Frontend API hívások dokumentációja

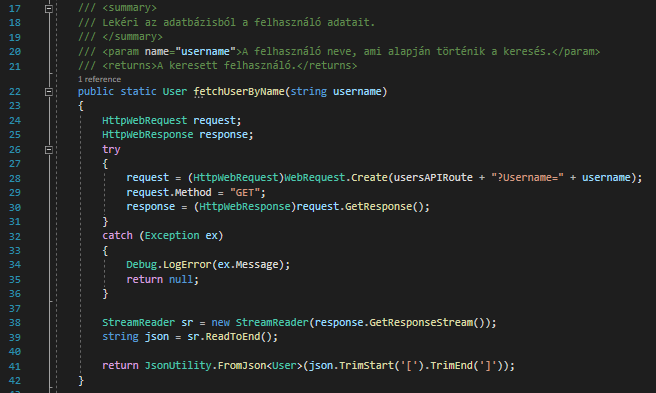
Fontos része a szoftver működésének, hogy a Unity-ban elkészített játék tudjon kommunikálni a szerverrel, hogy az adatok mentésre kerüljenek, illetve hogy a kliensek egymással tudjanak interaktálni. A szervernek a kliens kérésére ki kell szolgálnia azt adatokkal, illetve létre kell hozni, módosítani kell adatbázis rekordokat.

A frontenden a statikus APIHelper nevű osztály felelős ezért. Ezen osztály bármely másik szkriptből elérhető, publikus. Metódusaival szinkron és aszinkron API kéréseket tud küldeni a szerver felé.

Az, hogy szinkron vagy aszinkron kérést kell használnunk, attól függ, hogy a játéknak melyik fázisában tartunk. Nyilván a játék legelején, a menüben bejelentkezéshez nem szükséges aszinkron kéréseket használni, hiszen ha a bejelentkezés gomb megnyomására nem probléma, ha megakad a program, amíg megérkezik a válasz a szervertől. Amíg nincs bejelentkezve a felhasználó, addig a menüben úgysem tud mást csinálni.

Játék közben ez már nagyobb gondot okoz, hiszen a megfelelő játékélmény érdekében (pl. egy “ostrom indítása” gomb megnyomására) nem szabad várakoznia, megállnia a programnak. Így simább, minőségibb érzete lesz a szoftvernek.

Az APIHelper metódusai a System.Net és a Unity beépített UnityEngine.Networking osztályokat használják a kommunikáció megvalósítására. Ha adatot kér le az egyik metódus a szerverről (ez lehet például egy felhasználó, vagy egy város adatai, de akár ezekből több is egyszerre), akkor azt a Scripts/DBModels mappában található megfelelő modell formájában adja vissza (pl. egy felhasználó adatait a Scripts/DBModels/User modellben).

Vegyük például a fetchUserByName nevű szinkron függvényt. Küld egy kérést a szervernek, hogy küldje vissza a paraméterében megadott nevű felhasználó adatait. Először létrejönnek a ‘request’ és ‘response’ nevű változók, amikben később tárolódnak a kérés és az erre érkezett válasz adatai. Ez után egy try-catch blokk használatával elkészül a WebRequest osztály használatával a kérés. Ezen osztály Create metódusának megadjuk az URL-t, amire küldjük a kérést, majd implicit konverzió segítségével HttpRequest típusú kéréssé alakítjuk. Beállítjuk a kérés típusát ‘GET’ kérésre (hiszen adatokat kérünk), majd a GetResponse metódus segítségével elküldjük a kérést, és a visszaérkezett választ eltároljuk a ‘response’ változóban.

A try-catch blokk használata azért szükséges, mert a kérés elküldése során sok hibába ütközhetünk, de a leggyakoribb hiba az, hogy a kliensnek nincs megfelelő internetkapcsolata. Ilyenkor a ‘catch’ ágban a kapott hibaüzenetet a fejlesztőknek kiírjuk a Debug konzolra, és a függvény ‘null’ értékkel tér vissza, hiszen nem jött meg adat. Ha a függvény ‘null’ értékkel tér vissza, akkor arra következtethetünk, hogy vagy nincs ilyen nevű felhasználó, vagy a kapcsolat nem jött létre.

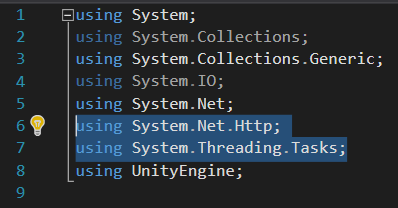
A metódus végén a StreamReader osztály használatával kiolvashatjuk a válasz tartalmát a GetResponseStream metódus segítségével. Nyilván az adatok JSON formátumban érkeznek meg, így ezt át kell alakítanunk a kívánt modellbe (ami jelen esetben egy User modell lesz). Ezt a JsonUtility osztály használatával a legegyszerűbb. A FromJson metódusnak megadjuk a kívánt típust, valamint paraméternek a ‘json’ formátumú adatokat. A json elejéről és végéről általában le kell szednünk a tömbzárójeleket (ha vannak benne), különben a metódus hibát adna.

Frontend API hívások dokumentációja

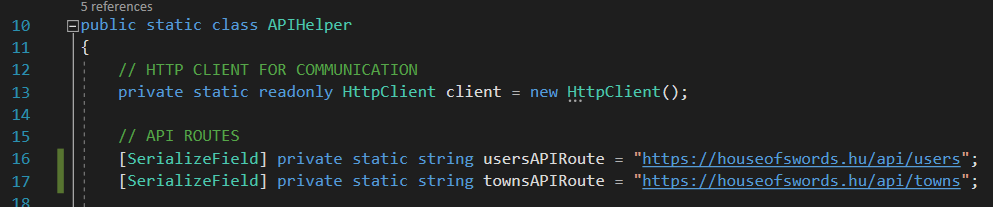
Fontos része a szoftver működésének, hogy a Unity-ban elkészített játék tudjon kommunikálni a szerverrel, hogy az adatok mentésre kerüljenek, illetve hogy a kliensek egymással tudjanak interaktálni. A szervernek a kliens kérésére ki kell szolgálnia azt adatokkal, illetve létre kell hozni, módosítani kell adatbázis rekordokat.

A frontenden a statikus APIHelper nevű osztály felelős ezért. Ezen osztály bármely másik szkriptből elérhető, publikus. Metódusaival aszinkron API kéréseket tud küldeni a szerver felé.

Fontos, hogy az összes metódusa aszinkron. Amíg a kérés elküldésre kerül, megérkezik a szerverhez, az adatok összegyűjtésre kerülnek, valamint visszaérkezik a válaszüzenet, programszinten nagyon sok időbe telik (átlagosan 4/10 másodperc). Ha a függvény szinkronban futna, az azt jelentené, hogy a program nem tud tovább haladni, amíg az le nem fut, vagyis eddig az ideig “lefagyna” a szoftver. Ha a metódus aszinkron, akkor a program nem áll meg, amíg válaszra vár.

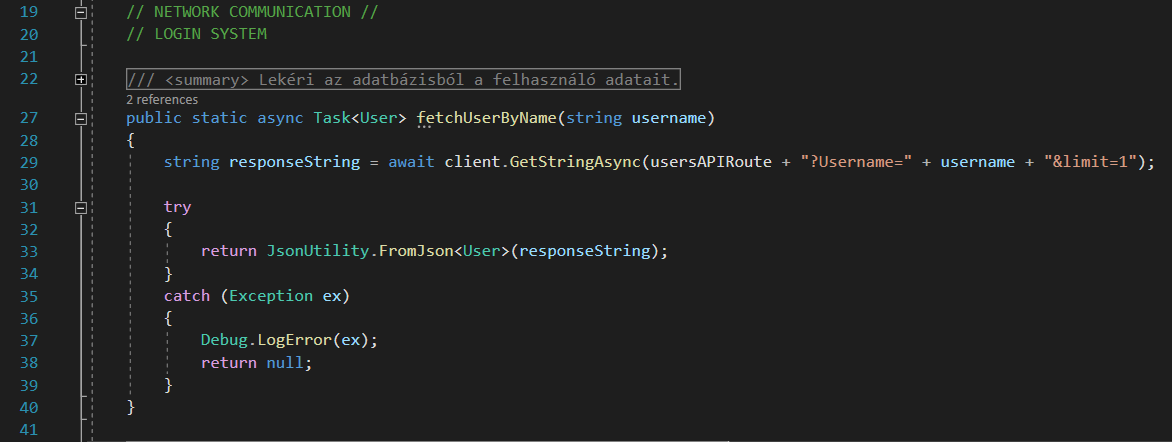
Az APIHelper metódusainak elkészítéséhez szükség van a System.Net.Http osztályra (a kérések kezeléséhez), és a System.Threading.Tasks osztályra (aszinkron függvények készítéséhez).

A kérésekhez a HttpClient osztályt használjuk. Ezzel az osztállyal nagyon egyszerűen, egy sorban tudunk egyszerű GET kéréseket küldeni a szervernek.



Az API végpontok is felvételre kerülnek statikus változókba, hogy könnyebben meg lehessen őket találni. A [SerializeField] címkére azért van szükség, hogy a Unity szerkesztőjében is látható lehessen egy privát változó.

Nézzünk egy példát egy egyszerű GET kérésre. Az alábbi kód lekéri az adatbázisból a paraméterben megadott nevű felhasználó rekordját a ’users’ táblából, és visszaadja azt egy User modell objektumként.



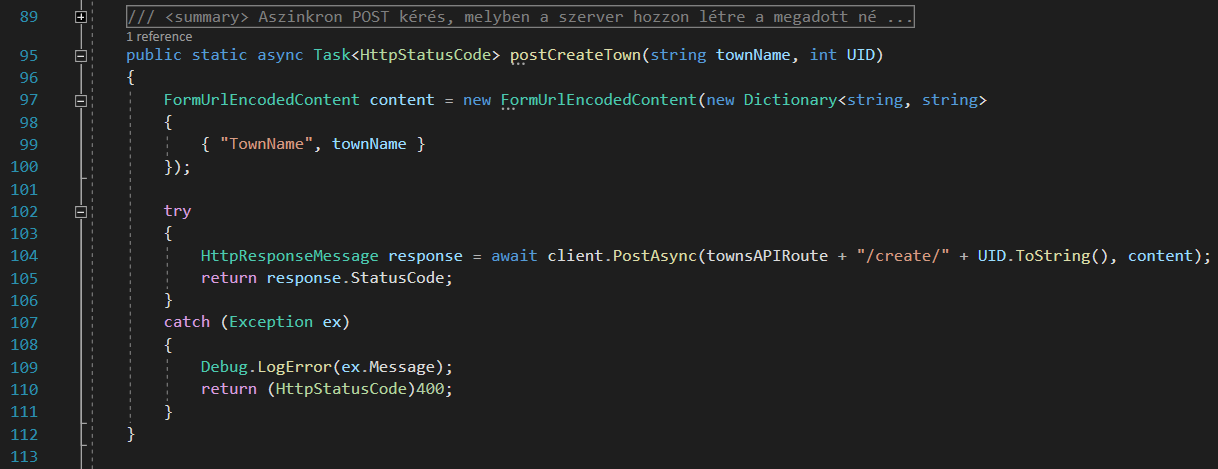
A kérés maga a 29. sorban történik, a fent létrehozott HttpClient osztály ’GetStringAsync’ metódusával. Ennek a függvénynek paraméterként megadjuk a kívánt API útvonalat, ez alapján elküldi a GET kérést, és a válaszban visszaérkezett JSON adatot szöveges változóban adja vissza, ami tárolásra kerül a ’responseString’ nevű változóban.

Az API útvonal úgy áll össze, hogy a tábla neve mögé ún. query paraméterekként megadjuk a kritériumokat. Jelen esetünkben két kritérium van: a felhasználó neve legyen egyező a paraméterként megkapott névvel (Username=username), és hogy egyetlen darab felhasználót szeretnénk kapni (limit=1; ez csak az anomáliák elkerülése érdekében szükséges, alapesetben csak egy darab felhasználót kapunk erre a kérésre, hiszen a felhasználónév egyedi).

Ezek után az érkezett felhasználói adatokat át kell alakítanunk JSON formátumból egy User osztályba. A try-catch blokk felel ezért. Ha sikeresen át tudjuk alakítani az érkezett adatokat a 33. sorban, akkor az eredményezett User modellt visszaadjuk a függvény eredményeképp.

Ha nem sikerül, akkor kiírjuk a Debug konzolra a hibát, és a függvény ’null’ értékkel tér vissza. Ha ez történik, akkor három dologra lehet következtetni: nem létezik ilyen nevű felhasználó (tehát nem jött adat a kérés hatására), nincs megfelelő internetkapcsolat, így a kérés el sem lett küldve, vagy a szerver nem elérhető.

Nézzünk egy példát egy POST kérésre, ahol mi küldünk adatot a szervernek további feldolgozásra. Az alábbi kód arra szolgál, hogy a megadott azonosítójú felhasználó új várost tudjon létrehozni.



Annyiban különbözik a GET kéréstől, hogy az adatokat meg kell adnunk, amiket el szeretnénk küldeni a szervernek. Erre a FormUrlEncodedContent osztályt használjuk.

Egy Dictionary objektummal példányosítjuk ezt az osztályt. Egy Dictionary szerkezete hasonló a JSON formátumhoz, minden kulcshoz tartozik egy érték, ebből tökéletesen létre lehet hozni a kérés adatait.

A 104. sorban a HttpClient ’PostAsync’ metódusával elküldjük a paraméterében megadott útvonalra a POST kérést, és második paraméternek megadjuk az adatokat. Válaszként kapunk egy HttpResponseMessage típusú objektumot. Ha sikeres a kérés (vagy nem a kliens-oldalon van a hiba), akkor visszaadjuk a válasz Http státusz kódját (sikeres kérés esetén ez 200 lesz).

A try-catch blokk a szintúgy a fentebb leírtak miatt szükséges, ha nem sikerül a kérés elküldése, akkor a függvény 400-as http hibakóddal tér vissza (Bad Request).