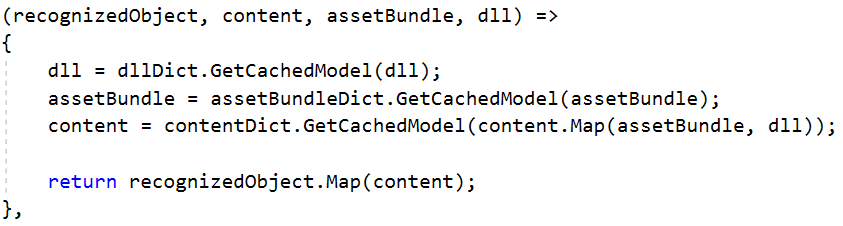
A Dapper ismertetéséhez szükséges bevezetnem az ORM fogalmát. Az ORM (Object Relation Mapping) magyarul objektum-relációs leképzés, két egymással nem kompatibilis objektum konvertálására használatos, ez első sorban az adatbázisból lekért táblák rekordjainak típusos objektummá való átalakítását jelenti. Léteznek vastag ORM-ek, mint például az Entity Framework, ezek átveszik a fejlesztőtől az SQL írás feladatát a generált kóddal, automatán kezelik az objektumok közti kapcsolatokat. Bizonyos esetekben az ilyen megoldások rossz minőségű SQL kódot generálnak, túl komplexek egy kis rendszerhez vagy sok tanulást igényelnének az effektív alkalmazásukhoz. A Dapper ezzel szemben egy vékony ORM, nem generál helyettünk SQL kódot, a lekért táblák objektummá alakítását végzi el, valamint a megírt query-k biztonságos paraméterezését.

### Működés

A Dapper több függvénnyel egészíti az IDatabaseConnection interfészt. Mivel a függvények interfészt használnak, ezért minden adatbázishoz tud kapcsolódni, ha az megfelelően implementálja az interfészt. A függvények közül a legfontosabb a Query. Több felülírása is van, amikkel legfeljebb 7 táblát tudunk leképezni és egy objektumként visszaadni. Ha több táblának az összekapcsolt eredményét szeretnénk objektumokká alakítani, akkor megadhatunk egy függvényt, ami az értékek összekapcsolását végzi. A függvény paraméterlistájának meg kell egyeznie a megadott típusokkal, melyekből az utolsó a rekord visszatérési értéke.

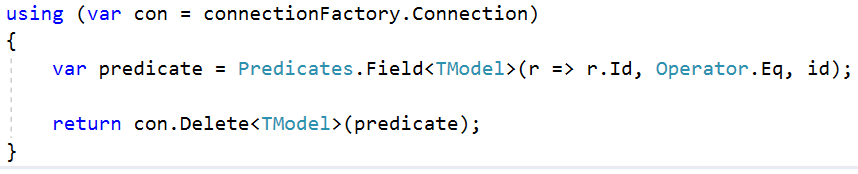


példa a típusok megadására



### Dapper Extensions

A Dapper egy bővítményét is felhasználtam, melynek az a célja, hogy az alapvető SQL műveleteket reflexiót és bővítő függvényeket – azaz futás időben való típus elemzést – alkalmazva SQL kódot generáljon beszúráshoz, frissítéshez, lekéréshez és törléshez. Képes egyszerű szűrési feltételekkel is ellátni a kódot, pl. a keresett azonosítóval, ehhez a Predicate osztályt kell használnunk, ahol lambda függvénnyel megadhatjuk a keresett tagot.



példa törlésnél használt predicate-re

## JSON .Net

A JSON .Net a legtöbbet letöltött NuGet csomag és a legnépszerűbb JSON konverter a .Net platformon. Az ASP .Net Api projekt alapértelmezett választása, mely rengeteg kényelmi funkciót tartalmaz, mint az XML-lé konvertálás, olvasás segítő behúzások alkalmazása stb. Nekem azért volt rá szükségem, mert a Unity által használt JSON segéd osztályból sajnos alapvető funkciók hiányoknak, mint például a property-k kezelése.

### JSON

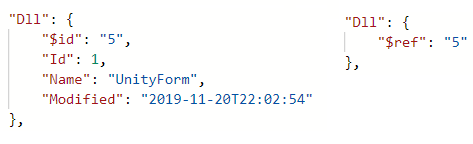


példa az API-ban használt JSON-re

A JavaScript Object Notation egy egyszerű módja az adatátadásnak. Emberileg könnyen olvasható, de a gépek is könnyen kitudják értékelni vagy generálni. Két nyelvi eleme van. Az egyik változó név és érték párok sorozata kapcsos zárójelek között, ezt a legtöbb nyelvben objektumnak, struktúrának, könyvtárnak stb. feleltethető meg. A másik egy rendezett lista, aminek szintén van valamilyen megfelelője a legtöbb nyelvben, legyen az tömb, lista, vektor stb.

### Referencia kezelés

Bizonyos JSON konverterek képesek megőrizni az objektum referenciákat. Ennek több előnye is van, egyrészt megoldhatóvá válik a körkörös referencia esete – amikor objektum A egyik eleme B, objektum B egyik eleme pedig A –, ha sok az ismétlődő adat, akkor csökkenti az üzenet méretét, valamint mivel megmaradnak a referenciák, ezért visszaalakításkor sem ismétlődnek az objektumok. Amikor referenciákat használunk minden objektum kap egy azonosítót, az „$id” értéket. Ha egy objektumot többször is át kellene alakítani, akkor másodjára már csak a „$ref” érték kerül a helyére, mely az első előforduláskor kifejtett objektumra utal.



# Rendszerterv

## Specifikáció

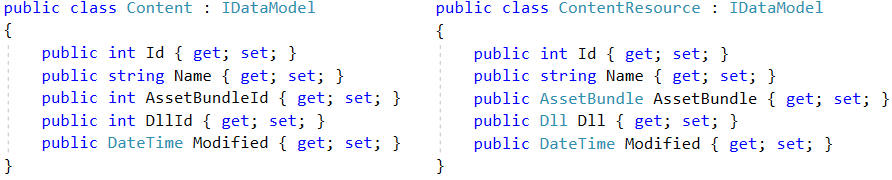
Cél olyan Android alkalmazás fejlesztése, mely a Unity és a Vuforia felhasználásával lehetővé teszi felismerhető tárgyak alapján interaktív tartalmak megjelenítését. A megjelenített tartalmakhoz és felismerhető tárgyakhoz tartozó adatokat mind szerveren keresztül érje el és töltse be, hogy ezek ne az alkalmazástól függjenek és ne azt kelljen frissíteni, csupán a letöltött adatokat. Az alkalmazás tárolja le a szerverről már lekért adatokat gyorsítótárazás valamint offline üzem esetére.

# Implementáció

## Szerver

Az API 3 projektből épül fel, a DataAcces projektből, ebben van elkülönítve az adatelérés logikája, a DataModels projektből, és magából az API projektből, mely a HTTP kérések kezelésért.

### DataModels projekt



A DataModels különleges abból a szempontból, hogy olyan kódot tartalmaz, melyet az API a DataAcces, valamint a kliens is felhasznál. A projekt lényegében két részre osztható, a modellekre, melyek olyan egyszerű osztályok, amik az adatbázis egy adott táblájának adattagjait írják le. A projekt másik fele a resource-ök, amik olyan szintén egyszerű osztályok, amikben a táblák külső kulcsai a táblához tartozó modellre vagy resource-re lett cserélve.

A projekthez tartozik 2 interfész, az IDataModel-t és az IFileModel-t. Az előbbi azt írja elő, hogy az osztály rendelkezzen egy int típusú Id nevű property-vel. A második azt, hogy legyen egy string típusú Name nevű property-je.

### DataAcces projekt

#### MySqlConnectionFactory

A factory osztályoknak általában véve az a szerepeük, hogy a példányok készítését elbasztraktálják valamint a logikáját egy helyre szervezzék. Az osztály implementálja az IConnectionFactory interfészt, melynek egyetlen megkötése egy Connection nevű IDbConnection típusú, legalább getter-rel rendelkező property. Mivel a repository-k is az IDbConnection-re épülnek, ezért a kód adatbázis független marad, a cseréje a factory osztály átírásával vagy helyettesítésével könnyen megtehető.

#### SimpleRepository

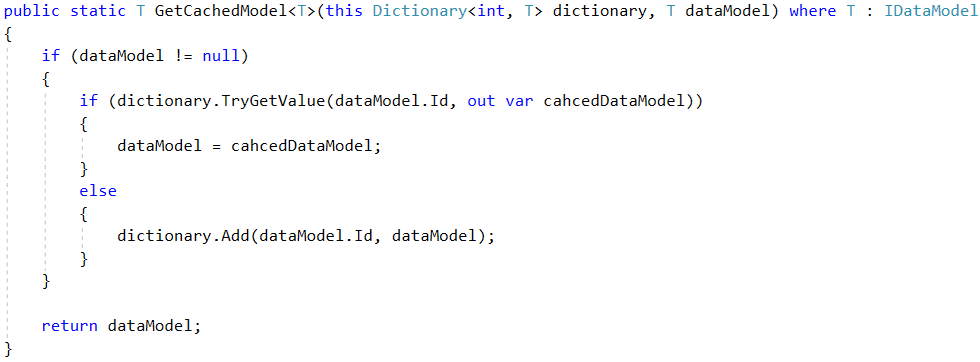
A SimpleRepository egy sablon osztály, amely, a kapott IDataModel-t implementáló osztály és a DapperExtensions segítségével képes a beilleszteni, lekérni egyszerű szűrési opcióval, törölni és frissíteni az adatokat egy táblában. A konstruktora egy IConnectionFactory-t vár, így nem függ erősen az adatbázis konkrét típusától. Az osztállyal eltudtam kerülni a kód ismétlést azokban az esetekben, amikor a lekérés csak egyetlen táblát érintene, ez az esetek többsége, ám azokat az eseteket nem tudja kezelni, mikor a Model több tábla összekapcsolásából jön létre.

#### CompositeRepository

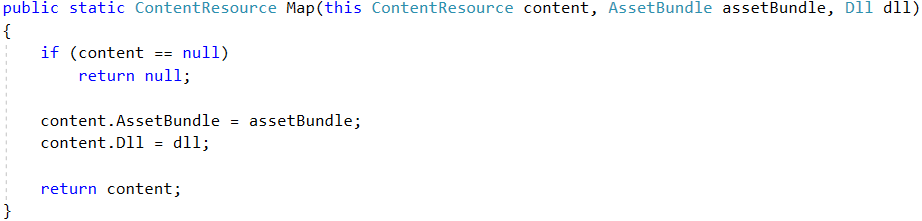
A CompositeRepository egy abstract osztály, ami SimpleRepository hiányosságát, az összekapcsolt modellek kezelését hivatott pótolni. Mivel a kliens nem igényelte, illetve az API-hoz nem készült másik kliens, ami megkönnyítené az adatok felvitelét a szerverre, ezért a CompositeRepository nem fejti ki az Add és Update metódusokat.

A CompositeRepository-ból egy osztály örököl, a RecognizedObjectRepository. Az osztály feladata, hogy kifejtse a Get és a GetAll függvényeket. Ehhez egy RecognizedObjectMapper nevű osztályt használok.

A MappFunctions file-ban helyezkedik el 3 bővítő függvény, amik nagyban segítik a kód ismétlés elkerülését és fontos bemutatnom őket, hogy a mapper osztály kódja érthető legyen.

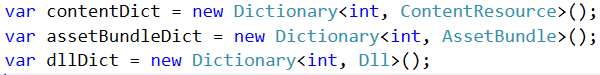


A GetCachedModel olyan dictionary-kre hívható meg, melyek int kulccsal és IDataModel értékekkel rendelkeznek. A függvény megvizsgálja, hogy a kapott modell jelen van-e a dictionary-ben és ha nincs hozzáadja. Ezután visszatér az értékkel a dictionary-ben most már biztosan jelen levő értékkel.



A másik két fontos függvény mindegyikét Map-nek neveztem, mindegyik resource-höz egy tartozik. A szerepük az, hogy a resource-höz kapcsolt modelleket a megfelelő módon beállítsák és egy olyan Resource-el térjenek vissza, ami kész a dictionary-hez adásra.

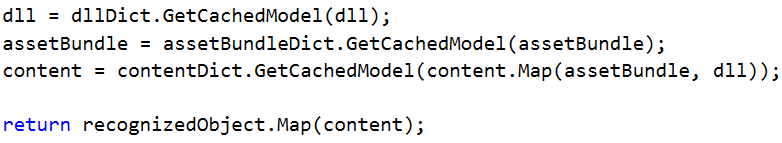
A mapper osztály a repository-hoz hasonlóan szintén egy IConnectionFactory-t vár a konstruktorában. A mapper egy függvénnyel rendelkezik, ami string formájában várja az SQL kódot, opcionálisan objectként a paramétereket és a tranzakciót.



A kapcsolat létesítése után létrehozok egy-egy dictionary-t minden modell részére. Erre azért van szükség, hogy ha egy modellt már egyszer beolvastam, akkor azt többször ne tároljam el, illetve küldjem el a kliensnek, csupán az arra való referenciát.



A Dapperhez tartozó Query sablon fügvény típus paramétereinek megadom az összes táblát, amit a lekérdezés tartalmazni fog, valamint az utolsó értéknek a kívánt visszatérési értéket. Fontos a megadott típusok sorrendje, ugyanis a Dapper felétételezi, hogy a visszakapott táblák így helyezkednek el egymás mellett, ezért a táblákat is mindig ugyan olyan sorrendben kell összekapcsolni egy adott mapper-hez. A paraméterek közt megadható egy opcionális függvény, ami felülírja a Dapper alapértelmezett függvényét, amit a rekordok objektummá alakítására használ.



A korábban létrehozott dictionary-ket és a fent bemutatott függvényeket használva, adat ismétlés nélkül kapom vissza a RecognizedObjectResource-ök listáját.

#### FileAccesRepository

A FileAccesRepository egy olyan bővített SimpleRepository, amely feltételezi, hogy hozzátartozó táblának minden sora egy másik fájlra utal. Csak olyan modelleket fogad el típusnak, amik az IFileModel interfészt is implementálják. A bővített funkciókkal a repository képes a táblához tartozó fájlokat beilleszteni, lekérni, módosítani, törlni. Ezekből a funkciókból azonban csak a lekérést használja az API. A konstruktorban megadható két opcionális paraméter, egy almappa – ez azoknál a fájloknál hasznos, amik csak bizonyos platformon használhatók, ilyen az AssetBundle – valamint a fájl kiterjesztése – ez azért opcionális, mert az AssetBundle például nem rendelkezik kiterjesztéssel.

A GetFile függvény feladata a fájlok lekérésére, ez Id alapján történik. A visszatérési értéke egy olyan tuple ami a byte tömböt és a fájl nevét tartalmazza. A metódus először lekéri az Id alapján a megfelelő modellt a SimpleRepository-ból örökölt Get-el. Ha a nem talált olyan sort az adatbázisban, aminek az Id-je megegyezik, akkor egy olyan tuple-el térek vissza, aminek mindkét értéke null. Különben egy IFileModel-t bővítő metódussal, a FilePath-al meghatározom a megfelelő URL-t. A függvény paraméterként megkapja a konstruktorban beállított *subFolder* és *extension* értékeket.



A ConfigurationManager-ről lekérem a kapott táblához tartozó mappa elérési útját – ehhez persze jelen kell, hogy legyen az appconfig fájlban a megfelelő név-érték pár. Ha a kapott almappa nem null, akkor hozzáfűzőm az eddigi elérési úthoz. Hasonlóan, ha a kapott kiterjesztés nem null, akkor egy ponttal hozzáillesztem a kapott modell *Name* property-ének végére. A kapott URL-t és nevet összefűzőm és visszatérek.

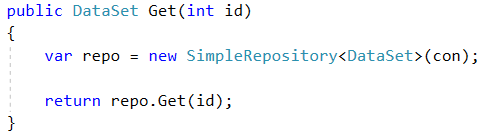
A GetFile megvizsgálja, hogy létezik-e a visszakapott elérési út, majd a ReadAllBytes metódussal először byte tömbbe, majd azzal egy MemoryStream-be töltöm azt, ezután visszatérek vele és a fájl nevével.

### API projekt

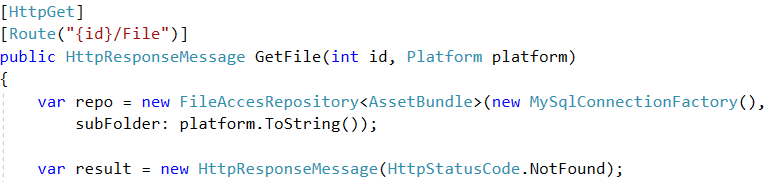
Az API rengeteg és sokféle fájlt tartalmaz, ám ennek nagyrésze a projekt generálásakor keletkezett. Ami számomra, illetve a dokumentáció szempontjából érdekes, azok a Controllers mappában elhelyezkedő osztályok. Controllerek file-al

#### Controllerek

Az API-hoz tartozó Controllerek csak Get-eket tartalmaznak, mivel a SimpleRepository és FileRepository is implementálja a Post, Put és Delete igéknek megfelelő műveleteket, ezért könnyen bővíteni lehetne vele az API-t, de mivel nem volt szükség ezekre a végpontokra, ezért nem implementáltam őket.



A legtöbb esetben a logika döntő többsége a repository-ban rejlik, ezért ezeket az eseteket nem érdemes bemutatnom. A különleges eseteket a file-ok lekérése jelenti, ezért erre bemutatok egy példát.



Ezeknek a kéréseknek explicit megadtam az elérési útjukat, hogy a REST konvenciókat be tudjam tartani. Az AssetBundle nem platform független, ezért egy plusz paramétere is van a *platform*, amit egy enum érték reprezentál. A metódus a repository létrehozásával indul, a konstruktorának átadom a platform paraméter, így például az Androidhoz tartozó fájlokat az Android almappában keresi majd. Létrehozok egy választ, kezdésnek NotFound értékkel.



Megvizsgálom, hogy a megadott platform helyes-e, ha nem az visszatérek, így a hívás eredménye NotFound lesz.



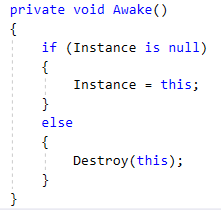
A repository-tól lekérem a kapott Id-hez tartozó fájlt. Ha a fájl létezik, a válasz tartalmaként beállítom azt, mint csatolmány.

A másik két fájlokat visszaadó Controller logikája nagyon hasonló, ezért nem mutatom be őket.

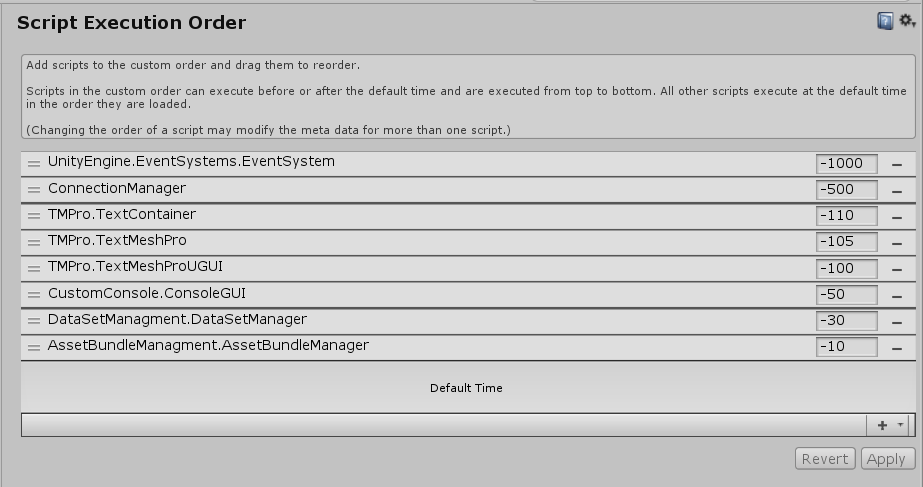
## Kliens

### Singleton tervezési minta

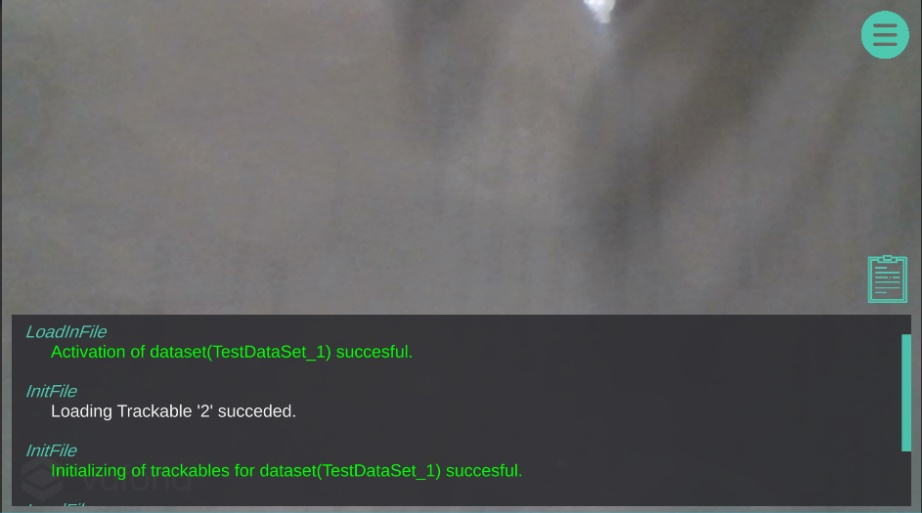
A kliensben több osztály, nevezetesen a AssetBundleManager, ConnectionManager, ContentTabBar, ConsoleGUI, DataSetManager, DllManager osztályok. Ennek a tervezés mintának a célja az, hogy a program minden pontjáról elérhető osztályt hozzon létre, melyből csak egy példány lehet a program futása alatt. Ezt tipikusan úgy érjük el, hogy létrehozunk egy statikus adattagot, melyet csak az osztály módosíthat. Az osztály függvényeit ezen az adattagon keresztül hívjuk meg. Általában C# nyelvben ez a példány egy privát setter-rel rendelkező property lenne, melyet a get-ben ellenőrzünk, hogy null-e és ha igen beállítjuk egy új példányra, ami a program futásának végéig él. Ebben az esetben a konstruktort privátra kell állítani, hogy a program más pontján ne jöhessen létre példány.



Mivel Unity-ben a script-eket nem a konstruktorukkal hozzuk létre, ezért más megoldást kell találni. Az általam használt singleton-ok az Awake metódusban vizsgálják, hogy létezik-e már a példány, és ha igen elpusztítják magukat. Ennek megközelítésnek az egyik hátránya, hogy a példánynak muszáj jelen lennie a játéktérben és már inicializálva kell lennie, mikor először hivatkozik rá egy másik script. Annak, hogy ezt biztosítsuk az egyik módja, hogy ha módosítjuk a script-ek végrehajtási sorrendjét. Ezt az „Edit>Project Settings>Script Execution Order” alatt tehetjük meg, ahol a singleton osztályokat a Default Time blokk elé kell helyeznünk.

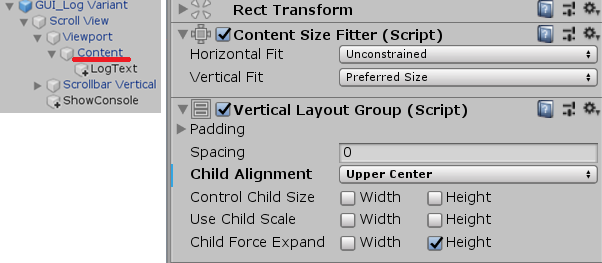


### ConsoleGUI

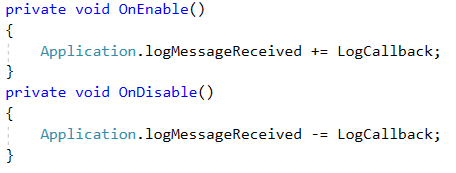


A program tartalmaz egy konzol ablakot, ezt azért készítettem, hogy elősegítse a hibák okának feltárását, illetve könnyebben megbizonyosodhassak a helyes működésről. A Unity ad lehetőséget arra, hogy a programunkból egy fejlesztői verziót készítsünk, amit lehet csatlakoztatni a Visual Studio-hoz, ha a telefont a számítógépre kötjük, de ezt igen nehézkesnek találtam, ugyanis a program nagyságrendekkel lassabb lett tőle. A konzol jobb felső sarkánál lévő gombbal el is lehet rejteni, majd újra megjeleníteni azt, valamint van egy beállítás, ami induláskor elrejti, ha szeretnénk.

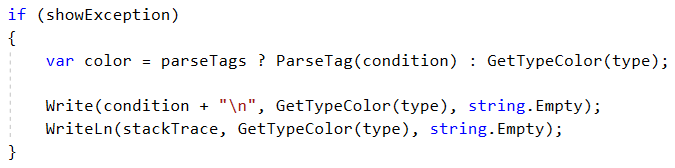
#### Megvalósítás



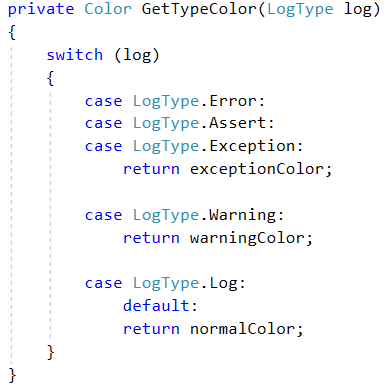
A konzol megvalósításához olyan UI elemeket használtam, melyeket a Unity alapból biztosít – a TMPro esetében egy csomagon keresztül. Először is leraktam egy ScrollView elemet, ennek csak a vertikális görgősávja kell. A „Content” nevű GameObject gyerekeként elhelyeztem egy TMProText címkét. Ez egy olyan UI elem, ami szöveg megjelenítésére alkalmas az alapvető Unity-s címkével ellentétben jól méretezhető, valamint kezeli a rich text-et – azaz könnyedén formázható tagek segítségével. A Content GameObject-re helyeztem egy ContentSizeFitter, mely a görgethető terület méretezését végzi majd, valamint egy VerticalLayoutGroup-ot. A ContentSizeFitter a VerticalLayoutGroup-ról kéri le, hogy mi az aktuális ideális mérete.



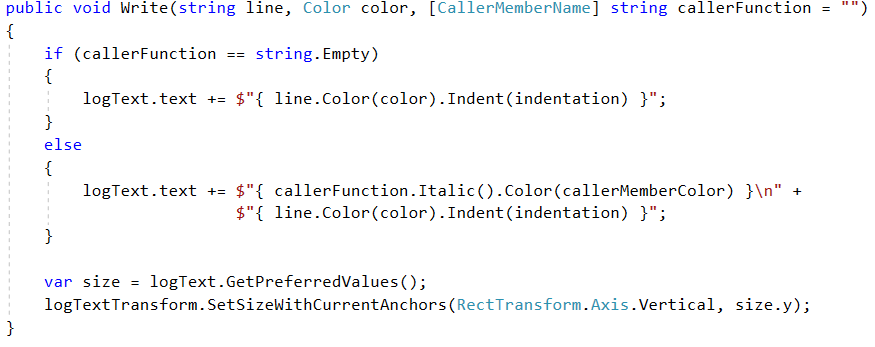
A Unity biztosít egy eseményt, ami a konzolra íráskor hívódik meg és tartalmazza az üzenetet, a stacktrace-t és a log fajtáját. Erre az eseményre iratkozunk fel a ConsolgeGUI aktiválódásakor, majd le a deaktiválásakor.



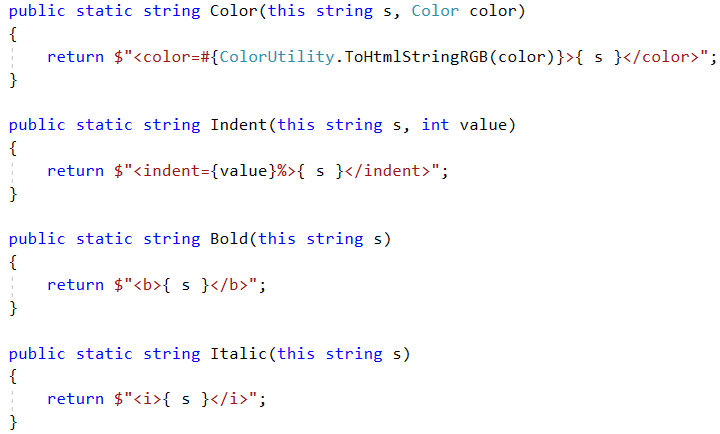
A függvény a log típustól függően 3 ágba kerület – exception, warning, debug –, ezek log-olását külön-külön ki lehet kapcsolni a komponens beállításainál. Az ágakban lévő kód igencsak hasonló, ezért csak egyet mutatok be ezekből. A *color* változó megállapítása attól függ, hogy a *parseTag* paraméter igaz-e. Ha igaz, akkor a *color* az üzenet elején lévő „#tag#” értéktől függ. Jelenleg a *parseTag* mindig hamis, ugyanis a ParseTag függvény hibás és nem fért bele az időkeretembe a javítása. Ennek köszönhetően mindig a A GetTypeColor függvényt hívja meg a kód.



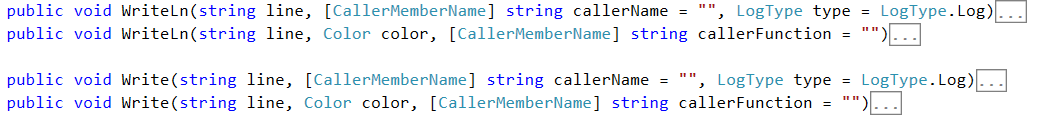
A GetTypeColor a komponens paramétereinek megfelelően tér vissza egy egyszerű switch-case-t alkalmazva, mely a típustól függően tér vissza különböző színekkel. Miután megvan a megfelelő szín meghívom a Write függvényt, melynek utolsó paramétere üres szöveg lesz, ugyanis oda függvényt meghívó függvény neve kerül, ha nincs felülírva, de a LogCallBack-et felesleges kiíratni.



Ha a hívó függvény neve üres – ez akkor következik be, ha szándékosan üres szöveget adok át, mint a LogCallBack esetében is – akkor csak az üzenetet iratom ki, ha nem üres, akkor a hívó függvény nevét is megjelenítem. A szöveget mindkét esetben formázom is rich text-el. Az erre használt függvényekre még visszatérek. A szöveget megjelenítő UI elemet ezután át kell méretezni, hogy a görgethető terület is vele nőjön.



Azért, hogy gyorsabban és olvashatóbban lehessen formázni a log-ok szövegét készítettem pár bővítő függvényt, melyeket szövegekre lehet meghívni és a megfelelő tageket hozzácsapva a szövegrészhez egy új, formázott szöveget adnak végeredményűl. Ilyen módon a formázásokat összelehet fűzni egymás után, tömören, egyszerűen, olvashatóan leírva azokat.

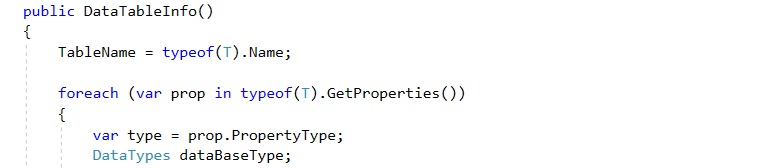


A Write-hoz több alternatíva is készült, melyekkel lehet közvetlenül log-olni a konzolra.

### Lokális adattárolás

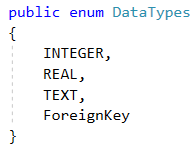
#### DataTableInfo

A DataTableInfo azzal a céllal készült, hogy a kliens egyszerű sql utasításait automatizálni tudjam. Az osztály képes a megadott adatbázis tábla modell alapján többek közt a tábla létrehozásához, sor beillesztéséhez vagy felülírásához, törléséhez stb. sql kódot generálni. Ezen funkciók közül végül csak a tábla létrehozása, valamint az adott Id-vel rendelkező sor beillesztése vagy felülírása lett kihasználva, mivel a helyi adatbázis csak az API-n található adatok lementésére szolgál és sosem ír felül, illeszt be vagy töröl sorokat.



A DataTableInfo egy sablon osztály egy típussal, ezért a konstruktor feladata, hogy feltérképezze a kapott osztályt, hogy annak, valamint adattagjainak nevét felhasználva tudjon sql utasításokat generálni.

A tábla neve a kapott típus neve lesz, az oszlopait pedig reflexióval kérem le, majd azokon iterálva eldöntöm, hogy milyen típus reprezentálja azt az SQLite adatbázisban.



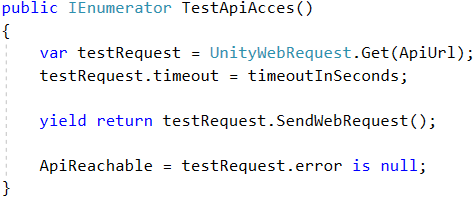
Az adattagok típusai a következő szerint alakulnak; ha típus int, akkor INTEGER, ha float, akkor REAL, ha implementálja az IDataModel interfészt, akkor ForeignKey, minden más esetben TEXT.



Az így kapott értékeket egy listához adom, amiben a reflexióval megkapott adattagok és a hozzájuk tartozó SQLite típus párok találhatók.

#### ConnectionManager

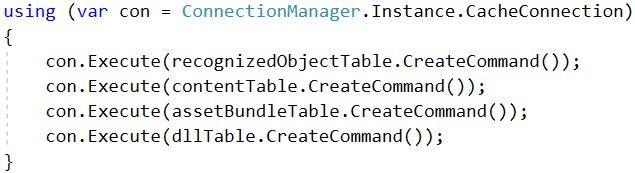
A ConnectionManager szintén singleton, azért felel, hogy az API vagy gyorsítótár elérési útját ne kelljen több helyen is kifejteni, ezzel nehezen módosítható konfigurációt létrehozni. Ehelyett ennek komponensnek a szerkesztőben elérhető paraméterei segítségével könnyen, egy helyen lehet konfigurálni ezeket az URL-eket.



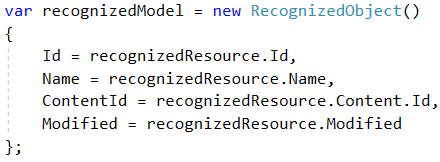
A másik fontos feladat a komponensnek, hogy az API elérhetőségét számon tartsa. Ehhez egy Coroutine-t használok, ami megvizsgálja, hogy az API címe elérhető és ennek megfelelően beállítja a példány *ApiReachable változóját.*

#### Repository

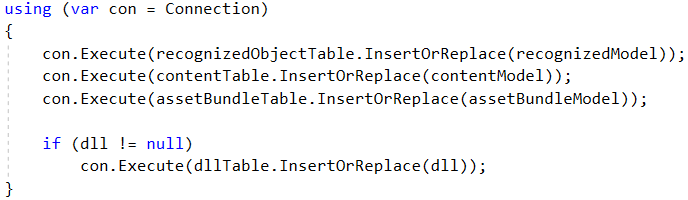
Az összes adattáblához, tehát AssetBundle, DataSet, Dll, RecognizedObject tartozik egy repository. Egy repository célja, hogy elrejtse az adat mentésének és kinyerésének komplexitását. A 4 repository közül a RecognizedObject tűnik ki, mivel ennek az adatai nem csak egy egyszerű táblában vannak eltárolva, hanem ezen felül másik 2 tábla összekapcsolásával kapható vissza, valamint GetAll helyett Get függvénnyel rendelkezik. Mivel az Api RecognizedObject típusú objektumot ad vissza, ezért a lementéskor ezt táblákra kell bontani.



A RecognizedObjectRepository konstruktorában létrehozom az összes táblát, amitől az osztály függ. Ezt egyébként a program futása során csak egyszer teszem meg, ugyanis egy statikus változóhoz van kötve. Ezt azért találtam fontosnak, mivel minden dinamikusan elhelyezett felismerhető objektum létrehoz egy példányt a repository-ból, így akár 100-1000x is lefuthatna feleslegesen a kód.



A Cache függvény paraméterként egy RecognizedObjectResource-t kap meg, melyből az adatokat kézzel másolom át a megfelelő táblák modelljében.



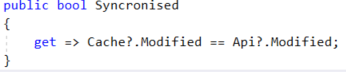
A kinyert modelleket rendre beillesztem a megfelelő táblába. Az InsertOrReplace beilleszti az új adatokat, vagy felülírja, ha már létezett sor a táblában a megadott azonosítóval.

A Get függvény lényegében azonos az API-n fellelhető RecognizedObjectRepository megegyező nevű függvényével.

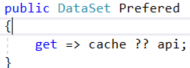
### DataSet kezelés

#### DataSetInfo\_Model

Ennek az osztálynak a célja az, hogy könnyebben számon tartható legyen az adott DataSet állapota, azaz, hogy csak az API-n keresztül érhető-e el vagy a gyorsítótárban is jelen van. Az osztály továbbá azt is képes eldönteni, hogy a már letöltött adat naprakész-e vagy lehet frissíteni.

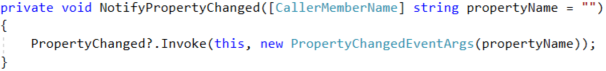


Ez a property akkor ad vissza igaz értéket, ha se a *Cache* se az *Api* nem null, illetve a *Modified* értéke megegyezik. Tehát ha tudjuk, hogy az API elérhető, biztos, hogy van értelme letölteni az új adatokat.

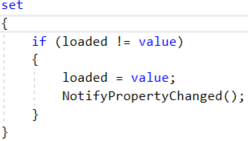


A *Prefered* property kényelemből lett hozzáadva, így olvashatóbb kódot lehet írni, mint ha minden alkalommal ellenőriznem kéne, hogy van-e gyorsítótárazott adat.

Ezek mellett az osztály még azt is tárolja, hogy be lett-e már töltve a Vuforia motor által a DataSet melyről hordozza az információt. A DataSetInfo-hoz párhuzamosan létrehoztam egy osztályt, ami a megjelenítésérét felelős. Ahhoz, hogy a megjelenítés gördülékenyen menjen implementáltam az INotifyPropertyChanged interfészt.

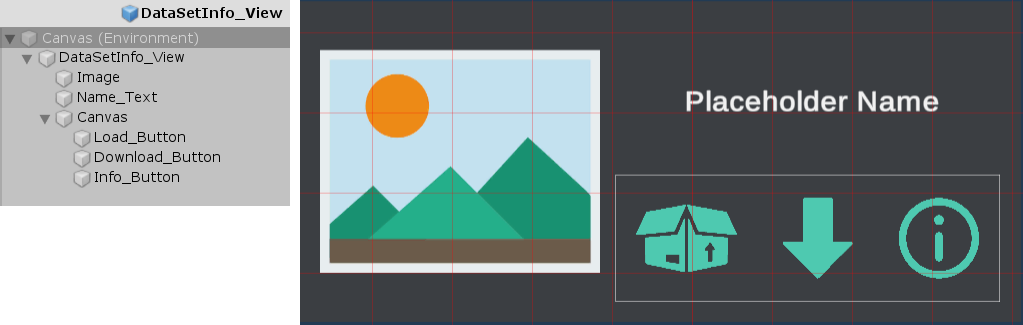


Az interfésznek arra szolgál, hogy értesítést tudjunk küldeni, ha a példány egy property-ének értéke megváltozik.

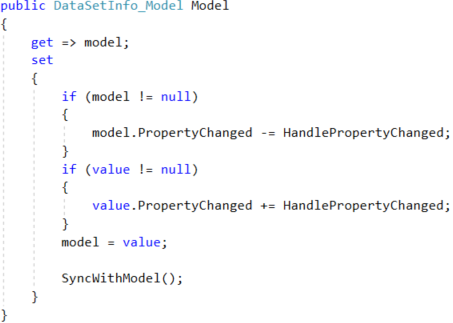


Azokat a property-ket, amikről értesítést szeretnénk küldeni úgy kell megírnunk, hogy a set részben megvizsgáljuk az új érték különbözik-e a régitől, majd, ha igen, akkor meghívjuk az interfészhez tartozó függvényt. A függvény egyetlen paramétere a hívó függvény neve, ami így a property neve lesz.

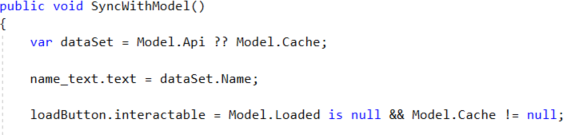
#### DataSetInfo\_View



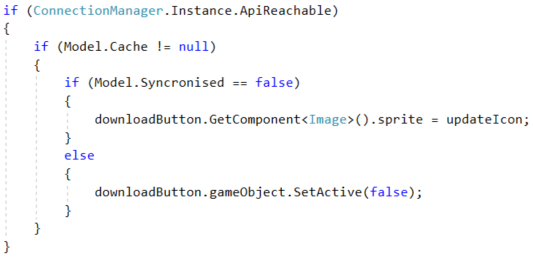
A DataSetInfo\_View a DataSet-ek megjelenítésért felelős komponens. Ahhoz, hogy megfelelően működjön szükséges hozzá a képen látható hierarchia és elrendezés. Három gombot tartalmaz, a betöltés, a letöltés/frissítés és a részletes információk gombját. A felhasználói felület tartalmaz egy képet is ám ez nem kapcsolható valódi adatokhoz.



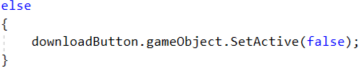
A View leköveti a Model értékeinek változásait. Ezt úgy értem el, hogy a *Model* property beállításakor feliratkozok a modell PropertyChanged eseményére a HandlePropertyChanged függvénnyel. Ha a *Model* előzőleg nem null volt, akkor szükséges leiratkozni a korábbi érték eseményéről. Mivel legközelebb csak akkor fut le az eseményre feliratkozott függvény, ha megváltozik a modell értéke, így ilyenkor szükséges a View-t a modellel szinkronba hozni.



A szinkronizáció során először meghatározom, hogy melyik modellt jelenítem meg. Ez az online modell, ha elérhető, ha nem akkor a gyorsítótárazott verzió, más lehetőség nem lehetséges, ugyanis, ha mindkettő érték null lenne, akkor nem létezne DataSetInfo ilyen értékkel. A betöltés gombja akkor csak akkor elérhető, ha van gyorsítótárazott modell, azaz a hozzátartozó fájl is jelen van a készüléken és még nincs betöltve.



A letöltés gomb már bonyolultabb, ha van lokálisan tárolt adat, akkor az a kérdés, hogy az naprakész-e, ha nem, akkor az ikont a letöltésről frissítésre állítom. Ha van helyi adat és naprakész, a gombot elrejtem, hiszen nem lát el semmilyen funkciót a panelen.



Ha az api nem elérhető, de nincs helyi adat – ez akkor fordulhat elő, ha elmegy az internet a program megnyitását követően –, akkor a gombot elrejtem, hiszen a letöltés hibára futna kapcsolat hiányában. A HandlePropertyChanged paraméterként kapja a megváltozott property nevét, illetve a modellt *sender* néven. Ezt a két adatot felhasználva egy switch-case formájában könnyen kezelhető a nézet frissítése. Fontos megjegyezni, hogy a megjelenített nézeteket az őket felhasználó menü minden megnyitásakor újra generálom. A property-k változásait kezelő kódrészletek hagyatkoznak erre a tényre, így nem kezelnek bizonyos helyzeteket, melyeket egyébként kellene, például azt amikor az api elérhetővé válik az elemek létrejötte után, ugyanis ez a menü következő megnyitásáig nem frissülhet.

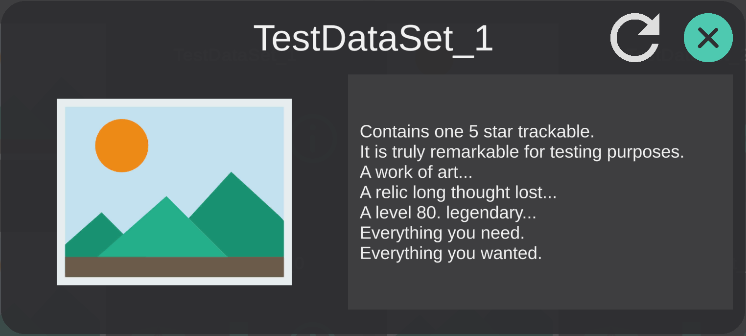


A *Loaded* esetében a hozzátartozó gomb akkor interaktálható, ha az null.



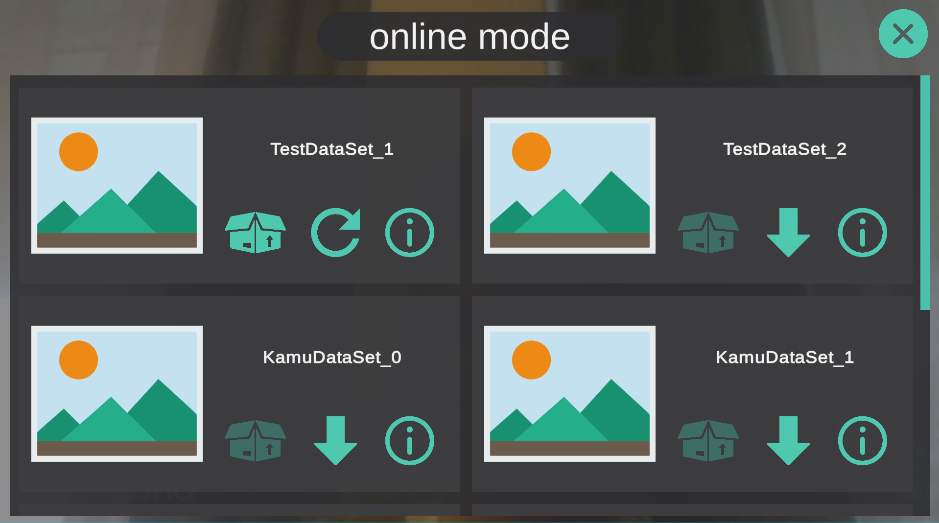
Ha a modell szinkronizálva van, akkor eltüntetem a letöltés gombot, mivel az már feleslegessé vált.

#### DataSetInfoDetails\_View



A DataSet-hez létezik egy részletesebb nézet is, mellyel összehasonlíthatjuk a naprakész és a letöltött adatokat (ha van különbség), valamint egy részletesebb adatlapot mutat, ahol elérhető a hozzátartozó leírás is. Ez a nézet nem alkalmazza a PropertyChanged eseményt, csak nagyon ritka esetben módosulnak az adatok a megnyitás és a bezárás között, valamint nem is olyan létfontosságú azok naprakészsége, mint a másik nézet esetén. Az osztálynak szintén van egy függvénye, amivel a megfelelő elemeket beállítja a modell adatai alapján. Ha mindkét modell elérhető megjelenik plusz gomb, mellyel a két leírás között lehet váltani.

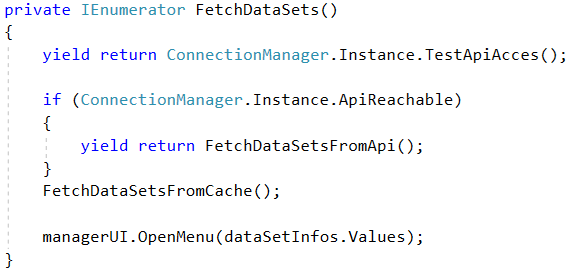
#### DataSetInfoManagerUI



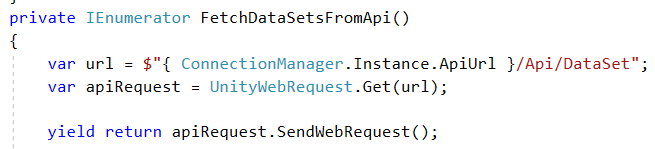
Az osztály DataSetInfo\_View-k kezelésére szolgál. Egy publikus függvénye van, ami megadott elemek egy listáját képes megjeleníteni. A függvény futásakor a listához tartozó Canvas-t aktívvá teszem, ezzel megjelenítve a listát. Az api jelenlegi elérhetőségének megfelelően az „offline” vagy „online” felirat kerül a képernyő tetejére. Törlöm a ScrollView Content Transform-ja alatt lévő elemeket, majd újakat hozok létre kapott modellek alapján. Az DataSetInfo-k kezelését innentől kezdve a létrehozott view-k végzik.

#### DataSetManager

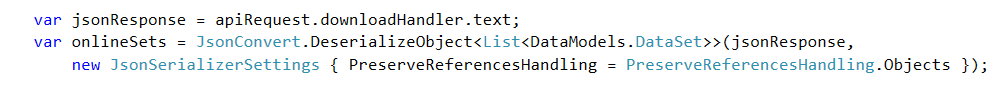
A DataSetManager osztály egy singleton, mely 2 fontos publikus függvényt biztosít. Mind két függvényt gombok segítségével hívom meg, most ezeket fogom bemutatni. Az első az OpenDataSetMenu, minek célja, hogy betöltse a rendelkezésre álló DataSet-eket – legyen az online vagy offline –, majd egy másik osztály – a DataSetManagerUI –segítségével megjelenítse azokat. A függvény meghívása elindítja a FetchDataSets Coroutine-t.



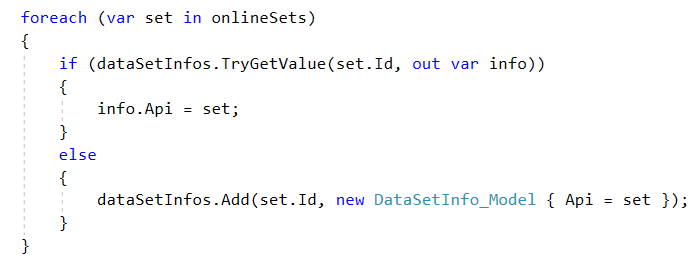
Az első lépés ellenőrizni az Api elérhetőségét. Ha elérhető, akkor lekérem az API-n jelenlevő DataSet-ek – ezt lent részletesebben bemutatom – információit is. Ha nem tudtam elérni az API-t, akkor csak a gyorsítótárazott adatokat töltöm be. A betöltött adatokat ezután a felhasználó felületért felelős osztály jeleníti meg.



Első lépésként beállítom az api híváshoz tartozó elérési utat. Ezt megadva létrehozok egy Get-et használó UnityWebRequestet. Mivel visszatérek a SendWebRequest függvénnyel, ezért nem blokkolom a szálat, a Coroutine futása a válasz megérkezése után folytatódik.



A válasz megérkezése után JSON .Net segítségével vissza alakítom a szöveget objektumokká. A második paraméter azért szükséges, hogy a visszaalakításkor megmaradjanak a referenciák, így, ha sok az ismétlődő hivatkozás, memóriát nyerhetünk.



A lekért adatokon végig iterálok, és ha még nem adtam őket hozzá a *dataSetInfos* dictionary-hez, akkor hozzáadok egy új DataSetInfo példányt, melynek az Api property-ét a lekért adatra állítom, ha már volt akkor a meglévő példány értékét állítom át. A FetchDataSetsFromCache függvény csupán annyiban tér el, hogy az adatokat a repository-ból kérem le a GetAll függvény segítségével, valamint a visszakapott adatokat a DataSetInfo Cache property-ének állítom be.

Másik fontos függvény az OpenFile, ami a DataSet fájl betöltését végzi el – ha szükséges le is tölti azt. A függvénynek paraméterként megadható, hogy hagyja figyelmen kívül a gyorsítótárat, ilyenkor mindenképpen letölti az API-ról az adatokat felülírva a meglévőket, abban az esetben, ha ez hamis, csak akkor tölti le az adatokat, ha a fájlokat nem találja a megfelelő mappákban. Emellett van egy másik opcionális paraméter, egy visszatérési érték nélküli, bool értéket váró függvényt meglehet adni, mely lefut a betöltés végén, megkapva, hogy sikeres volt-e. Az, hogy a folyamat sikeres volt-e a LoadInFile visszatérési értékétől függ.

A LoadInFile egy DataSet modellt vár paraméternek, aminek segítségével az alkalmazásnak fent tartott mappából betölti a DataSet-et, ha elérhető. Kezdetnek megvizsgálom, hogy a fájl nincs-e már betöltve, ha igen hamissal visszatérek.



Egyébként lekérem a Vuforia ObjectTracker-jét, ez azért szükséges, mivel ezen keresztül lehet új DataSet-eket hozzáadni. Az *info.Loaded* DataSet-re meghívom a Load függvényt, mely egy elérési utat vár, ahonnan betöltheti a tanított fájlt. Ha sikerrel jár, akkor le kell állítani az *ObjectTracker*-t a Stop függvényével, és valamint az Activate metódussal aktiválni a DataSet-et. Ezen a ponton már biztos, hogy a fájl megfelelő, csupán akkor lehet sikertelen a betöltés, ha a betöltött felismerhető objektumok száma meghaladná a limitet. Ha a DataSet-et valamilyen oknál fogva nem sikerült betölteni, akkor azt kiírom a ConsoleGUI segítségével, illetve azt is nyugtázom, ha sikeresen betöltődött. Sikeres betöltés után az *ObjectTracker*-t újra kell indítani.

A betöltött DataSet-et inicializálni kell, elhelyezni a kívánt tartalmat a követhető objektumot alatt. A Vuforia egy singleton osztály, a TrackerManager segítségével lehetőséget biztosít arra, hogy lekérjük az összes betöltött követhető objektumhoz tartozó GameObject-et. Amikor ezeken végig iterálunk, onnan tudhatjuk, hogy az aktuális Tracable-t most helyeztük le, hogy a hozzátartozó GameObject neve „New Game Object”. Ezt a nevet én a tanított fájlban szereplő névre cserélem, ami rendre megegyezik az adatbázisban levő RecognizedObject tábla egy-egy sorának azonosítójával. Az elnevezett GameObject-hez hozzáadok egy DefaultTrackableEventHandler valamint egy ContentHandler komponenst, utóbbinak elindítom az Inicializálásért felelős Coroutine-ját, ezt részletesebben a ContentHandler leírásánál fogom kifejteni.

### AssetBundle kezelés

#### AssetBundleInfo

Az AssetBundleInfo sokban hasonlít a DataSetInfo-hoz, ez az osztály hasonló szerepet tölt be az AssetBundle-ök kezeléséhez, mint a másik a DataSet-ek kezeléséhez az UI értesítésektől és egy másik fontos különbségtől eltekintve.



A fő eltérés az, hogy az AssetBundle betöltésére jöhet egymás után több kérés, hiszen a DataSet több elemet tartalmaz, az inicializálás után ezek mindegyike kérést indít az AssetBundleManager felé, hogy az ő hozzá tartozó tartalmat betölthesse, ám ez a tartalom helyezkedhet el ugyanabban az AssetBundle-ben vagy akár meg is egyezhet. Emiatt az AssetBundleInfo a rendelkezik egy listával, amihez a betöltést követő teendők hozzá vannak fűzve, hogy biztosan minden kérés ki legyen szolgálva.

#### AssetBundleManager

Az AssetBundleManager működése nagyon hasonló a DataSetManager-éhez, a komponens induláskor szintén lekéri az API-n és a gyorsítótárban elérhető modelleket, valamint ezeket egy AssetBundleInfo-kat számon tartó dictionary-be rendezi.

Az Manager egyetlen publikus függvénye, a UseBundle egy AssetBundle modellt és egy függvényt vár, melyet meghívhat, ha végzett a betöltéssel. Ha a modellhez tartozó AssetBundle már be lett töltve, akkor rögtön meghívja a paraméterben kapott függvényt. Ha még nem lett betöltve, akkor a függvény az AssetBundleInfo teendők listájához fűzi. Ha a betöltés folyamat még nem indult el, akkor el is indítja azt. A betöltés nagyban hasonlít a DataSetManager-re a lényeges különbség, hogy a metódus végén végig iterálok a hozzáadott függvények listáján, és mindet meghívom, majd törlöm őket a listából.

### Dll kezelés

A Dll-ek kezelése is nagyban hasonló a DataSet-ekéhez, a betöltés szekvenciális, az adatok letöltésén és mentésén kívül semmi bonyodalom nincs benne.

### ContentHandler

A ContentHandler komponens a felismerhető objektumhoz tartozó tartalom betöltésért felelős. A DataSet betöltésekor a DataSetManager minden GameObject-re helyez egy ContentHandler komponenst és elindítja annak az Initialize Coroutine-ját.

Az inicializálás első lépése, hogy szövegből egész számmá alakítom a GameObject nevét, mivel ez egy RecognizedObject azonosítóval kell, hogy megegyezzen. Ha név értelmezhető egész számként lekérem az azonosítóhoz tartozó RecognizedObject-et. Ez az API elérhetősége esetén mindig onnan történik, ami után az adatok lementése is kerülnek. Ellenkező esetben a gyorsítótárból töltődik be. Ha a RecognizedObject létezik, valamint a hozzá tartozó tartalom igényel Dll-t, akkor azt betöltöm. Ha nem igényel Dll-t csak az AssetBundle betöltése indul el. Az AssetBundleManager UseBundle függvényét a LoadContent metódussal hívom meg. A LoadContent, ha nem üres AssetBundle-t kap vissza, akkor megpróbálja betölteni a GameObject-et a tartalom neve alapján, majd ezt az objektumot maga alá elhelyezi.

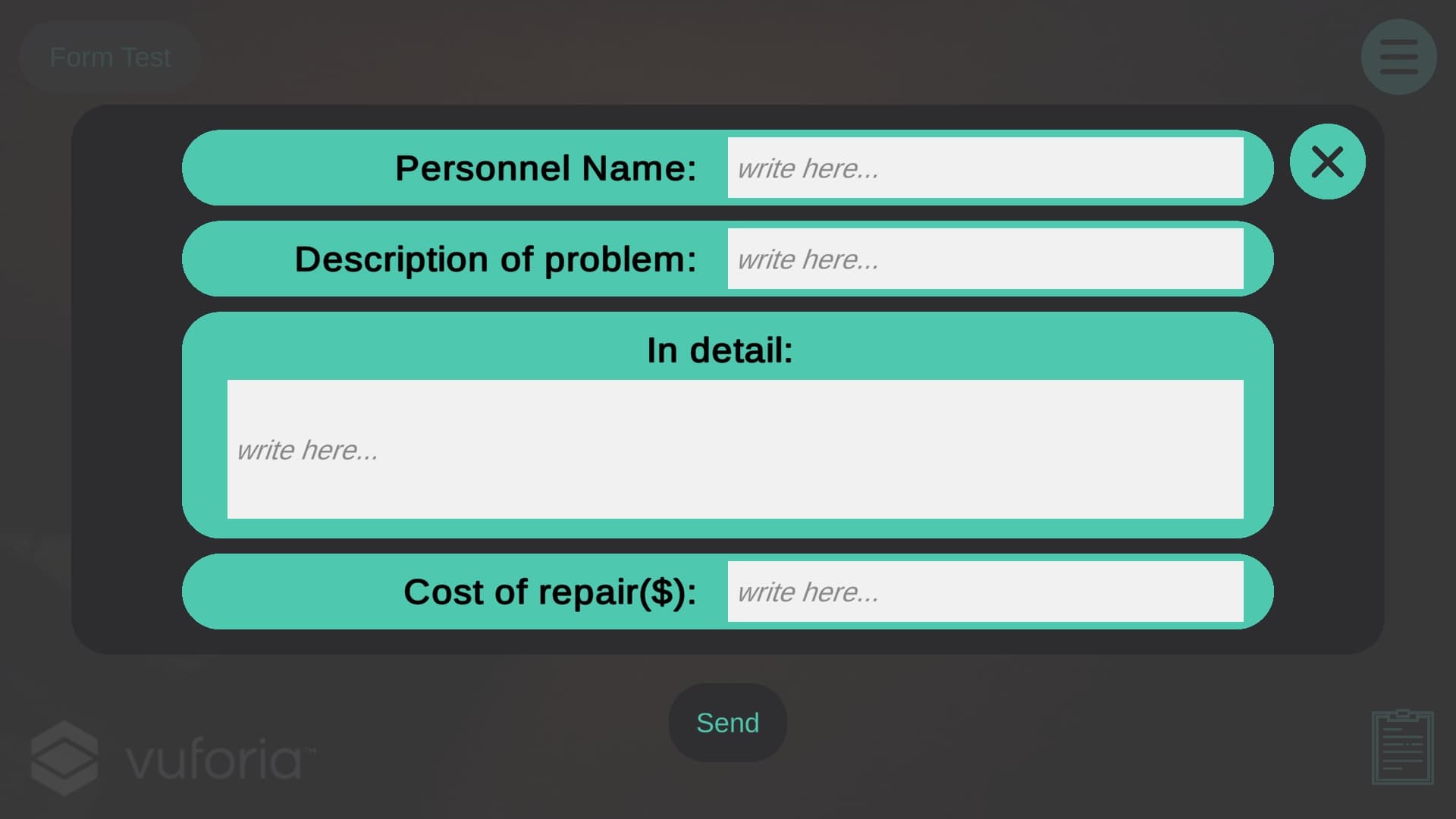
### ContentTabBar

A ContentTabBar koncepciója az, hogy biztosítson olyan felületet a bonyolultabb betöltött programoknak, ahol a felhasználó a felismerhető tartalmak hiányában is elérheti a funkciók egy részét. Például, ha a betöltött komponensnek szeretnénk biztosítani egy beállítások menüpontot, akkor azt a ContentTabBar-on megtehetjük. Az egész felépítése hasonlít a böngészőkben lévő fülekhez, ahol kiválaszthatjuk az aktív oldalt. A ContentTabBar-hoz egy publikus függvénnyel lehet új gombot adni. Ehhez szükségünk van egy ContentTab\_Model-re, ez tárolja a gomb feliratát, és két függvényt. Mindegyik metódus a modell *Open* property-jéhez van rendelve, az egyik akkor fut le, ha az érték igazra vált, míg a másik akkor, ha hamisra. Az *Open* változása akkor következik be, ha a jelenleg aktív gomb megváltozik. Ezt ahhoz hasonlítanám, mint amikor egy böngészőben oldalt váltunk. A böngészőkhöz hasonlóan, ha annyi ’oldalt’ nyitunk meg, hogy az már nem férne el a képernyő tetején, akkor görgethetővé válik fenti sáv.

A ContentTab\_Model a DataSetInfo\_Model-hez hasonlóan implementálja az INotifyPropertyChanged interfészt és tartozik hozzá egy View osztály, így kényelmesen lehet kezelni a modell és az UI szinkronizációját.

## Kliensben betölthető példa alkalmazás

#### Bemutatás

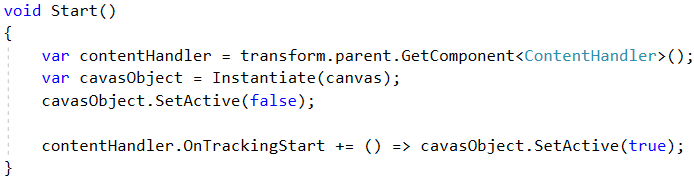


Némiképp összetettebb példának szántam a következő betölthető tartalmat, ami egy kitölthető űrlapot jelenít meg, ha felismeri a hozzátartozó objektumot. A kitöltött adatlapot aztán el is küldöm egy API-nak.

#### Megvalósítás

A példa alkalmazás külön Solution-ben készült, és kész dll-ként importáltam a Unity-be. Itt a komponenseket felhasználva elkészítettem a prefab-et, buildeltem az AssetBundle-t, majd áthelyeztem az Assets mappán kívülre, mivel nem kívánatos, hogy szerepeljen a buildelt alkalmazásban. Ilyen módon a komponens-ek arra a dll-re hivatkoznak, amit majd futás időben fogok betölteni és amíg a dll előbb töltődik be, mint a komponens a Unity képes helyesen betölteni az AssetBundle-ben lévő a program számára addig ismeretlen komponenseket is.

A példában különleges az, hogy a betöltött tartalom ugyan reagál a hozzá tartozó objektum felismerésére, de nem követi azt, valamint nem tűnik el, ha a felismert objektum többé nem található a kamera képén – mivel nagyban rontaná a felhasználói élményt. A betöltéskor a ContentHandler a tartalmat mindenképpen a követhető objektum-ot reprezentáló Transform alá helyezi el, ezért ez egy megoldandó probléma volt. Ezt a problémát úgy oldottam meg, hogy nem a megjeleníteni kívánt tartalomra hivatkozik a Content, hanem egy olyan komponensre, ami majd elhelyezi azt a játéktérben.



A komponens a létrejötte után lekéri a szülője, azaz a követhető objektum ContentHandler komponensét. Ezután a szerkesztőben beállított GameObject-et elhelyezem a játéktérben, mivel a Canvas úgy lett beállítva, hogy pozíciójától függetlenül rajzolódjon ki a képernyő fölé illesztve, ezért nem számít annak a pozíciója, csak elhelyezem. A ContentHandler-re azért van szükség, hogy az OnTrackingStart eseményre feliratkozhassak, és így tudjak reagálni arra, ha az objektum fel lett ismerve. Ebben az esetben a csupán arról van szó, hogy a tárgyat felismerte a Vuforia, akkor megjelenítem a Canvas-t. Ha a Canvas már eddig is jelen volt, akkor nem történik semmi, így a kezdeti felismerés után a felhasználó nyugodtan kitöltheti az űrlapot.



A ContentTab\_Model demozására egyszerű esetet készítettem, a MenuForm létrehozásakor hozzáad egy új ’oldalt’ a sávhoz, aminek megnyomására előugrik az űrlap. Mivel az űrlap betakarja az egész képernyőt, ezért nem lehetséges másik gombot megnyomni amíg az nyitva van, ezért nem is rendeltem semmilyen függvényt hozzá az esethez.



Ehelyett az űrlap bezárásakor az aktív oldalt – ami feltételezhetően az űrlap által elhelyezett gomb – nullra állítom. A ContentTabBar ezt a helyzetet automatikusan letudja kezelni, a korábban aktív gomb modelljének jelzi, hogy már nem aktív, ekkor az meghívja a bezáráshoz rendelt függvényt – ez jelenleg null, így nem történik semmi sem.