# 1.Bevezetés

# 2.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.

## 2.0. C#

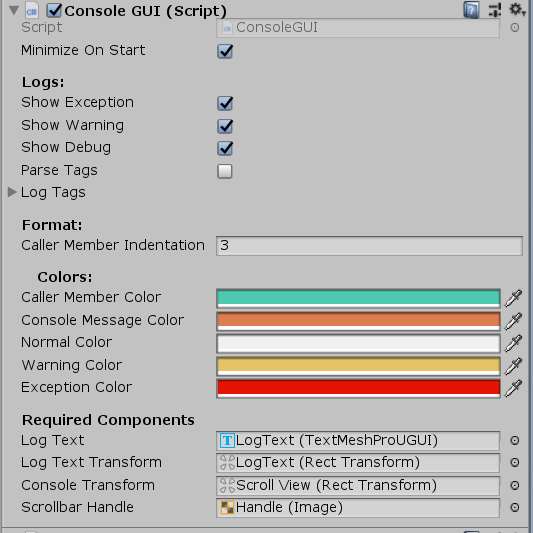
A C# egy erősen, statikusan típusos, objektum orientált nyelv mely leginkább a Java-hoz hasonlítható. A két nyelv eleinte kísértetiesen hasonlított egymásra, ám mostanra már külön vált a fejlődésük. A java a valódi cross-platform programozás nyelve, míg a C# a Windows fejlesztésé lett. A Microsoft az 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.

## 2.1. 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.

## 2.1.1. 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.

## 2.1.2. MonoBehaviour szerkezete

Ahhoz, hogy a kódunkat egy GameObjecthez 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.



## 2.1.3. Coroutine

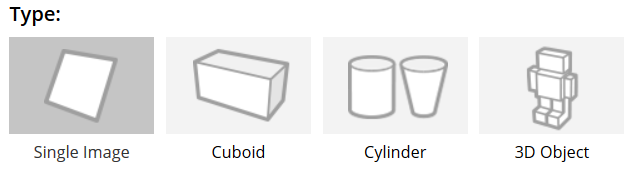
A MonoBehaviour-öket a Unity egy szálon kezeli, szekvenciálisan meghívva a hozzájuk tartozó különleges függvényeket. Ebből következik, hogy ha olyan kódot írunk, amelynek futtatása túl sokáig tart a programunk érezhetően meg fog akadni, így a program várakoztatására sincs lehetőség a hagyományos módon. 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.

## 2.1.4. AssetBundle

## 2.2. Vuforia SDK

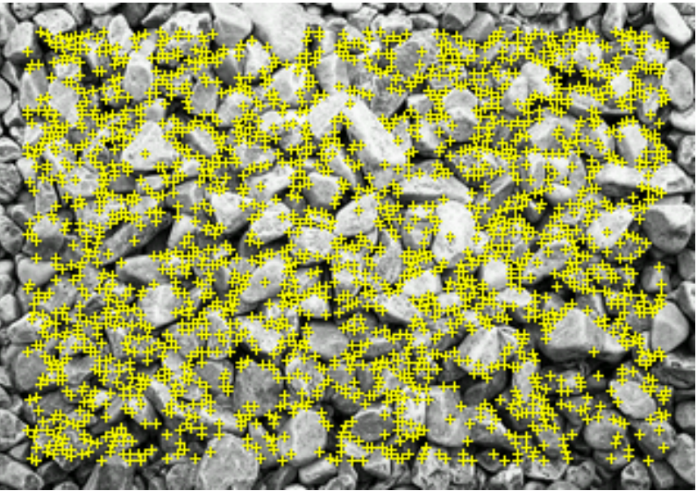
A Vuforia SDK célja, hogy mobilok kameráját és számítógépes látást használva felismerjen 2 és 3 dimenziós objektumokat, kövesse azok pozícióját, dőlését, méretét a kamerához képest. Az objektumokat jellegzetes pontjaik alapján képes felismeri. Ezeket a képességeket felhasználva a programok úgy tudnak 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.

## 2.2.1. Felismerhető objektum típusok



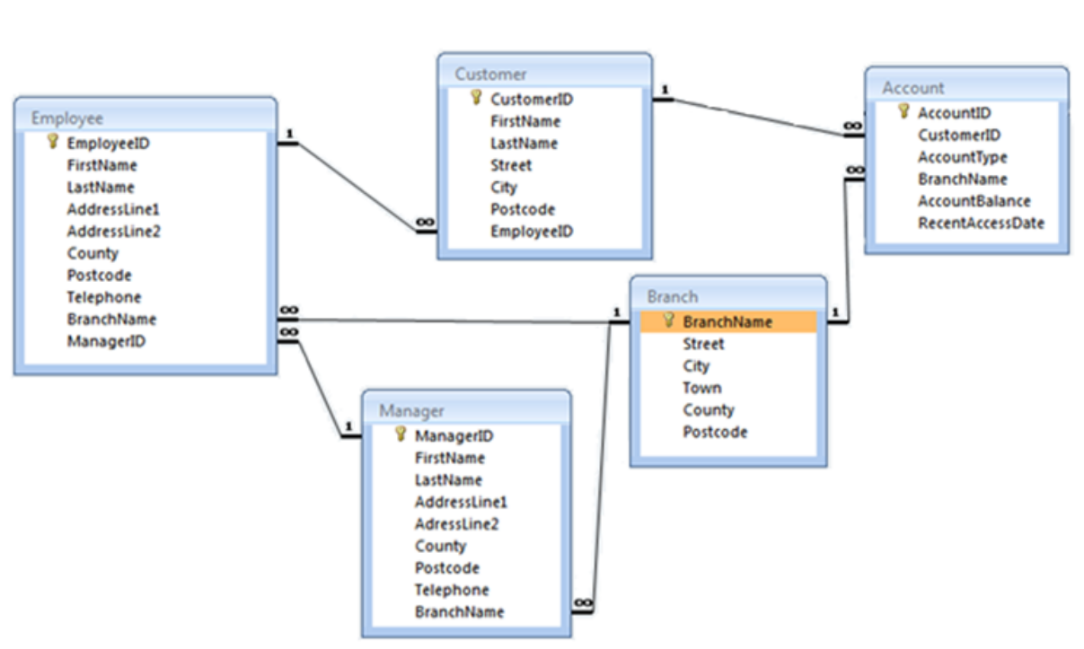
4 féle felismerhető és követhető objektum fajta van, mindegyik picit más alkalmazásra. Az egyszerű képet posztereket, kártyák, újságok és egyéb lapos tárgyakhoz ajánlatos használni. Hasábot olyan testekhez, melynek nem csak egy oldalán, ha minden a hat oldalán jellegzetes tartalom van. Hengert konzervek, 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.

## 2.2.2. Tanítás



Felismerhető objektumok adatait a Vuforia datasetnek 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.

## 2.3. 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.

## 2.3.1. MySql

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

## 2.3.3. 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, triggereket, 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.

# 2.4. 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.

## 2.4.1. 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.

## 2.4.2. REST

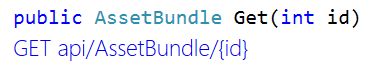
A REST szoftverarchitektúra nagy, hálózat alapú rendszer igényeit észben tartva jött létre, az internetét. Az architektúra az erőforrások fogalma köré épül, kliensekből és szerverekből áll. A kliens olyan kéréseket küld a szervernek, mely egy erőforrásra utal és azt vagy annak állapotának változását 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:

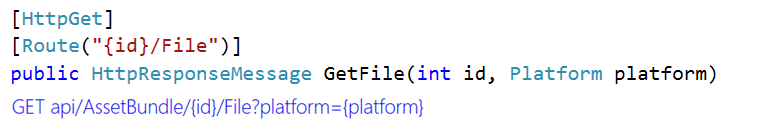
* kleins-szerver alapú
* állapotnélküli, tehát a szerver semmilyen plusz adatot nem tárol a kliens állapotáról, ez kizárólag a kliens feladata
* gyorsítótárazható, a kéréseknek megvannak jelölve, hogy a kapott válasz lementhető-e későbbre vagy sem
* unoform interfész, ahhoz, hogy ez létrejöhessen további négy megkötésnek kell megfelelni: a hivatkozott erőforrás a kérésből egyértelműen kiderül, erőforrás manipulálhatósága reprezentáció segítségével, minden üzenet önmagában elég információt tartalmaz az értelmezéshez, hypermedián 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

## 2.4.2. Controlerek és URL kezelés

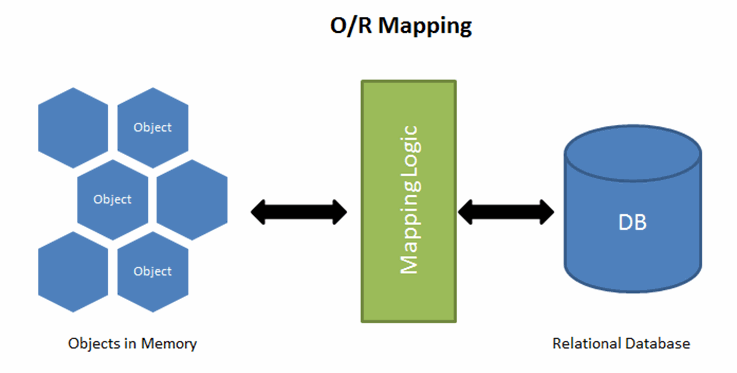
A web Api projekt legfontosabb elemei a controllerek, ezek olyan osztályok – tipikusan a „Controller” mappában elhelyezkedve – melyek az ApiController osztályból öröklődnek, és a nevük „Controller”-re végződik. A keretrendszer az osztályok nevét és a bennük lévő publikus függvények aláírását felhasználva 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.



## 2.5. Dapper



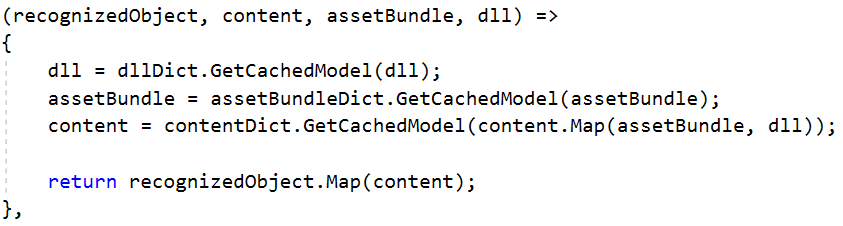
A Dapper ismertetéséhez szükséges bevezetnem az ORM fogalmát. Az ORM (Object Relation Mapping) magyarul objektum-relációs leképzés, két egymással nem kompatibilis objektum konvertálására használatos, ez első sorban az adatbázisból lekért táblák rekordjainak típusos objektummá való átalakítását jelenti. 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.

## 2.5.1. 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 a joinolt 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.



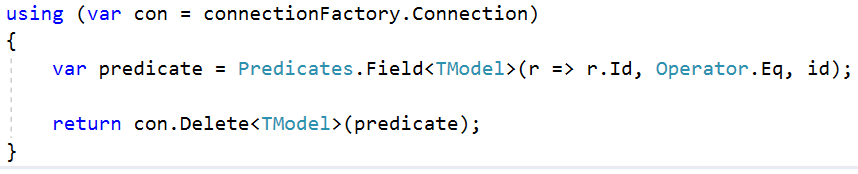
példa a típusok megadására



példa mapfunction-re

## 2.5.1. Dapper Extensions

A Dapper egy bővítményét is felhasználtam, melynek az a célja, hogy az alapvető SQL műveleteket reflectiont 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 predicatre

## 2.6. 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.

## 2.6.1. Json

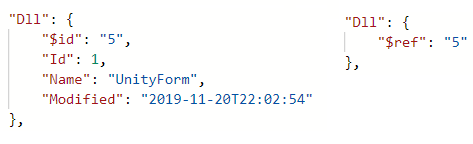


példa az apiban használt JSON-re

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.

## 2.6.2. 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.



# 3. Rendszerterv

## 3.1. Specifikáció

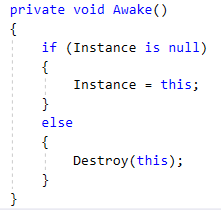
Cél olyan Android alkalmazás fejlesztése, mely a Unity és a Vuforia felhasználásával lehetővé teszi felismerhető tárgyak alapján interaktív tartalmak megjelenítését. A megjelenített tartalmakhoz és felismerhető tárgyakhoz tartozó adatokat mind szerveren keresztül érje el és töltse be, hogy ezek ne az 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.

# 4. Implementáció

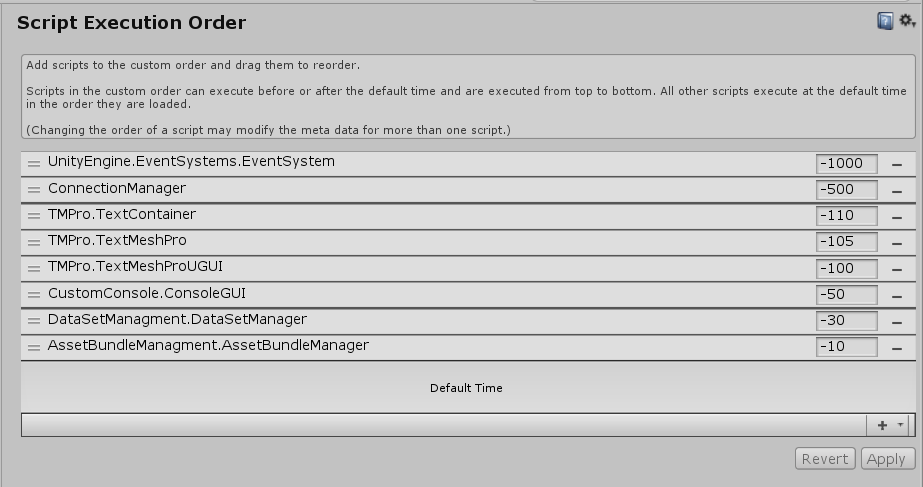
## 4.1. Kliens megvalósítása

## 4.1.1. Singleton tervezési minta

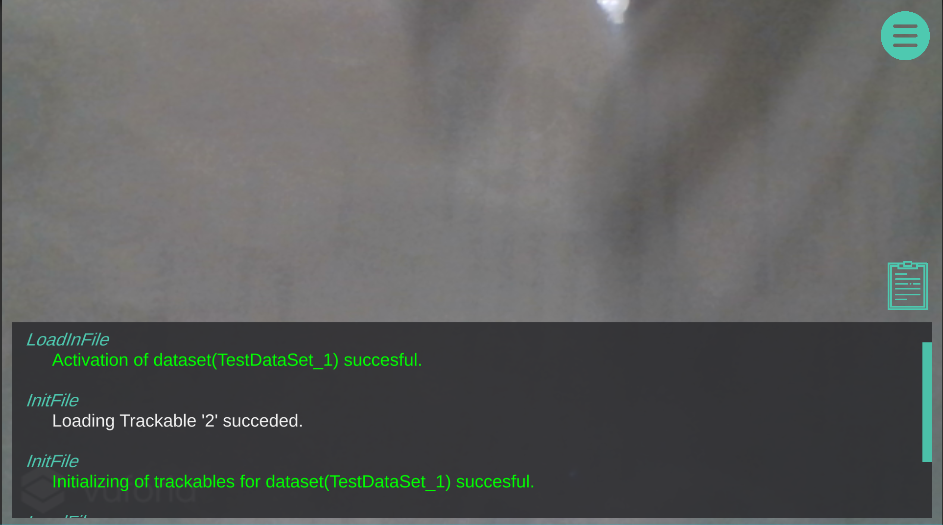
A kliensben több osztály, nevezetesen a AssetBundleManager, ConnectionManager, ContentTabBar, ConsoleGUI, DataSetManager, DllManager osztályok. Ennek a tervezés mintának a célja az, hogy a program minden pontjáról elérhető osztályt hozzon létre, melyből csak egy példány lehet a program futása alatt. 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 setterrel rendelkező property lenne, melyet a getben 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 szkripteket nem a konstruktorukkal hozzuk létre, ezért más megoldást kell találni. Az általam használt singletonok 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 szkript. Annak, hogy ezt biztosítsuk az egyik módja, hogy ha módosítjuk a szkriptek végrehajtási sorrendjét. Ezt az „Edit>Project Settings>Script Execution Order” alatt tehetjük meg, ahol a singleton osztályokat a Default Time blokk elé kell helyeznünk.

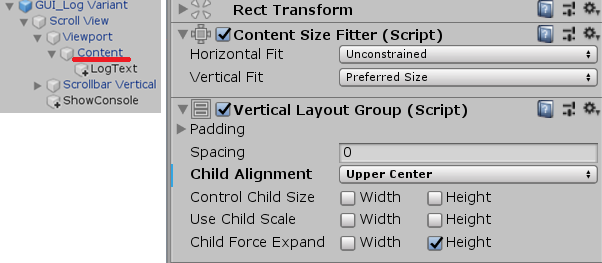


## 4.1.5. ConsoleGUI

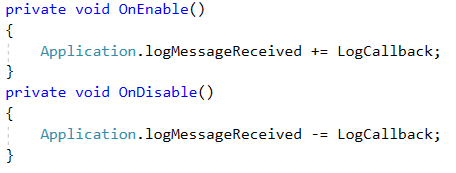


A program tartalmaz egy konzol ablakot, ezt azért készítettem, hogy elősegítse a hibák okának feltárását, illetve könnyebben megbizonyosodhassak a helyes működésről. A Unity ad lehetőséget arra, hogy a programunkból egy fejlesztői verziót készítsünk, amit lehet csatlakoztatni a Visual Studio-hoz, ha 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.

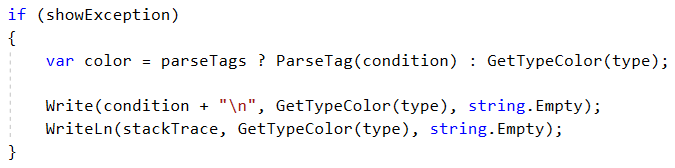
## 4.1.5.1. Megvalósítás



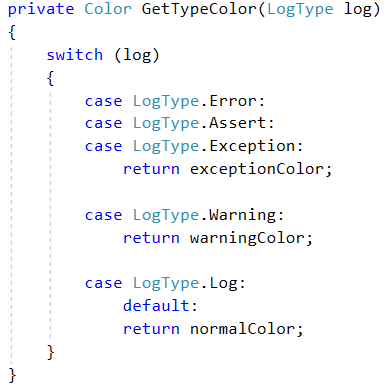
A konzol megvalósításához olyan UI elemeket használtam, melyeket a Unity alapból biztosít – a TMPro esetében egy csomagon keresztül. Először is leraktam egy ScrollView elemet, ennek csak a vertikális görgősávja kell. A „Content” nevű GameObject gyerekeként elhelyeztem egy TMProText címkét. Ez egy olyan 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 GameObjectre helyeztem egy ContentSizeFitter, mely a görgethető terület méretezését végzi majd, valamint egy VerticalLayoutGroupot. A ContentSizeFitter a VerticalLayoutGroupról kéri le, hogy mi az aktuális ideális mérete.



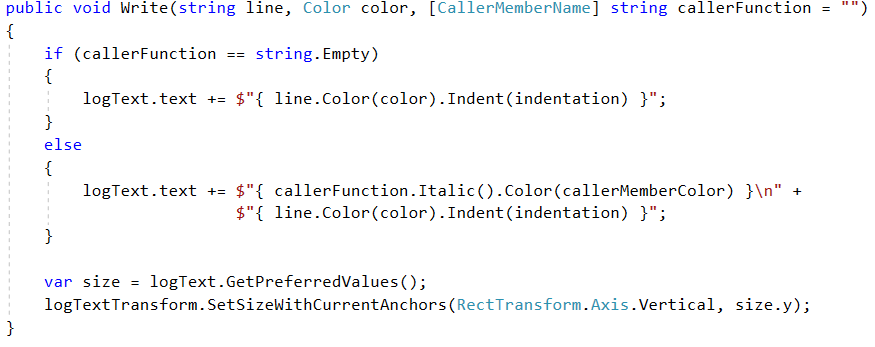
A Unity biztosít egy eseményt, ami a konzolra íráskor hívódik meg és tartalmazza az üzenetet, a stacktrace-t és a log fajtáját. Erre az eseményre iratkozunk fel a ConsolgeGUI aktiválódásakor, majd le a deaktiválásakor.



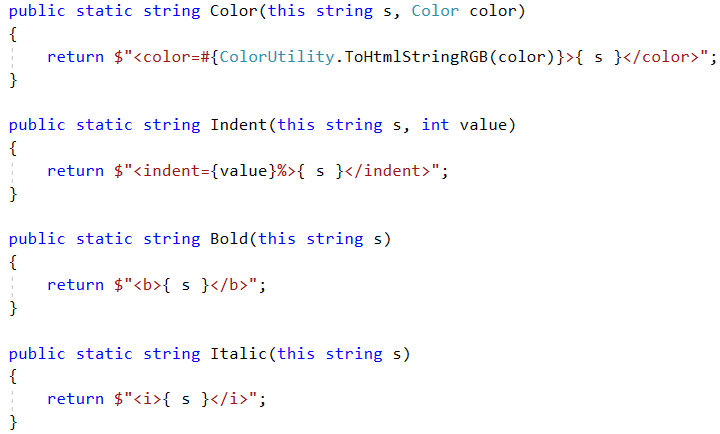
A függvény a log típustól függően 3 ágba kerület – exception, warning, debug –, ezek logolását külön-külön ki lehet kapcsolni a komponens beállításainál. Az ágakban lévő kód igencsak hasonló, ezért csak egyet mutatok be ezekből. A *color* változó megállapítása attól függ, hogy a *parseTag* paraméter 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.



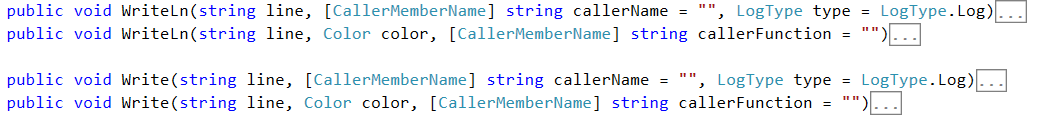
A GetTypeColor a komponens paramétereinek megfelelően tér vissza egy egyszerű swtich case-t alkalmazva, mely a típustól függően tér vissza különböző színekkel. Miután megvan a megfelelő szín meghívom a Write függvényt, melynek utolsó paramétere üres szöveg lesz, ugyanis oda függvényt meghívó függvény 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, mivel a felhasznált UI elem képes kezelni a rich textet. 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 logok 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.

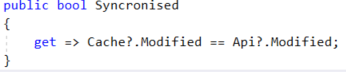


A Write-hoz több alternatíva is készült, melyekkel lehet közvetlenül logolni a konzolra.

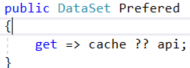
## 4.1.1. DataSet kezelés

## 4.1.1.1. DataSetInfo\_Model

Ennek az osztálynak a célja az, hogy könnyebben számon tartható legyen az adott DataSet állapota, azaz, hogy csak az Apin 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.

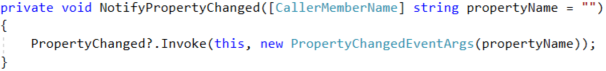


Ez a property akkor ad vissza igaz értéket, ha se a *Cache* se az *Api* nem null, illetve a Modified értéke megegyezik. Tehát ha tudjuk, hogy az Api elérhető, biztos, hogy van értelme letölteni az új adatokat.

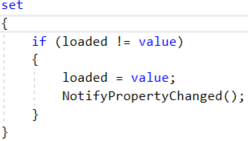


A Prefered property kényelemből lett hozzáadva, így olvashatóbb kódot lehet írni, mint ha minden alkalommal ellenőriznem kéne, hogy van-e gyorsítótárazott adat.

Ezek mellett az osztály még azt is tárolja, hogy be lett-e már töltve a Vufroia 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.

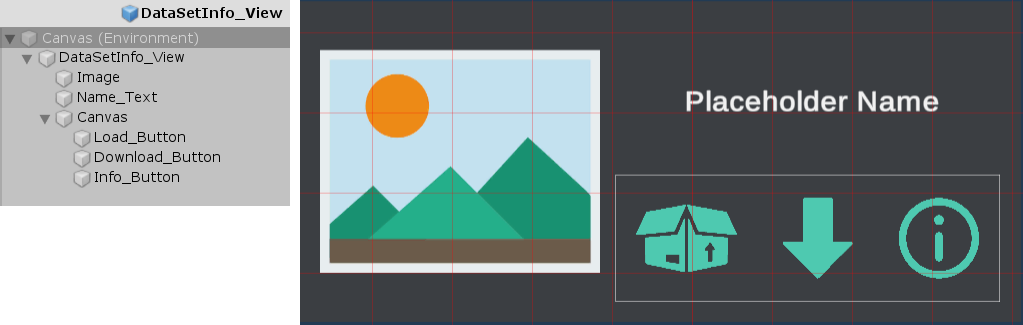


Az interfésznek arra szolgál, hogy értesítést tudjunk küldeni, ha a példány egy property-ének értéke megváltozik.

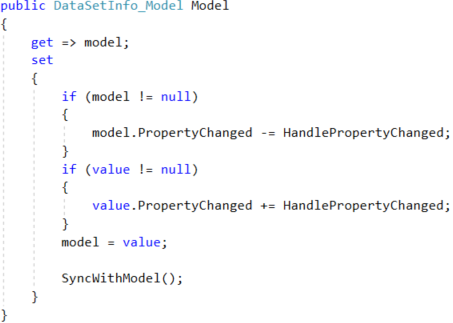


Azokat a property-ket, amikről értesítést szeretnénk küldeni úgy kell megírnunk, hogy a set részben megvizsgáljuk az új érték különbözik-e a régitől, majd, ha igen, akkor meghívjuk az interfészhez tartozó függvényt. A függvény egyetlen paramétere a hívó függvény neve, ami így a property neve lesz.

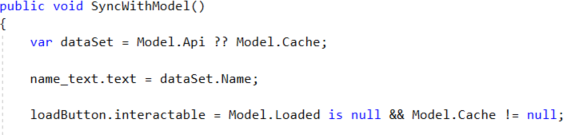
## 4.1.1.2. DataSetInfo\_View



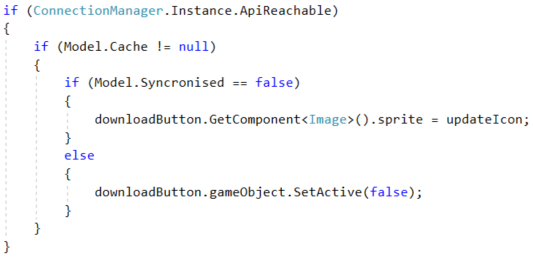
A DataSetInfo\_View a DataSet-ek megjelenítésért felelős komponens. Ahhoz, hogy megfelelően müködjön szükséges hozzá a képen látható hierarchia és elrendezés. Három gombot tartalmaz, a betöltés, a letöltés/frissítés és a részletes információk gombját. A felhasználói felület tartalmaz egy képet is ám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



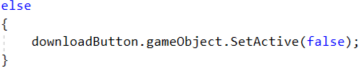
A View leköveti a Model értékeinek változásait. Ezt úgy értem el, hogy a *Model* property beállításakor feliratkozok a model PropertyChanged eseményére a HandlePropertyChanged függvénnyel. Ha a *Model* előzőleg nem null volt, akkor szükséges leiratkozni a korábbi érték eseményéről. Mivel legközelebb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

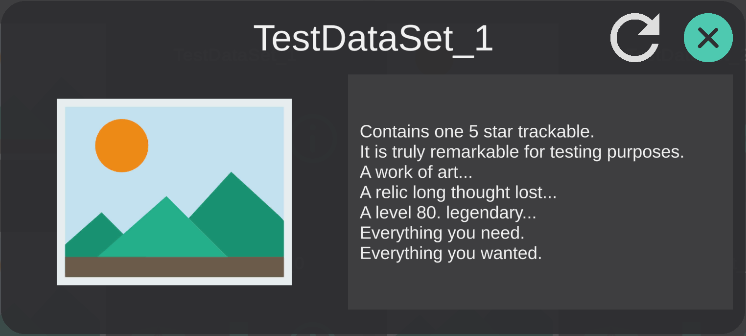


A *Loaded* esetében a hozzátartozó gomb akkor interaktálható, ha az null.



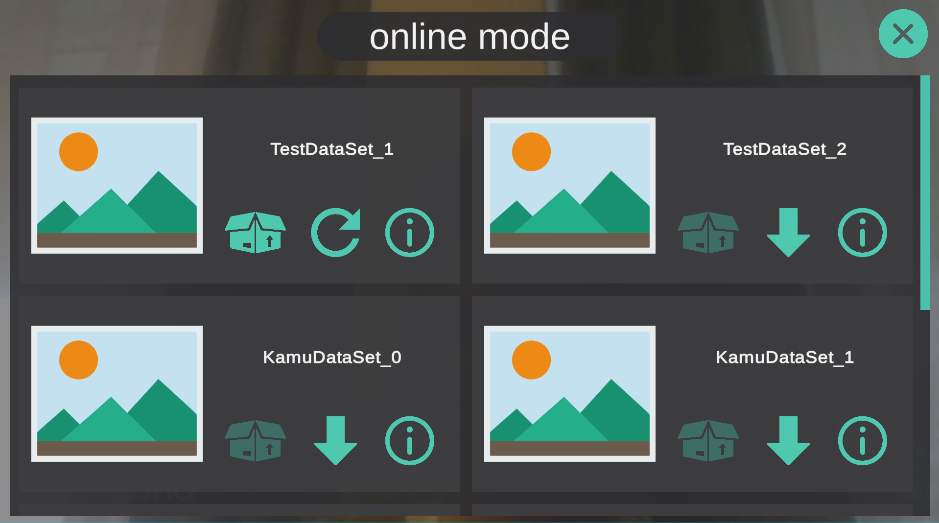
Ha a modell szinkronizálva van, akkor eltüntetem a letöltés gombot, mivel az már feleslegessé vált.

## 4.1.1.3. DataSetInfoDetails\_View



A DataSethez 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.

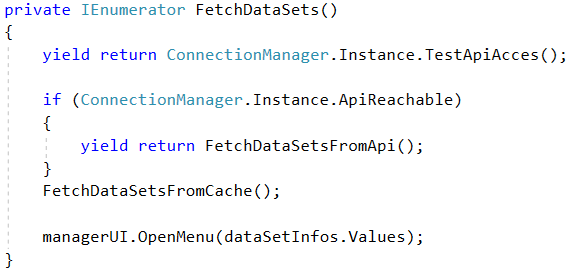
## 4.1.1.4. DataSetInfoManagerUI



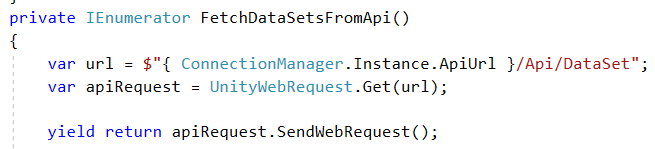
Az osztály DataSetInfo\_View-k kezelésére szolgál. Egy publikus függvénye van, ami megadott elemek egy listáját képes megjeleníteni. A függvény futásakor a listához tartozó canvast aktívvá teszem, ezzel megjelenítve a listát. Az api jelenlegi elérhetőségének megfelelően az „offline” vagy „online” felirat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

## 4.1.1.1. DataSetManager

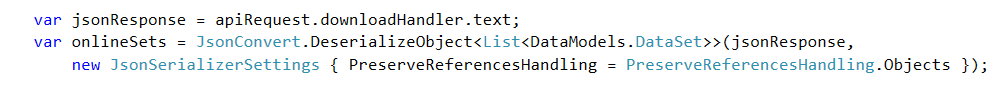
A DataSetManager osztály egy singleton, mely 2 fontos publikus függvényt biztosít. Mind kettő függvényt gombok segítségével hívom meg, most ezeket fogom bemutatni. Az első az OpenDataSetMenu, minek célja, hogy betöltse a rendelkezésre álló DataSeteket – legyen az online vagy offline –, majd egy másik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



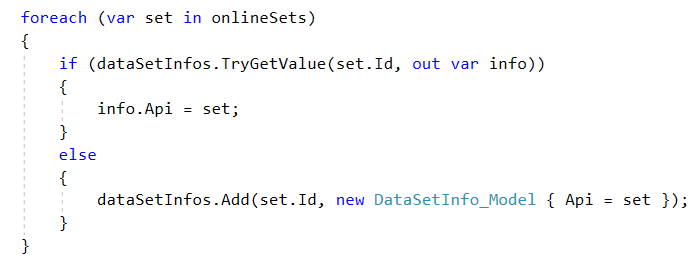
Az első lépés ellenőrizni az Api elérhetőségét. Ha elérhető, akkor lekérem az apin jelenlevő DataSetek – ezt lent részletesebben bemutatom – információit is. Ha nem tudtam elérni az Apit, akkor csak a gyorsítótárazott adatokat töltöm be. A betöltött adatokat ezután a felhasználó felületért felelős osztály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



Első lépésként beállítom az api híváshoz tartozó elérési utat. Ezt megadva létrehozok egy getet 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.

## 4.1.2. AssetBundle kezelés

## 4.1.3. Dll kezelés

## 4.1.4. ContentHandler