**Szegedi Tudományegyetem**

**Informatikai Intézet**

**SZAKDOLGOZAT**

**Mészáros Márton**

**2025**

**Szegedi Tudományegyetem**

**Informatikai Intézet**

**Interaktív weboldal fejlesztése falmászó közösség támogatására**

Szakdolgozat

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Készítette: |  | Témavezető: |  |
|  | **Mészáros Márton** |  | **Dr. Márkus András** |  |
|  | gazdaságinformatika BSc szakos hallgató |  | egyetemi adjunktus |  |

Szeged

2025

# Feladatkiírás

A szakdolgozat célja egy olyan weboldal létrehozása, amely a falmászás sportjára épül, és elsődlegesen egy Szegeden működő mászótermet, valamint az ott elérhető falmászó lehetőségeket mutatja be. Az oldal tartalmazza a terem részletes leírását, a megmászható utakat, valamint egy funkciót, amely lehetővé teszi a felhasználók számára, hogy saját utakat töltsenek fel. Ehhez egy beépített képszerkesztő is rendelkezésre áll, amely segít az utak vizuális megjelenítésében. Az egyes utakhoz hozzászólásokat lehet fűzni, ezzel véleményezni lehet őket, és a nehézségi szintjüket is módosítani lehet a közösségi visszajelzések alapján. Az oldalon egy beépített térképes megjelenítő is helyet kap, amely a terem helyét mutatja. Az új falmászó hely, fal, út és az utakhoz hozzászólni és a hozzászólásokra válaszolni kizárólag a regisztrált felhasználók számára elérhető, biztosítva ezzel a közösség szűrését. Három szerepkör van definiálva: vendég, regisztrált felhasználó és admin. Az admin jogosultsággal rendelkező felhasználó minden feltöltött adatot tud módosítani, valamint rendelkezik a regisztrált felhasználó összes jogosultságával.

# Tartalmi összefoglaló

* A téma megnevezése:

Interaktív weboldal fejlesztése falmászó közösségi támogatására

* A megadott feladat megfogalmazása:

Egy weboldal létrehozása, amelynek célja, hogy segítse a mászókat egyfajta közösségi oldalként, amellyel falmászó termeket és ottani falakat, valamint a falon lévő utakat tudják megosztani egymással.

* A megoldási mód:

A feladat megvalósítását egy modern, háromrétegű ASP.NET Core MVC alapú architektúra mentén végeztem, alkalmazva a Code First fejlesztési megközelítést. A perzisztencia réteget az Entity Framework Core (ORM) biztosítja egy SQLite adatbázis felett. A nagyméretű, vizuális adatok képek tárolására és gyors elérésére a Google Firebase Cloud Storage szolgáltatást használtam. A kliensoldali interaktivitást a Fabric.js JavaScript könyvtár beépítésével értem el, amely lehetővé teszi a mászóutak dinamikus, képalapú szerkesztését és a rajzolási állapotok kezelését. A felhasználói hitelesítést és a többszintű jogosultsági rendszert az ASP.NET Core Identity keretrendszerrel valósítottam meg

* Alkalmazott eszközök, módszerek:

Visual Studio 2022, ASP.NET Core, MVC, Razor View, C#, JavaScript, Entity Framework Core, SQLite, Google Firebase, Fabric.js

* Elért eredmények:

Létrehoztam egy működőképes, közösségi funkciókat támogató webalkalmazást, amely kezeli a fő entitások (falmászoHelyek, falak, utak) teljes CRUD műveletsorát. Kiemelt eredményként implementáltam egy Fabric.js alapú interaktív útvonalszerkesztőt (Undo/Redo funkcióval), valamint egy automatizált adatkezelési logikát, mely garantálja az adatbázis és a Firebase Cloud Storage konzisztenciáját (megakadályozva az árva fájlokat). A felhasználói adatok biztonságát a háromszintű szerepkör- alapú hozzáférés- szabályozás és a jelszavak biztonságos tárolása biztosítja.

* Kulcsszavak:

Falmászás, webalkalmazás, ASP.NET Core MVC, Fabric.js, Firebase Cloud Storage, adatintegritás

# Tartalomjegyzék

[Feladatkiírás 3](#_Toc215439057)

[Tartalmi összefoglaló 4](#_Toc215439058)

[Tartalomjegyzék 5](#_Toc215439059)

[1. Bevezetés 7](#_Toc215439060)

[1.1 Motiváció 7](#_Toc215439061)

[1.2. A konkurensek analízise és a funkcionális rés 8](#_Toc215439062)

[2. Technikai háttér 10](#_Toc215439063)

[2.1. A fejlesztői környezet Microsoft Visual Studio 2022 10](#_Toc215439064)

[2.2. Entity Framework Core (EF Core) 10](#_Toc215439065)

[2.3. Razor View 12](#_Toc215439066)

[2.4. Kliensoldali technológiák 12](#_Toc215439067)

[2.5. Fabric.js 13](#_Toc215439068)

[2.6. Firebase Cloud Storage 14](#_Toc215439069)

[2.7. Az alkalmazás architektúrája és rétegződése 15](#_Toc215439070)

[2.7.1. Az MVC architektúra működése és a függőségkezelés 16](#_Toc215439071)

[2.8. Az adatbázis-struktúra és kapcsolatkezelés 17](#_Toc215439072)

[2.8.1. Adatvalidáció és adatintegritás 18](#_Toc215439073)

[2.9. Verziókövetés és a fejlesztési metodológia 18](#_Toc215439074)

[2.10. Az alkalmazás futtatása 19](#_Toc215439075)

[2.11. Összegzés 19](#_Toc215439076)

[3. Az alkalmazás működése 20](#_Toc215439077)

[3.1. Felhasználói szerepkörök és jogosultságkezelés 20](#_Toc215439078)

[3.1.1. Jogosultságkezelés 20](#_Toc215439079)

[3.1.2. Felhasználó azonosítása 23](#_Toc215439080)

[3.2. Felhasználói műveletek 24](#_Toc215439081)

[3.2.1. Regisztráció 24](#_Toc215439082)

[3.2.2. Bejelentkezés 24](#_Toc215439083)

[3.2.3. Jelszó módosítás 25](#_Toc215439084)

[3.3. Adatbázis műveletek (CRUD) 25](#_Toc215439085)

[3.3.1. Az adatbázis felépítése 25](#_Toc215439086)

[3.3.2. Lekérdezési műveletek 27](#_Toc215439087)

[3.3.3. Létrehozási műveletek 28](#_Toc215439088)

[3.3.4. Módosítási műveletek 30](#_Toc215439089)

[3.3.5. Törlési műveletek 31](#_Toc215439090)

[3.4. Képkezelési és interaktív szerkesztési funkciók 33](#_Toc215439091)

[3.4.1 Interaktív szerkesztés és állapotkezelés 33](#_Toc215439092)

[3.4.2. Mászóutak képkezelése: szerkesztés és adatintegritás 35](#_Toc215439093)

[3.4.3. Adatintegritás és árva fájlok megelőzése (Törlés) 35](#_Toc215439094)

[3.5. Biztonsági mechanizmusok 36](#_Toc215439095)

[3.6. Külső szolgáltatások integrációja 38](#_Toc215439096)

[4. Összegzés 41](#_Toc215439097)

[Irodalomjegyzék 43](#_Toc215439098)

[Nyilatkozat 45](#_Toc215439099)

# 1. Bevezetés

## 1.1 Motiváció

A falmászással már egészen kis gyerekként megismerkedtem, de nem űztem mint sportot, csak nagyon ritkán több évente egyszer-egyszer foglalkoztam vele. Ezt követően egyetemi tanulmányaim alatt felvettem, mint kötelező testnevelési kurzust, ahol megszerettem, mint sportot és elkezdtem járni az esti edzésekre. Ott megismert barátoktól és saját tapasztalatból született meg az a gondolat a fejemben, hogy ezzel a témával kapcsolatban írjam meg a szakdolgozatomat. A weboldal egyik fő inspirálómondata az volt, hogy „milyen citromsárga nehézségű út volt ezen a falon legutóbb” mivel kéthetente újracetlizik az utakat azaz új utakat hoznak létre de az új út még nincs feljelölve, és nem emlékeznek arra, hogy hogy nézett ki a legutóbbi út. A teremben különböző színű ragasztószalaggal jelölik meg az adott útban használható fogásokat és lépéseket. Az utak újra rakásánál minden korábbi jelölést leszednek a falról, így nincs semmi fizikai nyoma, hogy melyik fogások voltak bejelölve a korábbi útnál. Ezen felül az is fontos volt, hogy ha van saját utad, akkor azt is meg tud adni úgy, hogy mások is lássák anélkül, hogy a meglévő utak cetlijeibe bezavarna (például abban az esetben, ha ugyan olyan nehézségű és nem lehetne elkülöníteni a két utat egymástól). Ez a spraywall típusú falakon hasznos, ahol az utak nem azonos színű fogásokból állnak. Másik számomra fontos rész, hogy ha egy másik mászótermet szeretnék meglátogatni, de a terem elérhetőségein nem találok olyan képet, ami az aktuális állapotot mutatja be, és érdekelne, hogy jelenleg hogy néznek ki a falak és milyen nehézségű utak, találhatóak ott. Ezen problémák megoldására és, hogy a falmászók megoszthassák egymással az információkat, hoztam létre a weboldalt, ami egyfajta online közösségi térként működik, és nem csak száraz adatokat tárol, hanem személyes véleményeket és tapasztalatokat is. A weboldalam célja, hogy egyfajta közösségi teret hozzon létre, hogy a mászók meg tudjanak osztani egymással személyes tapasztalatokat, véleményeket egy adott útról, vagy a meglévő fogásokból egy teljesen új utat hozzanak létre ezzel is színesítve a terem kínálatát.

## 1.2. A konkurensek analízise és a funkcionális rés

Mindkét weboldal célja, a sziklamászó és falmászó helyek listázása. A legfőbb különbség, hogy az általam fejlesztett weboldalon az utakhoz kell képet is társítani, míg a theCrag.com weboldalán ez nem feltétel csak egy opció. A „Csak mássz fel” weboldalon létezik az útnak leírása is, nem csak a nehézség és a név van megjelenítve, valamint az is látszik, hogy ki hozta létre. A theCrag.com weboldalon külön lapon, de csak az egyes falszakaszra van hozzászólási, fórum felület, míg nálam minden egyes útra van hozzászólási lehetőség. Külön falszakaszokra nincs hozzászólási lehetőség, mert a véleményt az útnál elég közölni, mert az esetek túlnyomó részében az úttal kapcsolatban fogalmaznak meg kritikát a mászók. A theCrag.com-on be lehet jelölni, ha már megmásztad az adott utat, míg az általam fejlesztett weboldalon ez a funkció nem érhető el. Ezen a weboldalon lehet a mászáshoz kapcsolódó könyveket is vásárolni, míg az általam létrehozott weboldal csak a falmászó helyeket és ezekhez kapcsolódó falakat, utakat és az utakhoz kapcsolódó hozzászólásokat jeleníti meg. A theCrag.com-on meg lehet tekinteni a felhasználókat is és néhány alap információt is megjelenít róluk. Nálam csak az admin jogosultsággal rendelkező felhasználó fér hozzá a felhasználók listájához. A bejelentkezetett felhasználó a „Csak mássz fel” weboldalon csak a jelszavát tudja módosítani, a profiljához nem fér hozzá, míg a theCrag.com weboldalon hozzáfér a felhasználó a profiljához és különböző statisztikákat is listákat is lát, mint például a kedvenc, megmászott utakat [1].

A theCrag.com weboldal nem csak a teremben található utakat jeleníti meg, hanem a szabadtéri mászásra alkalmas sziklafalakat is. Erre az általam fejlesztett weboldalam is alkalmas.

A theCrag.com részletes összehasonlítását követően elengedhetetlen a piacon elérhető további, mászókat támogató weboldalak rövid elemzése is. Számos online platform elérhető, amelyek közös jellemzője, hogy elsődlegesen sziklafalakat vagy mászótermeket listáznak, azonban funkcionalitásukban eltérések mutatkoznak a támogatott mászástipusok és a tartalom részletessége tekintetében.

A 27 Crags platform [2] kizárólag a szabadtéri sziklamászásra fókuszál mind bulder, trad és sport mászástípusokban egyaránt. Ezen a weboldalon az úthoz tartozik kép, nehézség típus, továbbá biztosított a hozzászólási lehetőség az utak véleményezésére.

A természetjáró [3] weboldal szélesebb körű célközönséget szólít meg (túrázók és sziklamászók), de a mászással kapcsolatos szekciójában szintén csak a szabadtéri sziklafalak vannak megjelenítve. Ez a platform hosszabb leírás keretén belül adja meg a szükséges alapinformációkat, és kizárólag a magyarországi sziklamászóhelyeről nyújt részletes magyar nyelvű leírást.

Ezen alkalmazások összehasonlító elemzése megerősített abban, hogy a jelenleg elérhető megoldások túlnyomórészt a sziklafalakat vagy a mászótermek statikus listázására, valamint az ezekhez tartozó alapinformációk megjelenítésére koncentrálnak. A vizsgált alkalmazások közül funkcionalitásban leginkább a Stökt [4] áll a legközelebb a „Csak mássz fel” weboldalhoz, ugyanakkor az elsősorban a spraywall típusú falak kezelésére fókuszál, míg az általam fejlesztett weboldal a nagyfalas utakra is hangsúlyt fektet. A piaci áttekintés eredményeként jól körvonalazódik a funkcionalitási rés, amelyet a „Csak mássz fel” weboldal az által biztosít egyedi értéket, hogy elsősorban a beltéri utak vizuális és interaktív szerkesztésére fókuszál, egy szorosan vett közösségi magot kiszolgálva, ami a konkurens rendszerekből hiányzik.

# 2. Technikai háttér

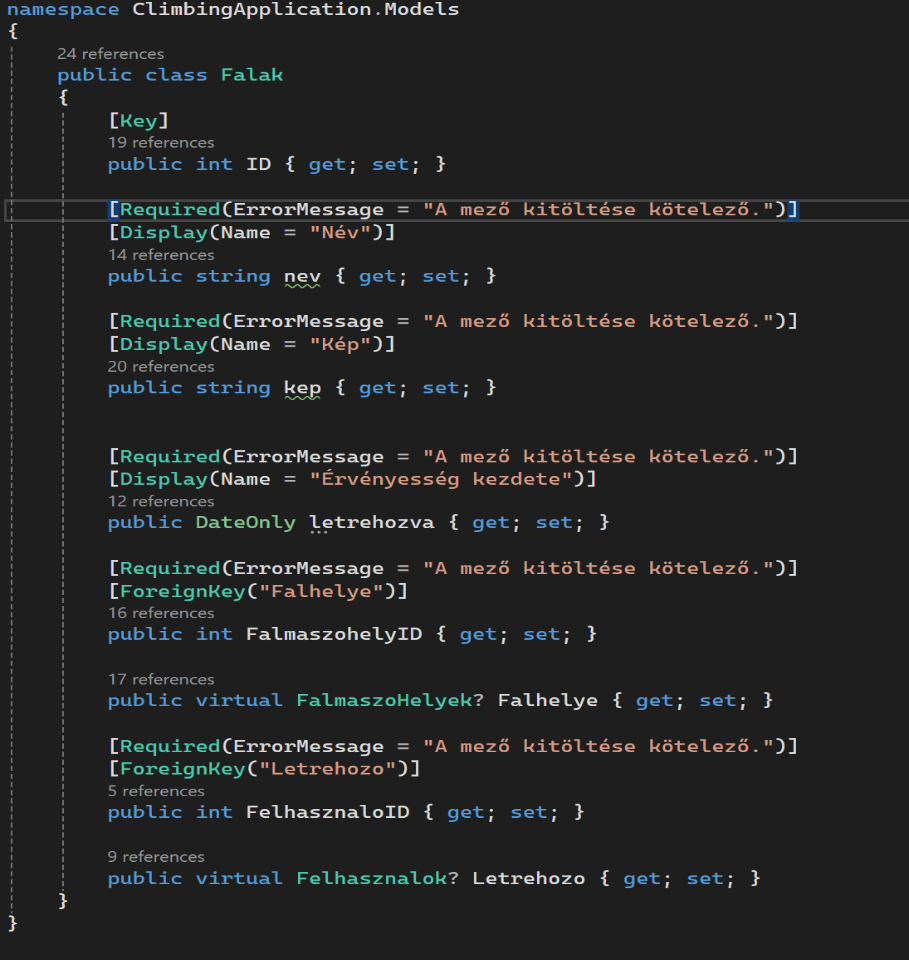
A projekt fejlesztése során egy modern, agilis, többrétegű architektúra és a Microsoft .NET Core ökoszisztémájára épülő technológiai stack került alkalmazásra. A választott eszközök és keretrendszerek, mint az ASP.NET Core, Entity Framework és a Fabric.js kiválasztásának célja a magas szintű produktivitás biztosítása volt. A Dependency Injection elv alkalmazása biztosítja a kód minőségét és karbantarthatóságát. A rendszer Code First megközelítéssel készült, amely lehetővé teszi a modellek alapján történő rugalmas adatbázis-fejlesztést. A következő fejezetek részletezik a fejlesztés során alkalmazott eszközöket, elveket és a futtatási környezetet.

## 2.1. A fejlesztői környezet Microsoft Visual Studio 2022

A fejlesztéshez a Microsoft Visual Studio 2022 fejlesztői környezet került kiválasztásra [5]. A döntés hátterében az áll, hogy a VS 2022 biztosítja a legtöbb, beépített a fejlesztést segítő funkciót az ASP.NET Core projekthez. Ez a környezet teljes mértékben támogatja a Code First megközelítést, amely lehetővé teszi a modellek alapján történő adatbázis-generálást. Ezen túlmenően a scaffolding funkció automatikusan létrehozza a hozzá tartozó Controller és View részeket is, jelentősen gyorsítva a munkafolyamatot. A Visual Studio 2022 olyan modern funkciókat is kínál, mint a Hot Reload, amely lehetővé teszi a kód változásainak valós idejű alkalmazását az alkalmazás újraindítása nélkül, ezáltal drasztikusan gyorsítva a hibakeresést és a fejlesztési ciklust.

## 2.2. Entity Framework Core (EF Core)

Az Entity Framework Core (EF Core) egy nyílt forráskódú objektum-relációs leképező (ORM) keretrendszer a .NET platformra írt alkalmazások számára. A keretrendszer használatával a fejlesztők nagyobb termelékenyéget érnek el, mivel elkerülik a külön adatbázisspecifikus kódok írását. Az EF Core-t használók a tartományspecifikus osztályokra fókuszálnak, nem pedig az adatbázisban szereplő oszlopokra. Az EF Core használatával az adatbázis-kapcsolat kódok létrehozása feleslegessé válik, mivel az EF Core absztrakt szinten biztosítja azokat a fejlesztők számára. Nagyban csökkenti az adatspecifikus kódok méretét, és növeli a kód olvashatóságát is. Az Entity Framework végrehajtja a szükséges lekérdezéseket az adatbázisban, majd az eredményeket a domén modell példányaivá alakítja, így az alkalmazás feldolgozhatja azokat. Az EF Core néhány fontos jellemzője a platformfüggetlen LINQ lekérdezések használata SQL lekérdezések helyett, és ezeknek az alkalmazása az adatmanipulációra is. Az EF Core a 2.1.1–es ábrán látható modell osztályban lévő kódot veszi alapul az adatbázis adott táblájának létrehozására. A Code First megközelítés lehetővé teszi a kulcs és külső kulcshivatkozások, valamint a mezők kötelező és opcionális értékeinek beállítását közvetlenül a modellosztályokban [6].

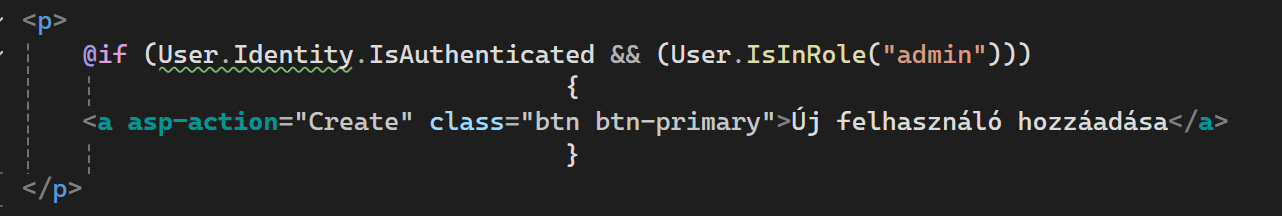


Ábra 2.1.1.Modell osztály példakód saját alkalmazásból

## 2.3. Razor View

A Razor egy jelölőszintaxis, amely az ASP.NET Core keretrendszerben a C# kód dinamikus beágyazására szolgál a weboldalakba. Ezt a technológiát a Microsoft fejlesztette ki, amely lehetővé teszi a C# és a HTML elemek zökkenőmentes kombinációját, ami a modern webalkalmazásokban a felhasználói felületeinek létrehozásának alapvető eszközévé válik. A Razor szintaxis leegyszerűsíti a szerveroldali logika és a kliensoldali megjelenítés elkülönítését, ezáltal növelve a kód olvashatóságát és karbantarthatóságát. Az ilyen nézetfájlok .cshtml kiterjesztést kapnak [7].

A @ karakter egy kulcsfontosságú karakter, ez jelzi a Razor számára, hogy ettől a karaktertől kezdve C# kódra kell váltani. Ez teszi lehetővé C# kifejezések, változók és utasítások beágyazását közvetlenül a HTML kódba. Implicit kifejezések esetén a @ szimbólum utáni C# kódot a Razor kiértékeli és HTML kimenetként jeleníti meg. Explicit kifejezések a @ utáni és kapcsos zárójelek közé írt C# kódot tartalmazzák ahogy a 2.3.1-es ábrán is látható, amely lehetővé teszi összetettebb műveletek beágyazását. A Razor teljes kódblokkot, például ciklusokat és feltételes utasításokat is értelmez. A Razor alapértelmezés szerint automatikusan kódolja a HTML és a C# kifejezések eredményét, ezzel kiküszöbölve a cross-site scripting (XSS) támadásokat. Az IHtmlContent típusú adatokat azonban kódolás nélkül jelennek meg, amely lehetővé teszi a szándékosan formázott HTML-tartalom beillesztését [8].



Ábra 2.3.1. Példa a Razora Ami egy feltételes utasítást tartalmaz nézet oldalon, hogy ki mit lásson

## 2.4. Kliensoldali technológiák

A modern webalkalmazások felhasználói felületei (UI) gyakran építenek a szerveroldali nézetmotorokon (mint Razor) túlmutató, fejlett kliensoldali technológiákra. A reszponzív megjelenés és a magas szintű interaktív felhasználói élmény biztosítása kiemelt cél. A felületek egységes kialakítását és a különböző eszközökön (például mobiltelefonokon és asztali gépeken) való megfelelő működést gyakran Bootstrap-hez hasonló CSS keretrendszerek biztosítják, a projectemben a reszponzív nézetekért és a minimalista dizájnért a Bootstrap és a CSS felel, mivel már a VS 2022 által generált weblapok ezeket alkalmazzák [9].

Az interaktív és dinamikus grafikus funkciók megvalósításához, különösen a canvas alapú alkalmazások esetében, speciális JavaScript könyvtárak alkalmazása szükséges, például a Fabric.js-t alkalmazzák. Ebben a projektben a Fabric.js könyvtár magas szintű objektumorientált API-t biztosít a grafikus elemek manipulálására [10]. Az általános viselkedésért, a DOM manipulációért és a dinamikus tartalomkezelésért a natív JavaScript felel, amely a webfejlesztés alapvető programozási nyelve.

## 2.5. Fabric.js

A Fabric.js egy nyílt forráskódú JavaScript könyvtár, amely interaktív objektummodellt, biztosít a HTML5 canvas elem leegyszerűsítésére. A könyvtár magasabb szintű absztrakcióval kezeli a formákat, képeket és szövegeket, ezzel megkönnyíti az interaktív grafikus alkalmazások fejlesztését a böngészőkben. A fejlesztők ennek a segítségével nem közvetlenül az alacsonyabb szintű canvas API hívással dolgoznak, hanem egy jól meghatározott, tulajdonságokkal és metódusokkal rendelkező objektumokkal [10].

A könyvtár objektumorientált megközelítést alkalmaz minden grafikus elemet – például geometriai alakzat, kép vagy szöveg – önálló objektumként kezel. Ezen objektumok olyan attribútumokkal rendelkeznek, mint a pozíció, méret, szín, átlátszóság vagy a forgási szög. Az objektumok csoportosíthatók, így több elem együttesen is kezelhető, amely különösen hasznos a komplex grafikus jelenetek szerkesztésekor [10].

A Fabric.js egyik legfontosabb ismérve az interaktív eseménykezelés. A könyvtár beépített eseménykezelővel lehetővé teszi a felhasználói műveletek (például egérkattintás, húzás vagy billentyűleütés) közvetlen kapcsolódását a grafikus elemekhez. Ezáltal a felhasználók intuitív módon mozgathatják, forgathatják vagy méretezhetik az objektumokat a vásznon, amely dinamikus és reszponzív felhasználói élményt biztosít [10].

A könyvtár nyílt forráskódú jellege, moduláris felépítése és aktív fejlesztői közössége révén stabil, jól dokumentált és rugalmas alapot, biztosít a modern, interaktív webes grafikai alkalmazások fejlesztéséhez [10].

## 2.6. Firebase Cloud Storage

A Google Cloud Storage for Firebase a Firebase platformon belül, a Google Cloud-alapú objektumtárolási szolgáltatás, amely kifejezetten a felhasználók által létrehozott tartalmak például képek, videók vagy dokumentumok biztonságos feltöltésére, tárolására és elérésére szolgál [11].

A rendszer a Google Cloud Storage infrastruktúrájára épül, ezáltal rendkívül jól skálázható, magas rendelkezésreállást biztosít, és képes petabájtos nagyságrendű adatok kezelésére is [12]. A fejlesztők a Firebase SDK-kon keresztül valósítják meg a fájlok feltöltését, letöltését vagy törlését anélkül, hogy közvetlenül a háttérrendszer kezelésével kellene foglalkozniuk [13].

A biztonság a szolgáltatás egyik legfontosabb eleme: a Firebase Authentication integráció révén a hozzáférések szabályozása felhasználói szinten is megvalósítható. A Firebase Security Rules segítségével pontosan meghatározható, hogy mely felhasználók milyen fájlokat érhetnek el, és milyen műveleteket hajthatnak végre rajtuk [14].

Az egyik legfőbb jellemzője az automatikus skálázhatóság: A Cloud Storage a terhelés növekedésével automatikusan alkalmazkodik, így nincs szükség manuális beavatkozásra [11].

Egy másik jellemző a biztonság és titkosítás. Az adatok titkosítva vannak tárolva és továbbítva, a hozzáféréseket pedig szabályok kontrollálják [15].

További fontos jellemzője az egyszerű integráció, azaz könnyedén kapcsolódik más Firebase-szolgáltatásokhoz például Firestore-hoz, valós idejű adatbázishoz vagy Authentication-hoz [16].

A fejlesztők úgynevezett bucket struktúrán belül dolgoznak, ahol minden fájl metaadatokkal együtt tárolható. A kliensoldali SDK támogatja a megszakított feltöltések folytatását, így a rendszer gyenge hálózati kapcsolatok mellett is megbízhatóan működik [13].

A Cloud Storage for Firebase ideális választás olyan web- és mobilalkalmazások számára, melyek nagy mennyiségű multimédiás vagy felhasználói tartalommal dolgoznak, például közösségi média-, oktatási vagy képszerkesztő rendszer esetében. A szolgáltatás rugalmassága, biztonsága és méretezhetősége révén hatékony háttértárat biztosít modern felhőalapú fejlesztésekhez.

## 2.7. Az alkalmazás architektúrája és rétegződése

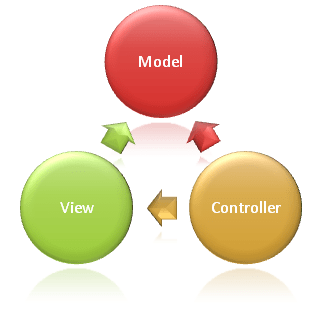
A modern webalkalmazások fejlesztésében alapvető elv, a többrétegű architektúra alkalmazása. Ez a megközelítés funkcionálisan elkülöníti az egyes rétegeket, ezzel elősegítve a kód újrafelhasználhatóságát, a karbantarthatóságot és a tesztelhetőséget.

Az általánosan alkalmazott megközelítés az MVC (Model-View-Controller), amely három fő részre tagolja az alkalmazást: Prezentációs (View) réteg: ez a réteg biztosítja a felhasználói interakciót és az adatok megjelenítését. A megvalósítás leggyakrabban HTML, CSS, JavaScript és különböző keretrendszerek (például Bootstrap, React, vagy Angular) használatával történik.

A következő réteg az üzleti (Controller) réteg, ez dolgozza fel a felhasználói kéréseket, koordinálja az adatok áramlását a View és a Model rétegek között.

A harmadik réteg az adatkezelési (Model) réteg, ez a réteg az alkalmazás aktuális állapotának kezeléséért, valamint az ehhez kapcsolódó üzleti logikáért és műveletekért felel. Ebbe a rétegbe építjük be az üzleti szabályokat és a perzisztencia logikát (az adatok adatbázisban való tárolásának módját) egyaránt ebben a rétegbe kell beágyazni [17].

A rétegzett architektúra alkalmazása lehetővé teszi a különböző rétegek egymástól független fejlődését [17].



Ábra 2.7.1. MVC ábra [16].

### 2.7.1. Az MVC architektúra működése és a függőségkezelés

Az MVC architektúra célja a felelősségek szétválasztása (Separation of Concerns, SoC). A vezérlő (Controller) nem közvetlenül módosítja az adatbázist vagy a nézetet, hanem közvetítőként funkcionál. Ez a laza csatolás (loose coupling) elve. A modern ASP.NET Core alkalmazásokban ezt a függőségi injektálás (Dependency Injection, DI) elvével valósítják meg. Ez azt jelenti, hogy a magasabb szintű komponensek, mint például a Controller nem közvetlenül hozzák létre az alacsonyabb szintű komponenseket, hanem egy interfészen keresztül kérik be azokat. Ez a gyakorlat jelentősen megkönnyíti a tesztelést, mivel a valós adatbázis-hozzáférés helyettesíthető tesztadatokat szolgáltató hamis objektumokkal (úgynevezett mock objectekkel) [18].

A Dependency Injection (DI) egy tervezési minta, amely megkönnyíti az objektumok közötti függőségek kezelését oly módon, hogy lazán kapcsolja össze a komponenseket. A DI ötlete az, hogy egy osztály nem közvetlenül hozza létre az általa használt függőségeket, hanem egy másik, kívülálló egység – az IoC (Inversion of Control) konténer – biztosítja azokat a példányosítás alkalmával. Az ASP.NET Core beépített DI rendszerrel rendelkezik, amely automatikusan kezeli a szolgáltatások életciklusát, ezzel megkönnyítve a tesztelést, az újrafelhasználhatóságot és a moduláris fejlesztést [18].

Az alkalmazás indításakor a regisztrációkat, mint például az adatbáziskuntextust, a feltöltési logikát vagy a hitelesítési szolgáltatásokat a Program.cs-ben adjuk meg. A Controllerek és más osztályok a konstruktorukon keresztül kapják meg ezeket a függőségeket, így nem szükséges az objektumok létrehozásáról gondoskodnunk. Ez a megközelítés egy átláthatóbb, könnyebben karbantartható és bővíthető architektúrát eredményez.

## 2.8. Az adatbázis-struktúra és kapcsolatkezelés

A komplex alkalmazások a domén modell és az adatbázis-hozzáférés részleteinek szétválasztása érdekében az absztrakciós rétegeket alkalmaznak, melyek célja a kódduplikáció minimalizálása és a karbantarthatóság növelése [19].

Az Entity Framework Core használata során az alkalmazásban a Code First megközelítés egyre elterjedtebb. Ez a paradigma azt jelenti, hogy a C# nyelvű osztályok (entitások) jelentik az elődleges forrást a perzisztencia számára, és ezek alapján generálódik le az adatbázis sémája. Ez a módszer jellemzően rugalmasabb, mint a Database First megközelítés, ahol a fejlesztő már egy létező adatbázisból generálja le a kódokat [20]. A Code First szorosan illeszkedik a modern objektumorientált tervezési elvekhez, és jobban támogatja a folyamatos reflektálást és a verziókövetést a migrációs mechanizmusokon keresztül. A fejlesztők a kódban, C# nyelven definiálják a kapcsolatokat és megkötéseket (például [ForeignKey] attribútumokkal vagy a Fluent API-val), melyek az adatbázis szintjén is érvényesülnek [21].

A szoftverfejlesztési életciklus során az alkalmazások adatmodelljei folyamatosan fejlődnek, ami szükségessé teszi a mögöttes adatbázissémák frissítését. Az Entity Framework (EF) Core keretrendszer Migráció (Migration) funkciója kulcsfontosságú eszközt biztosít ezen inkrementális sémafrissítések kezelésére anélkül, hogy a meglévő adatok sérülnének. A folyamat során a fejlesztő az EF Core parancssori eszközeivel generálja le a C# kódot, amely a bekövetkezett modellváltozásokat írja le (migrációs forrásfájlok). Ezek a fájlok a verziókövető rendszer részét képzik. Az adatbázis frissítésekor az EF Core egy belső előzménytáblája alapján határozza meg, mely változtatásokat kell még végrehajtani a séma és az adatmodell szinkronizálása érdekében. A migrációs mechanizmus biztosítja, hogy az adatbázisséma változásai verziókövethetőek és bármikor visszavonhatóak legyenek. Ez a funkció különösen fontossá válik nagyobb projektek vagy csapatmunkában végzett fejlesztések során, amikor az adatszerkezetek dinamikusan módosulnak [22].

Az olyan könnyűsúlyú, fájlalapú adatbázis-megoldások, mint az SQLite, fejlesztési és tesztkörnyezetben különösen előnyösek, mivel nem igényelnek dedikált szervert, gyorsan beállíthatók, és platformfüggetlenül működnek [23]. Az alkalmazott ORM-rétegnek köszönhetően a későbbiekben akár más, robusztusabb adatbázis-motor is alkalmazható (például Microsoft SQL vagy PostgreSQL) anélkül, hogy az alkalmazás felsőbb rétegeit jelentősen módosítani kellene [24].

### 2.8.1. Adatvalidáció és adatintegritás

Az alkalmazás megbízhatóságának biztosítása érdekében a bemeneti adatok érvényesítése (validációja) kritikus fontosságú. Az ASP.NET Core MVC keretrendszer beépített mechanizmust kínál a System.ComponentModel.DataAnnotations névtérből származó attribútumok ([Required], [StringLength], [Range]) segítségével. Ez a deklaratív megközelítés lehetővé teszi, hogy a validációs szabályokat közvetlenül a C# entitásosztályok tulajdonságaihoz illesszük. A keretrendszer a validációt szerveroldalon automatikusan érvényesíti a ModelState.IsValid állapot ellenőrzésekor, megakadályozva a hibás adatok adatbázisba való kerülését, ezzel is biztosítva az adatok integritását [25].

## 2.9. Verziókövetés és a fejlesztési metodológia

A szoftverfejlesztés egyik alapköve a verziókövetés, ami lehetővé teszi a fejlesztők számára, hogy a forráskód különböző állapotait nyomon kövessék, és segítse az együttműködést több fejlesztő között, valamint a korábbi verziókhoz való visszatérést. A legszélesebb körben elterjedt eszköz a Git, míg távoli repozitóriumként gyakori a GitHub szolgáltatás alkalmazása [26].

## 2.10. Az alkalmazás futtatása

A szoftverfejlesztési életciklus során a fejlesztési és a tesztelési fázisban az alkalmazás helyi futtatása a Microsoft Visual Studio beépített Kestrel webkiszolgálóján keresztül történt. A Kestrel az ASP.NET Core alapértelmezett, keresztplatformos webkiszolgálója, melyet a fejlesztői környezet indít el a helyi gépen a gyors iteráció és hibakeresés érdekében [27].

A helyi környezet a localhost címen, egy előre meghatározott porton keresztül biztosítja a szolgáltatás elérését. Mivel a szoftver .NET Core alapokon épül, a végleges telepítés (deployment) lehetőségei rendkívül szélesek. Megvalósítható Docker környezetben, Azure App Service-en, vagy bármely más felhőszolgáltatónál (például AWS, Google Cloud), ami támogatja a .NET Runtime-ot, ezzel biztosítva a platformfüggetlen éles üzemet [28].

## 2.11. Összegzés

Összefoglalva, a projekt egy tudatosan megválasztott technológiai stackre épül, amely a Microsoft .NET Core rugalmasságát ötvözi a modern frontend megoldások interaktivitásával. A Code First és az Entity Framework használata gyors adatmodell-fejlesztést tesz lehetővé, míg a lazán csatolt, MVC alapú architektúra a Dependency Injection elvén keresztül biztosítja a könnyű tesztelhetőséget és a kód hosszú távú karbantarthatóságát. A választott technológiák összessége egy robosztus, skálázható és jól karbantartható webalkalmazás létrehozását szolgálja.

# 3. Az alkalmazás működése

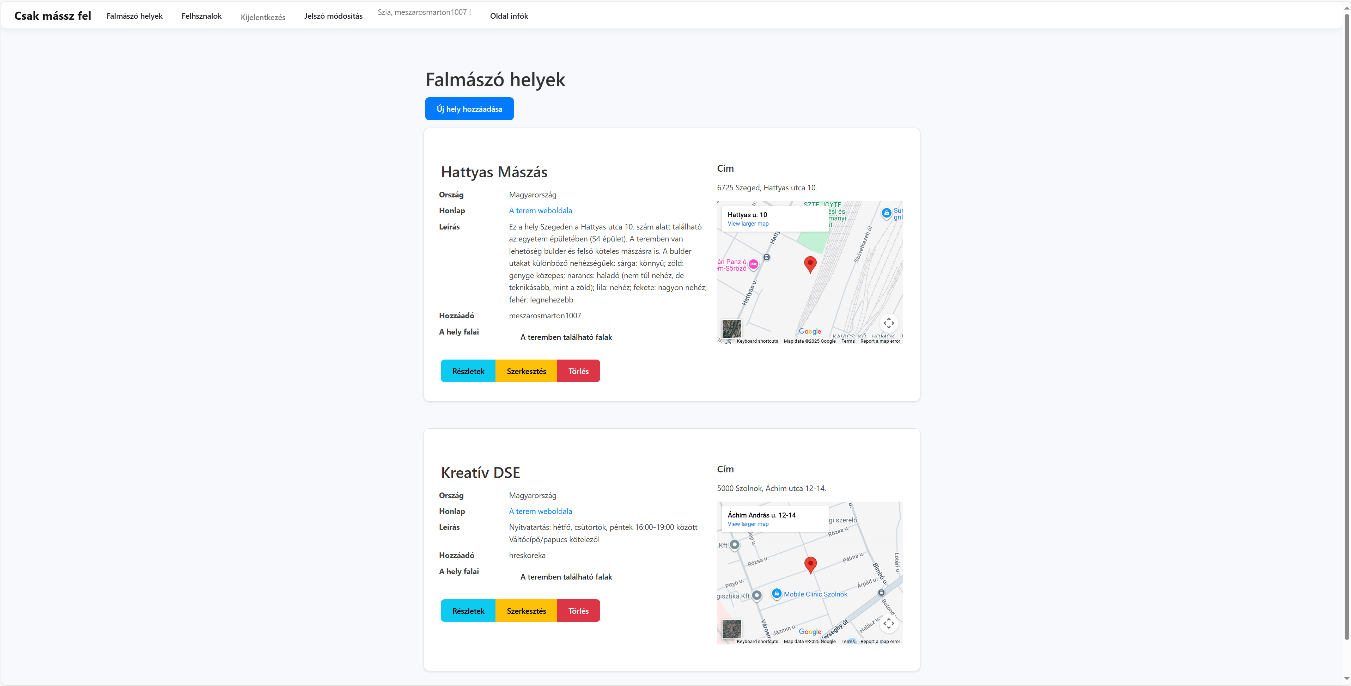
Ez a fejezet a Csak mássz fel névre keresztelt weboldal technikai működését mutatja be funkcionális szempontól, részletesen ismertetve az azalkalmazás architektúráját, az adatok kezelési folyamatát és a felhasználói interakciók technikai megvalósítását. A rendszer alapját az ASP.NET Core MVC keretrendszer adja, amely lehetővé teszi a logikai rétegek – a megjelenítés, az üzleti logika és az adatkezelés – világos szétválasztását. Az adatok perzisztenciája SQLite adatbázisban történik, az Entity Framework Core ORM technológiájának segítségével, amely biztosítja az egyszerű és biztonságos adatkezelést. A felhasználók által feltöltött képek tárolására a webalkalmazás a Firebase Cloud Storage szolgáltatást használja. A fájlokat biztonságosan helyezem el a felhőben, míg az elérési útvonalakat és a metaadatokat azadatbázisban tárolom. A felhasználói felületet a Razor nézetmotor, a Bootstrap és a JavaScript technológiák együttműködése biztosítja, ezzel a weboldal reszponzív, vizuálisan egységes és könnyen kezelhető marad. A dinamikus, interaktív elemek különösen a képszerkesztési funkciók megvalósításához a Fabric.js JavaScript könyvtár nyújt támogatást, amely lehetővé teszi a böngészőben, valós idejű rajzolást a képeken a canvas vásznon keresztül, valamint az undo és redo műveletek implementálását a vászonállapot mentésével.

## 3.1. Felhasználói szerepkörök és jogosultságkezelés

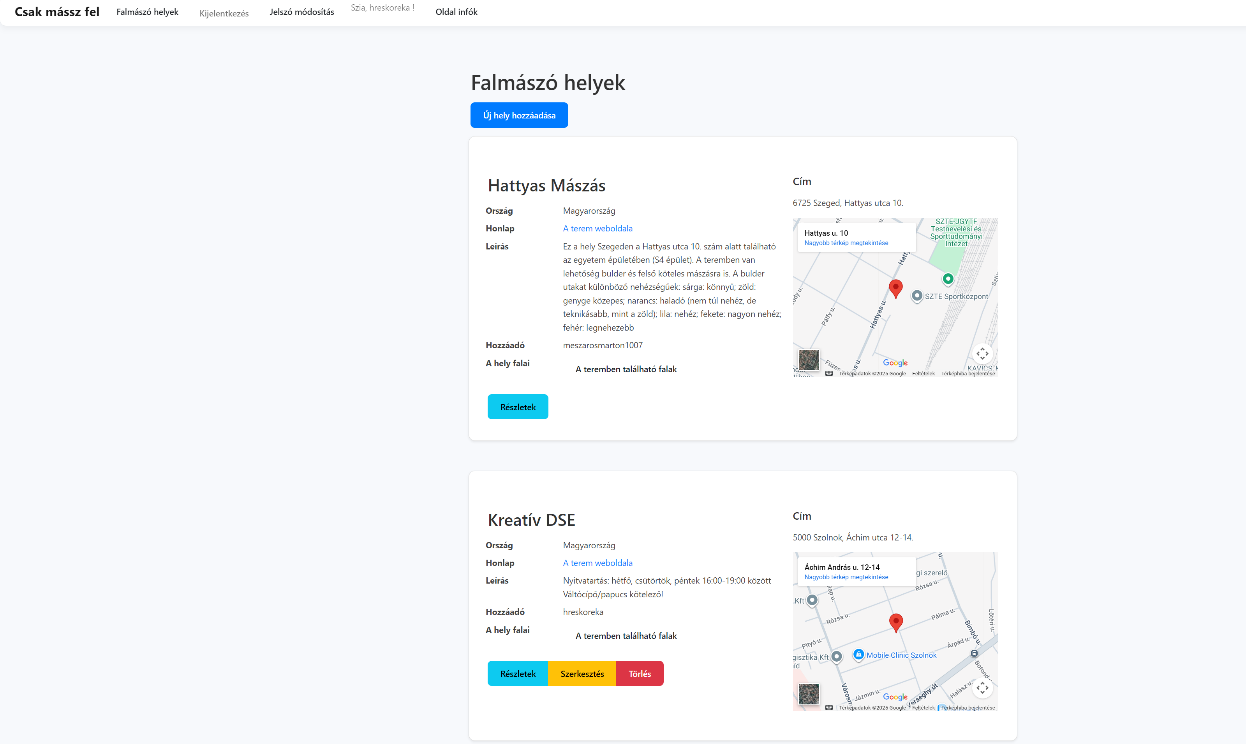
### 3.1.1. Jogosultságkezelés

Az alkalmazás három fő felhasználói szerepkört különböztet meg: az adminisztrátort, a regisztrált felhasználót és a vendég felhasználót. Mindhárom szerepkör különböző jogosultságokkal rendelkezik, így eltérő funkciókhoz fér hozzá az alkalmazáson belül.

Az adminisztrátor a legmagasabb jogosultsági szinttel rendelkezik, ezáltal teljes hozzáférést biztosít az alkalmazás minden eleméhez. Feladatai közé tartozik az új falmászóhelyek, falak és utak hozzáadása, a feltöltött utakhoz való hozzászólás és a hozzászólásokra adott válaszok, valamint mindezek szerkesztése és törlése. Emellett módosíthatja a már regisztrált felhasználók adatait és a felhasználók által feltöltött összes adatot. Így biztosítja a moderálást, megelőzve, hogy nem valós vagy sértő információ túl sokáig jelen legyen.

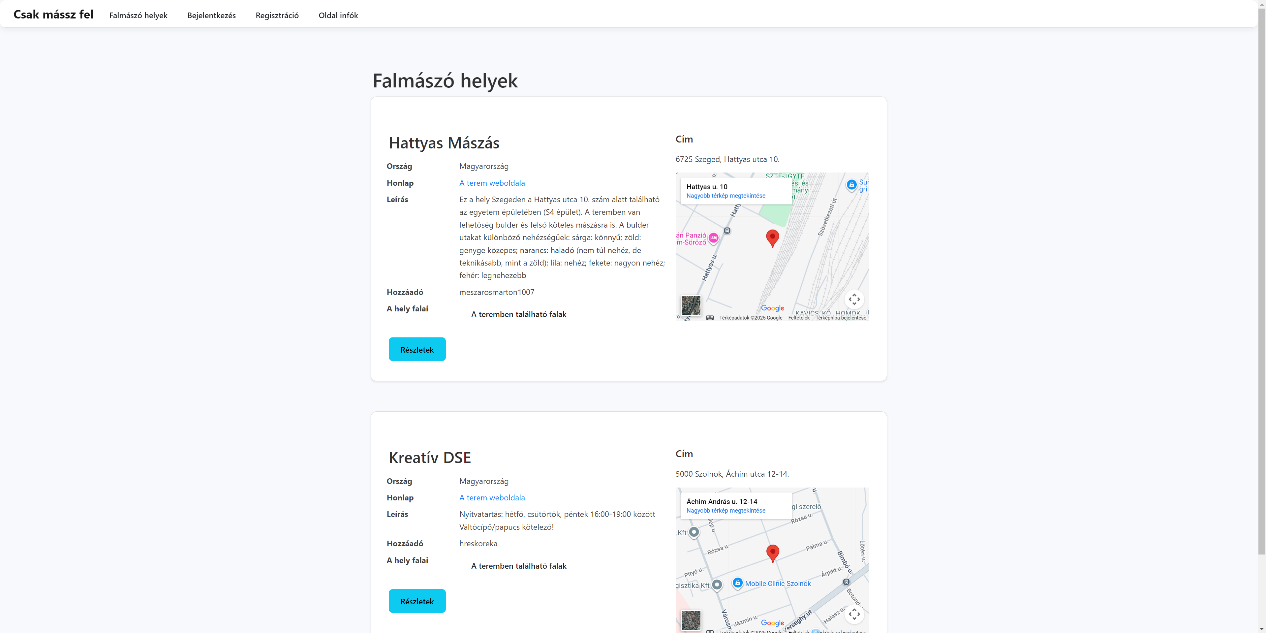


Ábra 3.1.1.1 Adminisztrátori nézet. Más által feltöltött adatot is tud szerkeszteni



Ábra 3.1.1.2 Regisztrált felhasználó nézet példa. Csak azt tudja módosítani, amit ő töltött fel

A regisztrált felhasználó rendelkezik saját profillal, amelyben saját adatai – felhasználónév, email, név, telefonszám és rang – nem módosítható, ezeket csak az adminisztrátor tudja módosítani. A regisztrált felhasználó a személyes adatai közül kizárólag a jelszavát tudja módosítani. A regisztrált felhasználó ugyanúgy, mint az adminisztrátor tud új falmászóhelyeket, falakat és utakat feltölteni. A feltöltött utakhoz hozzászólni és a hozzászólásokra válaszolni is képes. Az általa feltöltött adatokat tudja módosítani és törölni. Mások által feltöltött adatokat (például út, fal) nem módosíthatja, azokat csak megtekintheti.

A vendég felhasználó csak a nyilvánosan elérhető adatokat láthatja. Böngészhet a már feltöltött falmászóhelyek, falak és utak között, megtekintheti a hozzászólásokat és az ezekre adott válaszokat. Semmilyen adatmódosítást nem végezhet. Ahhoz, hogy bármilyen adatot fel tudjon tölteni, regisztrálnia kell, vagy ha már van felhasználói profilja, akkor be kell jelentkeznie, ezen műveletek elvégzése után egy magasabb jogosultsági körrel fog rendelkezni.

Ábra 3.1.1.3 Vendég nézet. Nem tud semmit sem szerkeszteni.

A felhasználókezeléssel kapcsolatos műveleteket – így a regisztrációt, a bejelentkezést, a kijelentkezést és a jelszómódosítást – egy külön az erre a célra létrehozott AccountController komponensein keresztül valósítja meg. Az ASP.NET Core MVC keretrendszerben a Controllerek felelősek a beérkezett kérések fogadásáért, az adatok validálásáért és a megfelelő üzleti logika megvalósításáért, így az AccountController központi szerepet tölt be a hitelesítési folyamatok irányításában.

A bemeneti adatok érvényesítését a rendszer a szerveroldalon végzi, az ASP.NET Core beépített modellvalidációs mechanizmusára támaszkodva. A jelszavak tárolásakor a webalkalmazás nem tárolja a jelszavakat szöveges formátumban, hanem a Microsoft által biztosított PasswordHasher<TUser> komponenssel generált, egyirányú, sózott hash értéket menti el. A PasswordHasher a PBKDF2 algoritmuson alapuló, több ezer iterációval működő jelszóhash-elési stratégiát alkalmaz, mely által az adatbázis megfelel az alapvető adatvédelmi követelményeknek [29].

A felhasználói adatokat a Felhasznalok entitáshoz tartozó táblában tárolódnak, amely a szükséges személyes adatokat (név, e-mail, telefonszám, születési idő) és a hitelesítéshez szolgáló adatokat (felhasználónév, rang, hashelt jelszó) tartalmazza.

A jogosultságok alapján való szűrést a nézetoldalakon keresztül valósítjuk meg. Itt a Razor motor segítségével a kérdéses részek megjelenítését egy feltételes utasítás igaz kimeneti ága esetén tesszük elérhetővé, ezzel elkerülve, hogy külön adminisztrátori, regisztrált felhasználói és vendég nézetoldalakat kelljen létrehozni.

### 3.1.2. Felhasználó azonosítása

A felhasználók azonosítása a webalkalmazás működésének egy alapvető biztonsági és funkcionalitási eleme. A rendszerben az azonosítást a HTTP-sütik (cookies) segítségével végezzük, amelyek lehetővé teszik, hogy a szerver felismerje a böngészőt, és kapcsolatot teremtsen egy bejelentkezett felhasználó és a hozzá tartozó munkamenet között.

A sikeres bejelentkezést követően a szerver egy cookie-t generál, amelyet visszaküld a kliens böngészőjének. Ennek a sütinek nem az a célja, hogy a jelszót vagy bármilyen érzékeny adatot tároljon, hanem kizárólag egy titkosított munkamenet-azonosítót (token) helyez el, amely a szerveroldali session-hoz kapcsolódik. Ez a token lehetővé teszi, hogy a szerver az azonosító alapján minden további HTTP-kérésnél egyértelműen azonosítsa a kérést indító bejelentkezett felhasználót. Az azonosítás technikailag az ASP.NET Core cookie-alapú hitelesítési middleware-én keresztül valósul meg, amely automatikusan kezeli a sütik létrehozását, titkosítását és lejáratát [30].

A felhasználó azonosítása minden kérésnél automatikusan megtörténik: a böngésző elküldi a sütit, a szerver összeveti az abban tárolt munkamenet azonosítót a saját nyilvántartásával. Amennyiben az azonosító érvényes, a rendszer bejelentkezett felhasználóként kezeli azt. Ha a token érvénytelen, lejárt vagy manipulált, a rendszer visszavonja a munkamenetet és kijelentkezteti a felhasználót. Ez a sütialapú hitelesítési mechanizmus biztosítja, hogy a felhasználó folyamatosan egyedi módon azonosítható maradjon.

## 3.2. Felhasználói műveletek

Ebben a fejezetben az alapvető felhasználókezelést fogom bemutatni: a regisztrációt, a bejelentkezést és a jelszó módosítást.

### 3.2.1. Regisztráció

A webalkalmazás teljes körű használatához felhasználói profil szükséges. A regisztráció fülre kattintva elérhetővé válik az űrlap, ahol meg kell adni a vezeték- és keresztnevet, az e-mail címet, a születési időt, a telefonszámot és egy egyedi felhasználónevet, valamint a jelszót. A rangot a felhasználónak nem kell megadnia, mivel minden regisztrációnál automatikusan az alapértelmezett "user" rang adódik ki.

Az adat-annotáció révén a bemeneti inputokba csak olyan adat kerülhet, amit a rendszer fogadni is tud, nullértékű adatokat nem lehet továbbítani. Miután minden mezőt megfelelően kitöltöttünk, elindul a regisztráció folyamata. Első lépésként leellenőrizzük, hogy a születési idő nem mutat-e jövőbeli időpontra. Ezután ellenőrizzük az e-mail címet, ezzel elkerülve, hogy egy e-mailhez több felhasználó is tartozzon. Ezt követően ellenőrizzük a felhasználónevet is, amelynek szintén egyedinek kell lennie. A jelszavak ellenőrzését az adat-annotációk által generált szerveroldali validációval történik. Ha minden ellenőrzés sikeres, létrehozzuk a felhasználót a megadott adatokkal és a jelszavát hashelt formátumban továbbítjuk a Felhasznalok modellhez tartozó táblába. A sikeres regisztrációt követően a felhasználót automatikusan bejelentkeztetjük.

### 3.2.2. Bejelentkezés

Amennyiben a felhasználónak van már felhasználói profilja és hozzá szeretne férni a webalkalmazás teljes funkcionalitásához, akkor be kell jelentkeznie. Ehhez a bejelentkezés fülre kattintva, a felhasználó a dedikált bejelentkezési felületre jut.

Itt is az adat-annotáció révén nem szükséges az adatok ellenőrzésével külön foglalkozni, mivel a validáció már a LoginViewModel tartalmazza és érvényesíti a szabályokat, és nem engedélyezi a hibás adat továbbítását. A bejelentkezéshez meg kell adni az e-mail címet és a hozzá tartozó jelszót. Ha a megadott e-mail létezik, és a jelszó megfelelő, a bejelentkezési folyamat sikeresen végbemegy. A jelszó ellenőrzése az űrlapról beérkező jelszómező tartalmát az ASP.NET Core által beépített jelszóellenőrző metóduson keresztül kerül ellenőrzésre, amely a beérkező jelszót a tárol, sózott hash alapján hitelesíti.

A bejelentkezési hibakezelés során kiemelt figyelmet fordítunk a biztonságra. A hibák két fázisban jelenhetnek meg a felhasználónál: a ViewModel szintű ellenőrzésén megbukik, például nem ad meg jelszót (adat-annotációs hiba), megkapja az alap „A mező kitöltése kötelező.” hibaüzenetet vagy ha hibás az e-mail akkor a „Hibás email!” üzenetet. Ezt követően a hitelesítési fázisban, ha hibás e-mail-jelszó párost ad meg a felhasználó, akkor a rendszer csak egyszerűen „Hibás email vagy jelszó” üzenetet adja vissza. Ezzel a megoldással megakadályozzuk annak lehetőségét, hogy a támadók információt szerezzenek az adatbázisban tárolt e-mail címekről, növelve az adatbázis biztonságát.

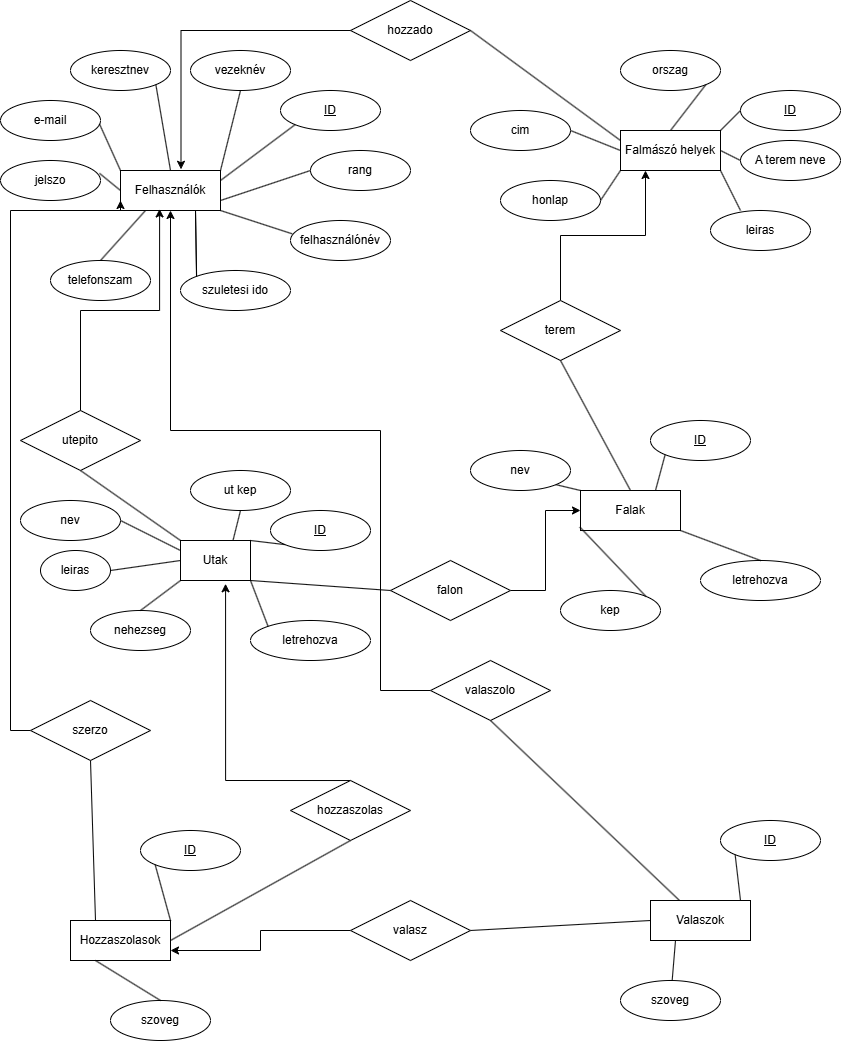
### 3.2.3. Jelszó módosítás

A bejelentkezett felhasználó egyetlen profilmódosítást tud végrehajtani, ez pedig a jelszó módosítása. A művelet végrehajtásához először be kell jelentkezni, majd a jelszó módosítása fülre kattintva megjelenik a jelszómódosítási felület. Ezen a felületen három beviteli mező áll rendelkezésre: a régi jelszó, az új jelszó és az új jelszó megerősítése mező. Itt is az adat-annotáció segítségével először ellenőrizzük, hogy minden mező helyesen van-e kitöltve. Amennyiben minden adat ki van töltve, akkor ellenőrizzük, hogy az új jelszavak megegyeznek-e, ha igen, akkor az adatokat elküldjük a szerver felé, ahol ellenőrizzük, hogy a régi jelszó illeszkedik-e az adatbázisban szereplő, hashelt változattal. Ha az ellenőrzés sikeres, végrehajtjuk a jelszómódosítást. Az új jelszó hashelt változatával frissítjük az adatbázist.

## 3.3. Adatbázis műveletek (CRUD)

### 3.3.1. Az adatbázis felépítése

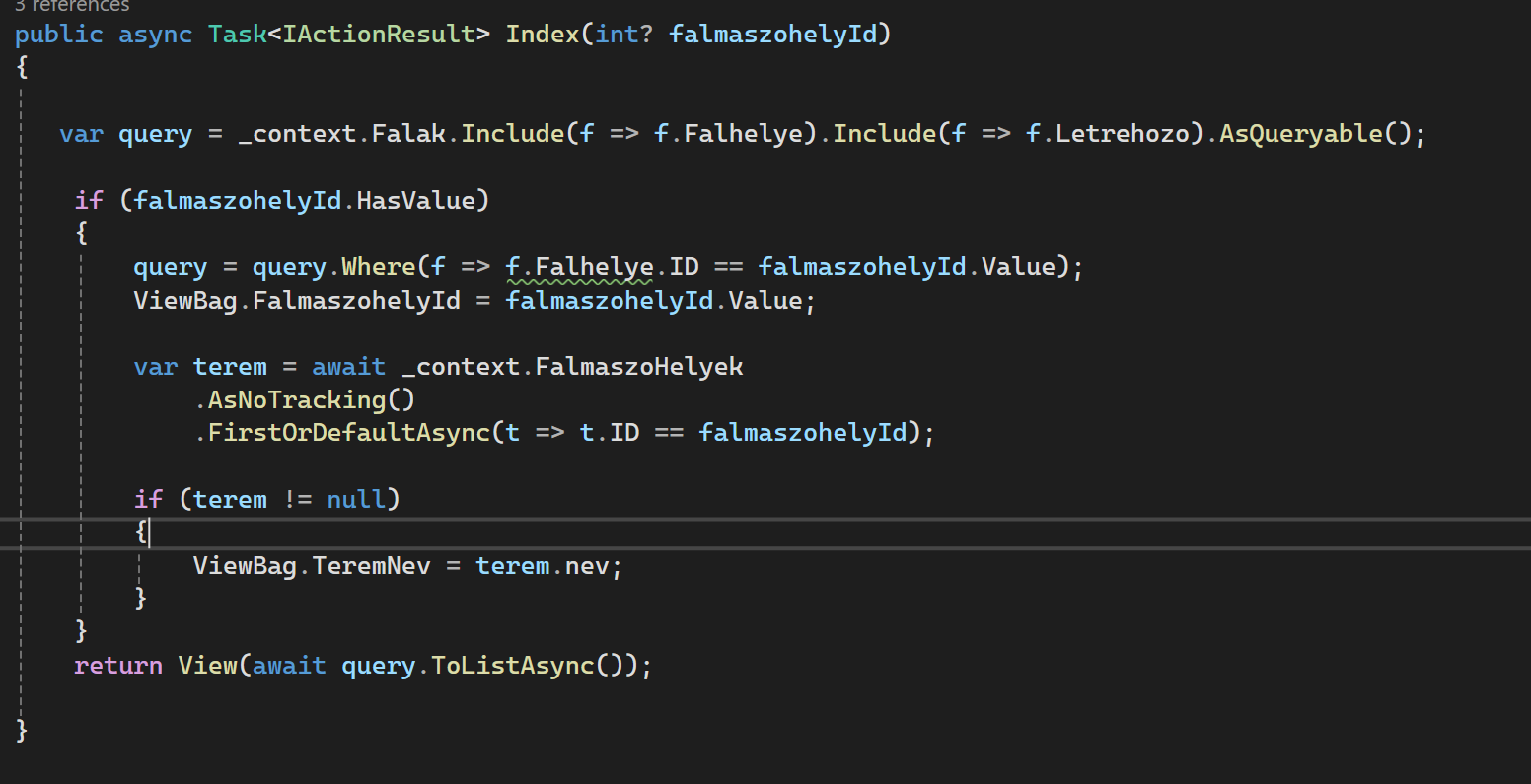
A webalkalmazás adatperzisztenciáját a SQLite adatbázis biztosítja, amely a projekt igényeihez igazodva egyszerű, gyors és könnyen karbantartható adattárolást tesz lehetővé. Az adatokhoz való hozzáférést és a sémakezelést az Entity Framework Core (EF Core) ORM technológiája valósítja meg az EFContext osztályon keresztül. Ez az osztály felelős az összes releváns entitás, úgymint a FalmaszoHelyek, a Falak, az Utak, a Hozzaszolasok, a Valaszok és a Felhasznalok tábláinak kezeléséért. A 3.3.1.1-es ábra szemlélteti az adatbázist. Az EFContext biztosítja az adatok létrehozásához, módosításához, lekérdezéséhez és törléséhez szükséges alapvető CRUD műveleteket. A projekt logikájában minden adatkezelési művelet a megfelelő Controllerhez érkezik: a FalmaszoHelyekController felel a falmászó helyekkel kapcsolatos műveletekért, a FalakController a falakért, az UtakController az utakért. A HozzaszolasokController és a ValaszokController kezeli a hozzászólásokat és az azokra adott válaszokat. A felhasználói hitelesítési folyamatokat (bejelentkezés, regisztráció, jelszóváltoztatás, kijelentkezés) az AccountController látja el, míg a FelhasznalokController felel az egyéb felhasználói adatok kezeléséért. Ezek a controllerek nem csupán összegyűjtik a felhasználótól érkező adatokat, hanem elvégzik az érvényesítést, és meghívják az adatbázist módosító aszinkron műveleteket.



Ábra 3.3.1.1. E-K diagram az adatbázis tábláiról és a köztül lévő kapcsolatokról

### 3.3.2. Lekérdezési műveletek

A webalkalmazás működésének egyik leggyakoribb folyamata az adatok lekérdezése, amelyet a felhasználók az oldal böngészése során folyamatosan igénybe vesznek. A navigáció hierarchikusan épül fel: a falmászó helyek listájáról kiindulva, a „A teremben található falak” gombra kattintva kerül sor az adott helyhez tartozó Falak listázásra. Tovább lépve, a falak oldalon lévő „A fal útjai” gombra kattintva az adott falhoz tartozó Utak kerülnek listázásra. Ez a szelektív betöltés biztosítja, hogy a lekérdezések minden esetben csak a szükséges adatokat töltsék be, ami gyorsabb oldalbetöltési időt és jobb felhasználói élményt eredményez.



Ábra 3.3.2.1. A fal szűrése falmászó hely alapján, C# kód a FalakController-ből

Az utak listázása oldalon jelennek meg a hozzá tartozó hozzászólások és az azokhoz tartozó válaszok is. Az adatok betöltéséről a FalakController-ben lévő Utak(int id) akció során egyetlen adatbázis-lekérdezéssel történik. Az EF Core Eager Loading mechanizmusának (Include és a ThenInclude metódusoknak) köszönhetően az adatok több hierarchikus szinten is betöltődnek, biztosítva, hogy az egyes utakhoz tartozó felhasználói visszajelzések naprakészek legyenek.

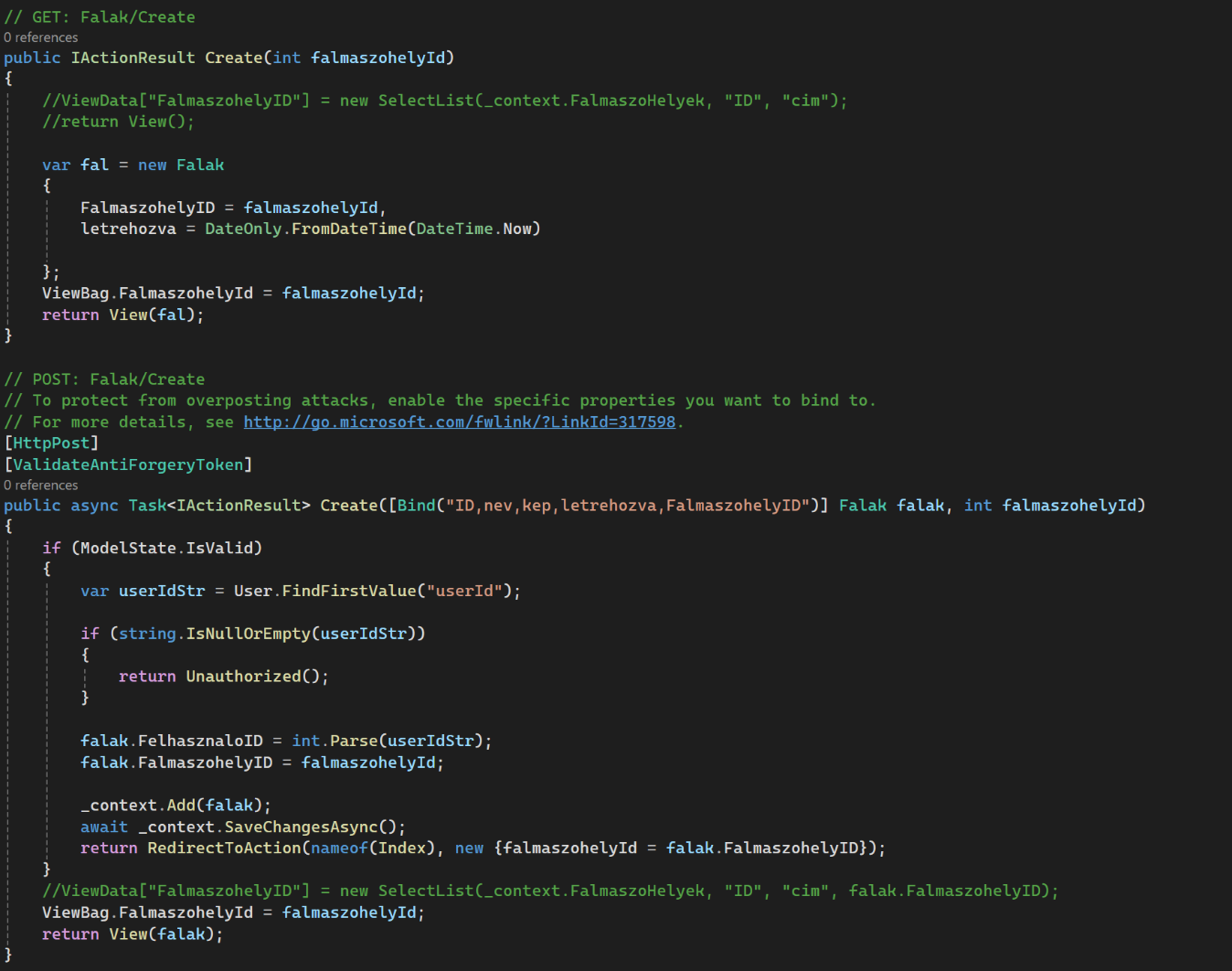
### 3.3.3. Létrehozási műveletek

Az új adatok adatbázisban történő rögzítését a felhasználó által kitöltött űrlapok elküldése indítja el. Mielőtt az alkalmazásban egy új entitást – legyen szó FalmaszoHelyek-ről, Falak-ról, Utak-ról, Hozzaszolasok-ról vagy Valaszok-ról – létrehoznánk, az alkalmazás elvégzi az adat-annotáción alapuló szerveroldali érvényesítést annak biztosítására, hogy a felhasználó az űrlapmezőket helyesen és hiánytalanul töltse ki. Az űrlapmezők kitöltésének a helyességét az ASP.NET Core beépített modellvalidációs rendszere biztosítja. A modellosztályokban elhelyezett adat-annotációs attribútumok (például [Required] [DataType] [EmailAddress]) határozzák meg a megengedett formátumot és a kötelező mezőket. Ezeket a keretrendszer automatikusan ellenőrzi a ModelState.IsValid értékén keresztül, így a Controllerek csak akkor hajtják végre a tényleges létrehozási műveletet, ha minden mező megfelel a validációs szabályoknak. Sikertelen ellenőrzés esetén az oldal újratöltődik, és az űrlapmezők alatt a megfelelő hibaüzeneteket jeleníti meg. Amennyiben a bemeneti adatok megfelelnek a modell elvárásainak, az adott Controller meghívja a Create metódust, majd a sikeres mentést követően a rendszer lekéri a szülő entitást, és ez alapján visszatér a listanézet oldalra.

A létrehozás folyamatában kiemelt hangsúlyt kap a hierarchikus integritás és a jogosultság rögzítése. Például egy út hozzáadása esetén az UtakController először azonosítja a bejelentkezett felhasználót, akinek adatait a claim-ekből olvassuk ki. A claim-ek a felhasználóhoz kapcsolódó metaadatokat (például felhasználónév, azonosító, szerepkör), amelyeket az ASP.NET Core sütialapú hitelesítési rendszere tárol és kezel. Ha a felhasználó nincs hitelesítve, a webalkalmazás automatikusan elutasítja a kérést. A létrehozási műveletek során minden entitásnak rendelkeznie kell a létrehozón kívül egy másik érvényes szülő entitással kivéve a FalmaszoHelyek entitás, aminek csak létrehozó szülő entitása van. Ennek érvényesítését a Controller végzi, ami post metódusként, külön paraméterként megkapja a szülő entitás azonosítzóját, majd ezt az értéket beállítja az ehhez az új objektumhoz. Amennyiben a szülő entitás nem létezik, abban az esetben a művelet nem hajtódik végre.

Fontos eltérés a hozzászólások és a válaszok esetén az, hogy azokat nem a hozzájuk tartozó nézetoldalon rögzítjük, hanem egy dedikált myCreate metódus hívásával, amelyet a hozzászólások esetén a HozzaszolasokController-ben, míg a válaszok esetén a ValaszokController-ben definiáljuk. Ezeket az entitásokat azért kezelem külön, mert ezen entitásokhoz kapcsolódó CRUD műveletek nem a hozzájuk kapcsolódó nézetoldalon kerülnek megvalósításra, hanem az egyes falakhoz tartozó utaknál. A metódus automatikusan hozzárendeli a felhasználót (claim-ek alapján) és a megfelelő szülő entitást (utat vagy hozzászólást). A sikeres mentést követően a rendszer lekérdezi a szülő entitás azonosítóját, és erre hivatkozva visszatér az út listanézet oldalra. Ez a mentési logika csak abban tér el a többitől, hogy egy másik nézetoldalon belül valósítjuk meg a hozzáadást.

A létrehozás lépései az utaknál a következők: a bejelentkezett felhasználó az oldal tetején a címsor alatt található „Új út létrehozása” gomra kattintva átirányításra kerül egy új oldalra, ahol az új utat hozzá tudja adni, itt minden szükséges adat kitöltését követően a „Mentés” gombra kattintva elmenti az adatokat. Amennyiben vissza akarunk lépni az előző oldalra, az adatok mentése nélkül a „Vissza az utakhoz” gombra kattintva visszatérünk a listanézetre. A szigorú szülő- gyermek kapcsolat biztosítása után a Controller az entitást az EFContext segítségével hozzáadjuk az adatbázishoz, és a SaveChangesAsync aszinkron hívásával az új rekord véglegesen rögzítjük.



Ábra 3.3.3.1 FalakController Create metódusa új fal hozzáadása

### 3.3.4. Módosítási műveletek

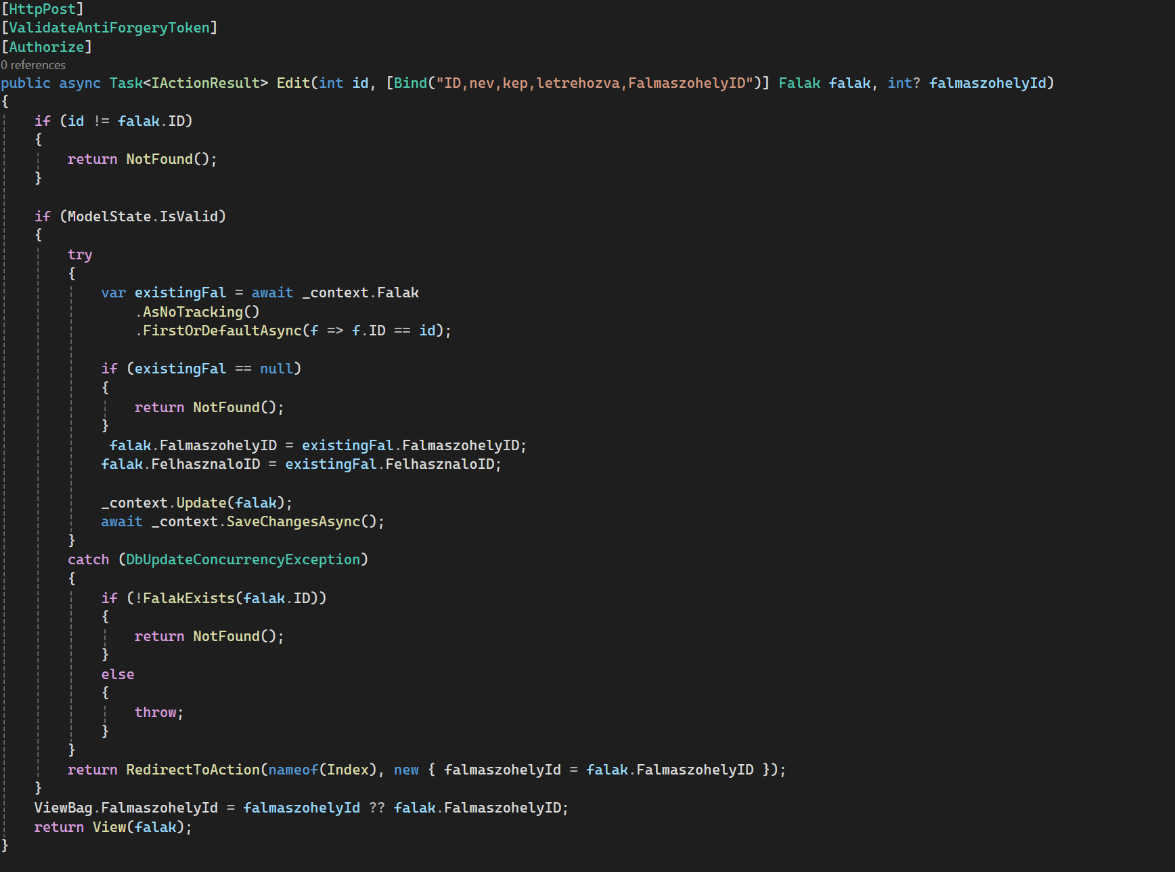
A felhasználók által bevitt adatok módosítása a szerkesztőfelületeken történik. Egy falmászó hely, fal, út, hozzászólás, válasz szerkesztését az adott Controller kezeli, amely először betölti a módosítani kívánt rekordot, majd megjeleníti a jelenlegi adatait. A felhasználó ezeket átírhatja, majd a módosításokat visszaküldi a szervernek.

Az adatmódosítás az Edit metódus hívásával történik, amely egy HTTP POST kérés. Ez alkalmas arra, hogy az érvényesített adatok feldolgozás előtt a rendszer olyan biztonsági intézkedéseket hajtson végre, amelyek megőrzik az adatbázis integritását. Mivel az űrlapmezők kizárólag a módosítható adatokat tartalmazzák, a kritikus külső kulcsok például a szülő entitást jelölő FalmaszoHelyID, illetve a felhasználót azonosító FelhasznaloID az űrlapról nem érkeznek vissza. Annak megakadályozására, hogy ezek az értékek manipulálhatók legyenek, a Controller az eredeti entitást az adatbázisból AsNoTracking módban újra lekéri. Az így kapott eredeti rekord alapján a rendszer a kritikus kulcsokat visszatölti a módosított objektumba, biztosítva, hogy a felhasználó ne változtathassa meg sem a hierarchikusan betöltött helyét sem a létrehozó személyét.

Az AsNoTracking az Entity Framework Core egy olyan lekérdezési módja, amely nem követi nyomon az entitás állapotát. Ez kifejezetten alkalmas biztonsági ellenőrzésre, mert így garantált, hogy az összehasonlítást egy „tiszta”, nem módosított adatbázis-rekord alapján végezzük el.

A módosított, és helyes logikai kulcsokkal rendelkező entitást az EFContext az Update metódussal nyomon követi. A változások véglegesítését a SaveChangesAsync metódus végzi. Ezt a szakaszt egy try-catch blokkal védjük, amelynek az a célja, hogy kezelje a *DbUpdateConcurrencyException*-t, biztosítva, hogy több felhasználó egyidejű módosítási kísérlete esetén a rendszer megfelelően reagáljon, és fenntartsa az adatok konzisztenciáját. A sikeres módosítást követően a felhasználót átirányítjuk a frissített listanézetre.

A frissítés menete: a felhasználó kiválasztja azt az entitást, amit módosítani szeretne, és rákattint a szerkesztés gombra, ez átviszi a szerkesztési oldalra, ahol a létrehozáshoz hasonló űrlappal találkozik. Ebben az űrlapban az adatok már előre ki vannak töltve, és a módosítani kívánt adatot csak át kell írni, ezután a ”Mentés” gombra kell kattintani, és a rendszer elvégzi a mentést. Hozzászólások és válaszok esetén eltérő, mert az utaknál rögtön szerkeszthetjük a hozzászólásokat és a válaszokat, anélkül, hogy egy külön oldalra lépnénk át.

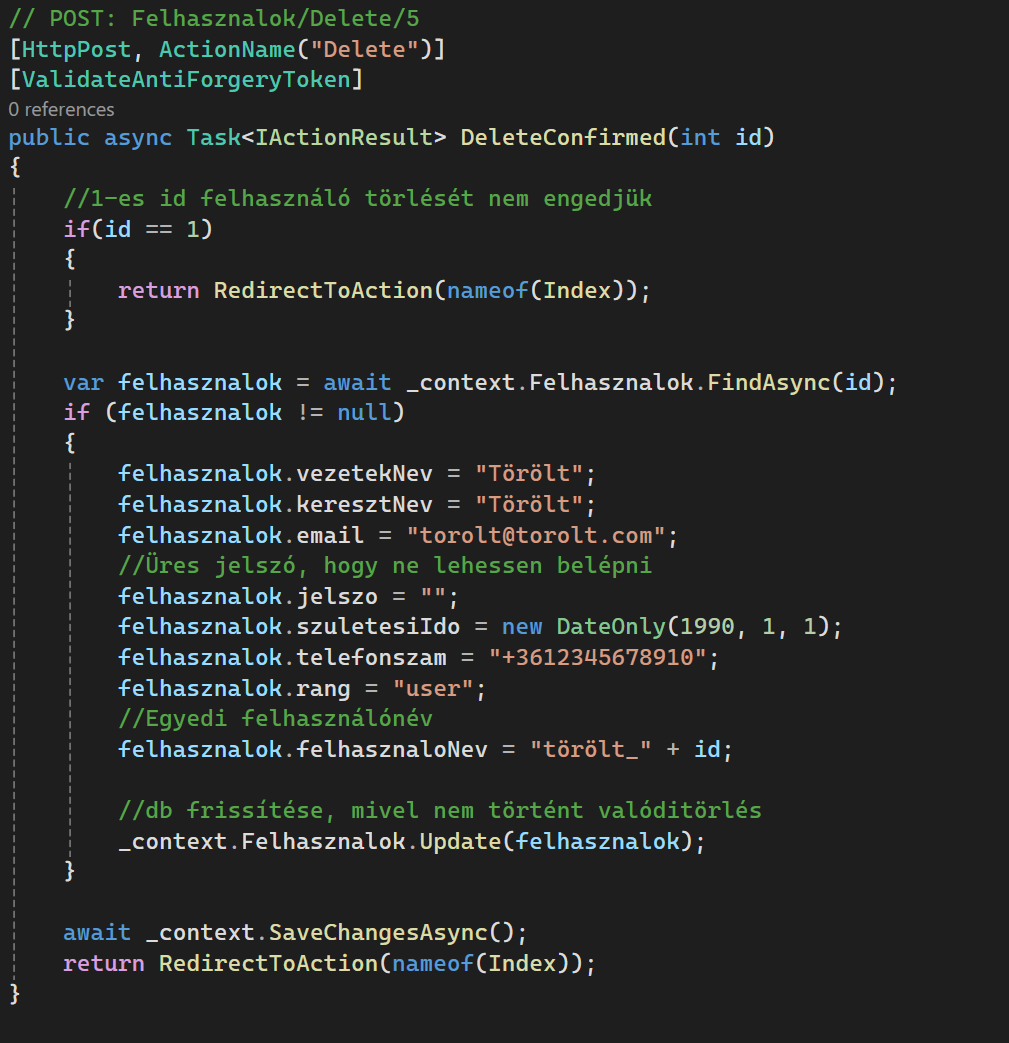


Ábra 3.3.4.1. FalakController Edit Post metódusa

### 3.3.5. Törlési műveletek

A törlési műveletek az adatkezelés legkritikusabb pontja, mivel a DeleteConfirmed metódusban nem csupán az adatbázis tartalmát, hanem a falak és az utak esetében a Firebase Cloud Storage-ban tárolt képeket is el kell távolítanunk. Erről a Firebase műveletről későbbi fejezetben fogok írni.

A törlési folyamatban először a képet távolítjuk el. Ez azért van, hogy abban az esetben, ha a képet nem sikerül törölni a Firebase-ből, akkor a rekord se törlődjön, ezzel megakadályozzuk, hogy egy nem használt kép felesleges helyet foglaljon el a Firebase Cloud Storage-ban. Miután a fal képét töröltük, az EFContext a Remove metódus hívással eltávolítja a rekordot az adatbázisból, melyet a SaveChangesAsync aszinkron metódus hívása véglegesít. Az EF Core kaszkád törlési szabályai eközben automatikusan biztosítják, hogy az egymáshoz tartozó gyermek entitásokat is eltávolításra kerüljenek, megelőzve az árva rekordok keletkezését és fenntartva az adatbázis integritását.



Ábra 3.3.5.1. Felhasználó törlése és az 1-es ID-val rendelkező felhasználó törlésének megtiltása

A felhasználók törlése másképpen zajlik, itt ugyanis nem konkrét törlést végzünk, hanem frissítést, a felhasználó minden személyes adatát átírjuk egy egységes mintára. A felhasználóneve a ” törölt” szó plusz az ID-ja lesz és ez fogja azonosítani az általa létrehozott entitásokat. Erre azért van szükség, hogyha a felhasználó sok adatot töltött fel, akkor az esetleges törlése esetén az adatbázis konzisztenciája megmaradjon. Egy másik biztonsági lépés, hogy az 1-es ID-val rendelkező admint nem lehet törölni, ezzel megakadályozzuk, hogy adminisztrátor nélkül maradjon az oldal.

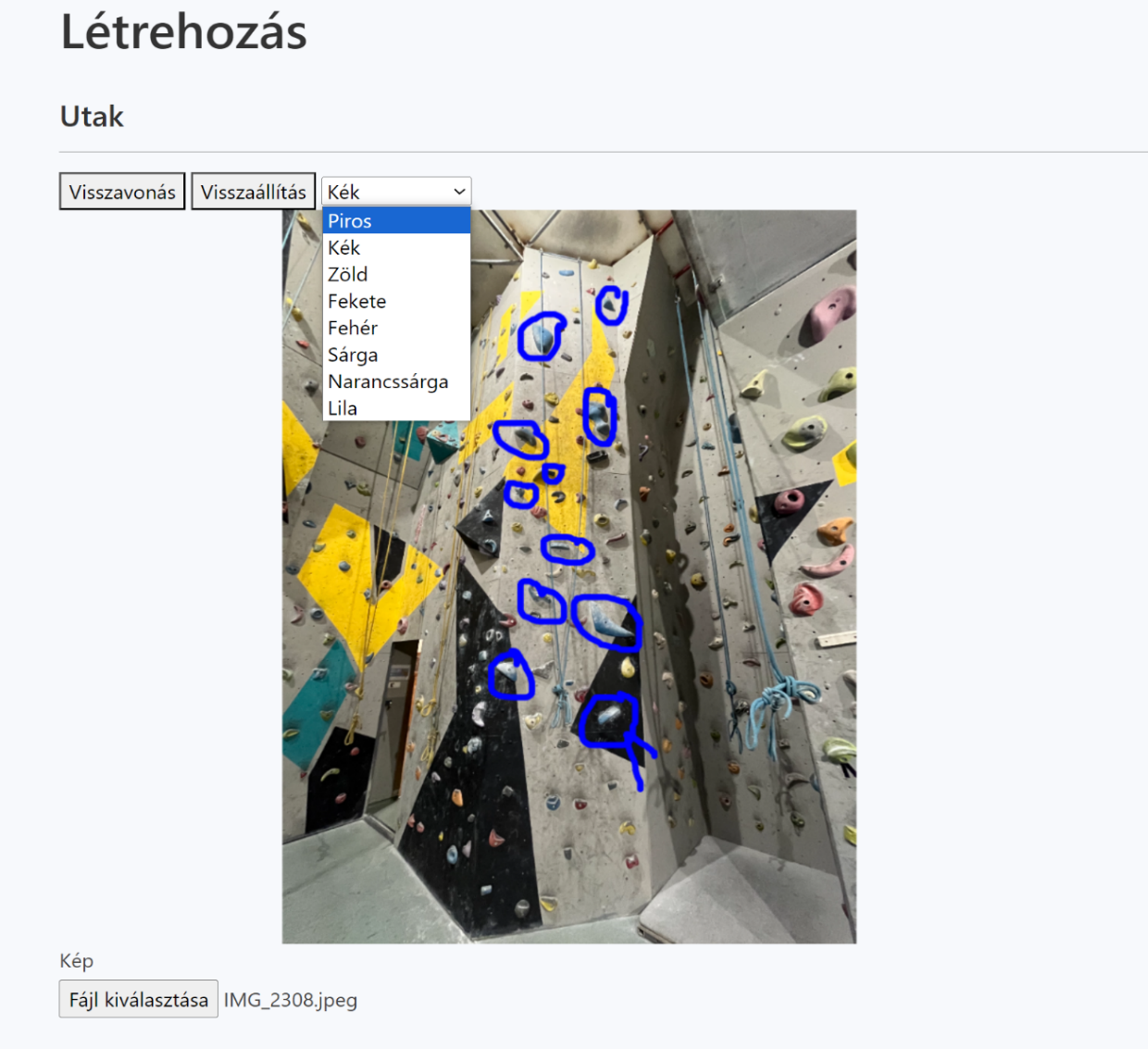
## 3.4. Képkezelési és interaktív szerkesztési funkciók

A webalkalmazás képkezelési logikája a moduláris FeltoltesController és a kliens oldali Fabric.js könyvtár szoros integrációjára épül, amely teljes körű CRUD műveleteket biztosít mind a Falak, mind az Utak entitásokhoz tartozó vizuális adatok számára. Az egész folyamat a Base64-kódolású képadatok aszinkron kezelésén alapul. A felhasználók által szolgáltatott képadatok stringként érkeznek a szerverre, ahol a FeltoltesController felelős az adatok dekódolásáért, egyedi azonosítók (GUID) alapján fájlnév generálásáért, majd a Google Cloud Storage Client segítségével történő hitelesített feltöltésért a Firebase Cloud Storage dedikált tárhelyére. A sikeres tárolás eredményeként a metódus a kép nyilvános URL-jét küldi vissza a kliensnek. Ezt az adatbázisban külső hivatkozásként rögzítjük, biztosítva ezzel egy skálázható és biztonságos megoldást a nagyméretű médiafájlok kezelésére.

### 3.4.1 Interaktív szerkesztés és állapotkezelés

Az alkalmazásban az Utak vizuális megtervezésére a Fabric.js Canvas interaktív képességeire támaszkodunk. Ezt az Utak/Create.cshtml és az Utak/Edit.cshtml nézetek implementálják. Míg a falak esetében a kép csak statikus háttérként szolgál, itt a funkcionalitás kiterjed a valós idejű szerkesztésre. Amikor egy felhasználó új képet tölt fel, vagy egy meglévő kép betöltődik, a Canvas aszinkron módon betölti a képet, és automatikusan beállítja azt háttérképként. Ezt megelőzően a képet arányosan skálázzuk, hogy optimálisan illeszkedjen a Canvas előre meghatározott méreteihez (width=”800” height=”600”), megakadályozva a torzulást és optimalizálva a rajzolási felületet. A Canvas ezután szabadkézi rajzolási módba kerül, lehetővé téve a felhasználók számára, hogy szabad kézzel felvigye a mászóutak vonalát vagy az útban használható fogásokat megjelölje.

A felhasználói műveletek nyomon követésére és a hibák kijavítására egy állapotkezelő mechanizmust valósítunk meg. Minden sikeresen befejezett rajzolási művelet után a vászon teljes állapota (a háttérképpel és az összes felvitt grafikus elemmel együtt) JSON formátumban mentésre kerül egy speciális JavaScript tömbbe, a *stateStack*-be. Ez a verem adatszerkezet biztosítja, hogy a felhasználó a Visszavonás (Undo) funkcióval bármikor vissza tud lépni egy korábbi állapotra. A Visszaállítás (Redo) funkció támogatására egy kiegészítő *redoStack* verem is használatban van. Az űrlap elküldésekor a JavaScript ellenőrzi, történt- e valódi módosítás a Canvas tartalmában. Ha az állapotváltozás detektálható, az aszinkron saveImage metódus végrehajtja a kép feltöltését a Cloud Storage-be. Ez a folyamat a Fabric.js canvas.todataURL(’img/png’) metódusát használja a vászon aktuális tartalmának egyetlen Base64 kódolású képfájlként történő kinyerésére, melyet aztán a szerver oldalra továbbítunk. A sikeres mentés után kapott új URL-t a kliensoldali logika beállítja a modellhez tartozó rejtett mező értékének, ezzel biztosítva, hogy a szerveroldali Create vagy Edit Action metódus már a frissített képhivatkozással dolgozzon.



Ábra 3.4.1.1. Képek szerkesztése az utak vizuális reprezentációjához

### 3.4.2. Mászóutak képkezelése: szerkesztés és adatintegritás

Az Utak entitás képeinek kezelése a Falak esetében alkalmazott, kétlépcsős aszinkron képkezelési logikát követi, kiterjesztve azt az útvonalrajzokat is tartalmazó Canvas-állapotra. A képfrissítés egy kritikus, aszinkron tranzakcióra épül, amelynek célja a tárhely optimalizálása és az adatintegritás fenntartása.

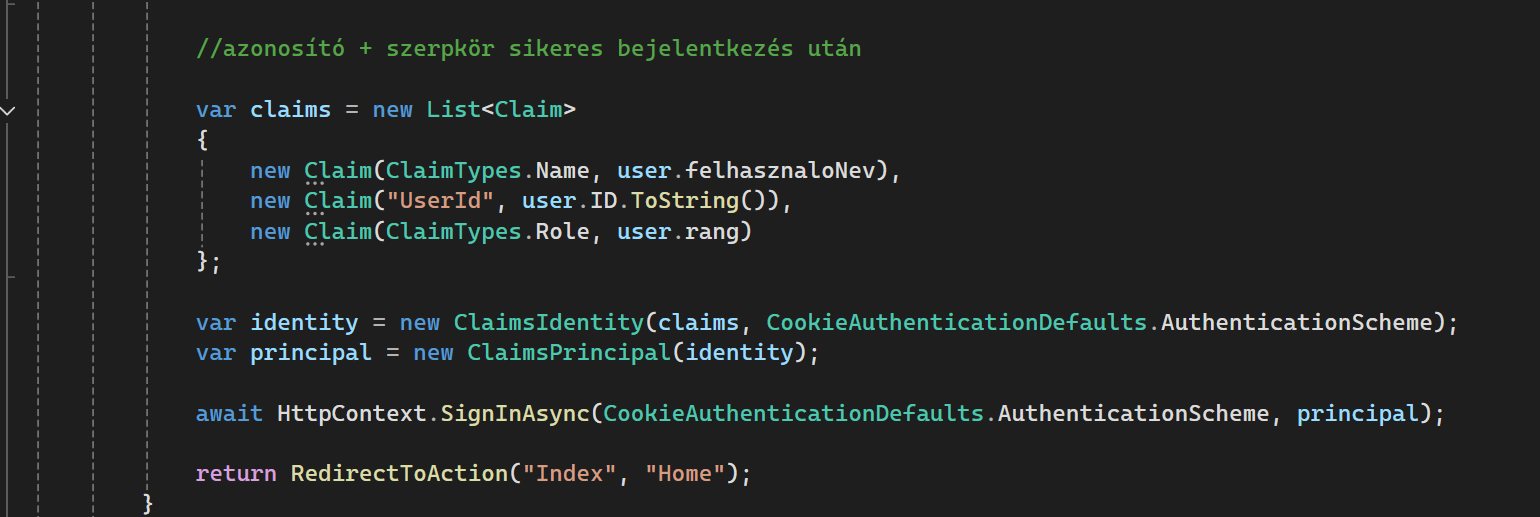
Amikor a felhasználó frissíti a falnak a képét, vagy az utat bemutató képet, a deleteOldImage aszinkron metódus első lépésben meghívja a szerveroldali api/Feltoltes/deleteimage végpontot. Ezzel a művelettel proaktívan eltávolítja a Firebase Storage-ból a lecserélt, korábbi képfájlt, amely tartalmazza a régi útvonalrajzot. A kód robosztussága érdekében a hibakezelés itt kiemelt szerepet kap: amennyiben a régi kép törlésekor hiba lép fel (például a kép már nem létezik a Storage-ben), a hiba naplózásra kerül, de a mentési folyamat nem szakad meg. Ezt követően a saveImage aszinkron metódus végrehajtja a Canvas aktuális rétegzett tartalmának feltöltését. A Canvas-ról kinyert új kép a Cloud Storage-ba kerül. A sikeres mentés eredményeként a kapott új URL beíródik a megfelelő rejtett mezőbe, garantálva, hogy a szerveroldali Edit Action metódusa már a frissített képhivatkozással frissítse az adatbázisrekordot. A Falak entitás képeinek frissítése csak abban különbözik, hogy ott nincs képszerkesztés, csak a régi kép törlése és az új kép feltöltése.

### 3.4.3. Adatintegritás és árva fájlok megelőzése (Törlés)

Az útvonal vagy a fal végleges törléséért felelős nézetek esetében az adatintegritás és a Google Cloud Storage tárhelyének tisztán tartása a legfontosabb szempont, elkerülve az árva fájlok kialakulását. A logika az Utak vagy Falak entitás adatbázisából való eltávolítás előtt felhasználja a törlendő kép URL-jét, és a FeltoltesController törlési logikáját alkalmazva eltávolítjuk a képet a Google Cloud Storage-ból. Ez a sorrend garantálja, hogy felesleges, már hivatkozás nélküli képfájlok – amelyek az utakat és a falakat vizuálisan reprezentálják – ne halmozódjanak fel a tárolóban, biztosítva ezzel a rendszer erőforrás-hatékonyságát és konzisztenciáját.

## 3.5. Biztonsági mechanizmusok

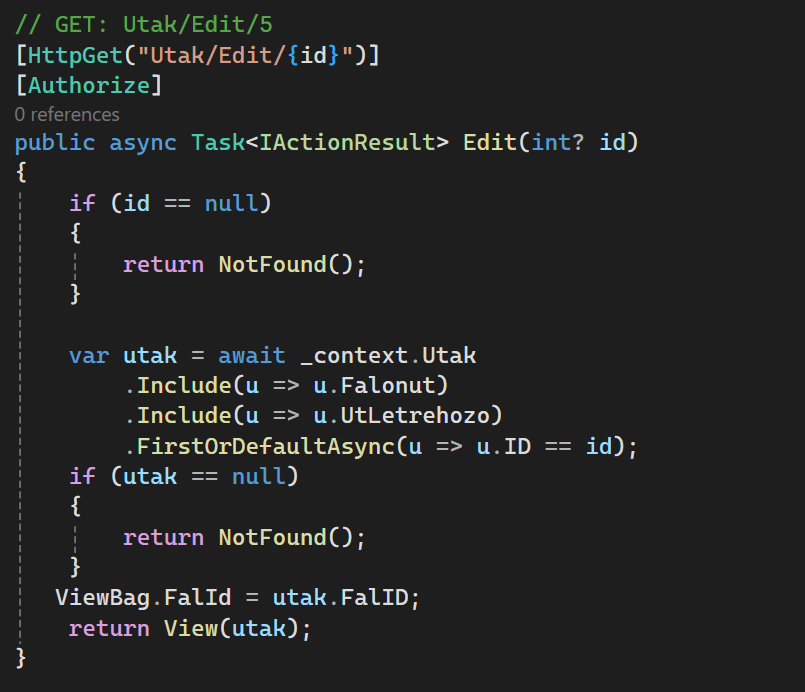
A webalkalmazás biztossági rétege a hitelesítési és a jogosultságkezelési folyamatokra épül, amelyek együtt biztosítják, hogy a felhasználók csak azokhoz férjenek hozzá, amelyhez megfelelő engedéllyel rendelkeznek. Az alkalmazásban a hozzáférés-szabályozást az Authorization elvre épül, amely az ASP.NET Core beépített jogosultságkezelési mechanizmusát használja, annak biztosítására, hogy a felhasználók kizárólag azokhoz az erőforrásokhoz férjenek hozzá, amelyekhez rendelkeznek megfelelő jogosultsággal [31]. A bejelentkezési folyamatot az AccountController valósítja meg, ahol, a felhasználó által megadott adatok szerveroldali hitelesítése után létrejön az a titkosított munkameneti cookie, amely a teljes felhasználói munkamenetet alapját adja. A bejelentkezést követően a rendszer egy olyan cookie-t hoz létre, amely egy titkosított tokennel azonosítja a felhasználót, és a böngésző minden további kérése során automatikusan visszaküld. A szakdolgozat 3.1.2. fejezetében már megjelenő logikát a belépési metódus konkrét implementációja valósítja meg: a bejelentkezés után a SingInManager a saját kódbázisban szereplő AddClaimsIdentity eljárással hozza létre azt az Identity.Cookie munkamenetet, amely alapján a szerver minden kérésnél felismeri a felhasználót.



Ábra 3.5.1. Bejelentkezési süti létrehozása a felhasználók azonosítására

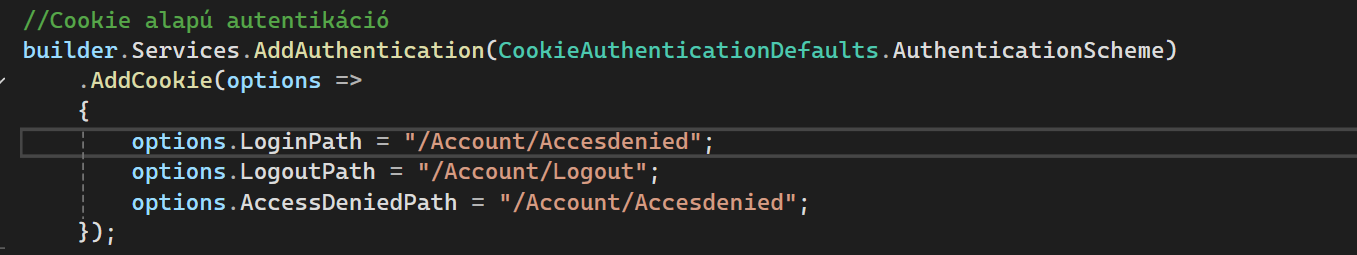
A jogosultágkezelés ehhez a hitelesítési réteghez társul. A rendszer a vezérlők és műveletek védelmét az ASP.NET Core beépített [Authorize] attribútumával valósítjuk meg. A forráskódban ez a jelölés több helyen is megjelenik, például az UtakController módosító műveleteiben, ahol a rendszer nem engedi végrehajtani az adatváltoztatást olyan felhasználónak, aki nincs bejelentkezve. A jogosultsági szabályok nem csak azt ellenőrzik, hogy a felhasználó, be van-e jelentkezve, hanem azt is, hogy a saját rendszerben szereplő rangja – amelyet a Felhasznalok modell tart nyilván – elegendő-e az adott művelet végrehajtásához. A legtöbb esetben csak azt ellenőrizzük, hogy az adott felhasználó be van-e jelentkezve. Annak eldöntésére, hogy az adott felhasználónak milyen jogosultsága van, azt a bejelentkezéskor létrehozott sütikben van tárolva és onnan kerül kiolvasásra annak eldöntésére, hogy rendelkezik-e megfelelő jogkörrel. A csak admin jogkörrel elérhető Felhasznalok tábla Controllerjében az [Authorize(Roles = "admin")] annotációt használtam. Arra pedig, hogy a Hozzaszolasok és a Valaszok entitások nézetoldalait senki se érje el arra pedig a [Authorize(Roles = "senki")] annotációt használtam. Ehhez egy nem létező jogkört írtam bele. A szakdolgozat 3.1. fejezetében már bemutatott rangok kezelése itt válik ténylegesen működő biztonsági tényezővé: a rendszer például nem enged módosítani olyan adatot, amelyet nem az adott felhasználó hozott létre, vagy nem rendelkezik adminisztrátori szinttel.

Ábra 3.5.2. Jogosultság korlátozása az UatkController Edit Get metódusára



A jogosultsági hibák kezelésére a rendszer az AccessDeniedPath útvonalat alkalmazza, amelyet a Program.cs tartalmaz. Amikor a felhasználó egy olyan erőforrást próbál elérni, amelyhez nincs megfelelő jogosultsága, a rendszer automatikusan átirányítja erre az oldalra. Ez az oldal az AccesDenied.cshtm. Ezzel a megoldással biztosítjuk, hogy a jogosultsági hiba ne technikai kivételként jelenjen meg, hanem felhasználóbarát módon jelezze a hozzáférés megtagadását. Az implementáció lényegi eleme, hogy a rendszer a hitelesítést és a jogosultságkezelést teljesen külön kezeli: először azonosítja a felhasználót a cookie alapján, majd külön folyamatban vizsgálja, hogy az adott felhasználó jogosult-e a kért művelet végrehajtására.

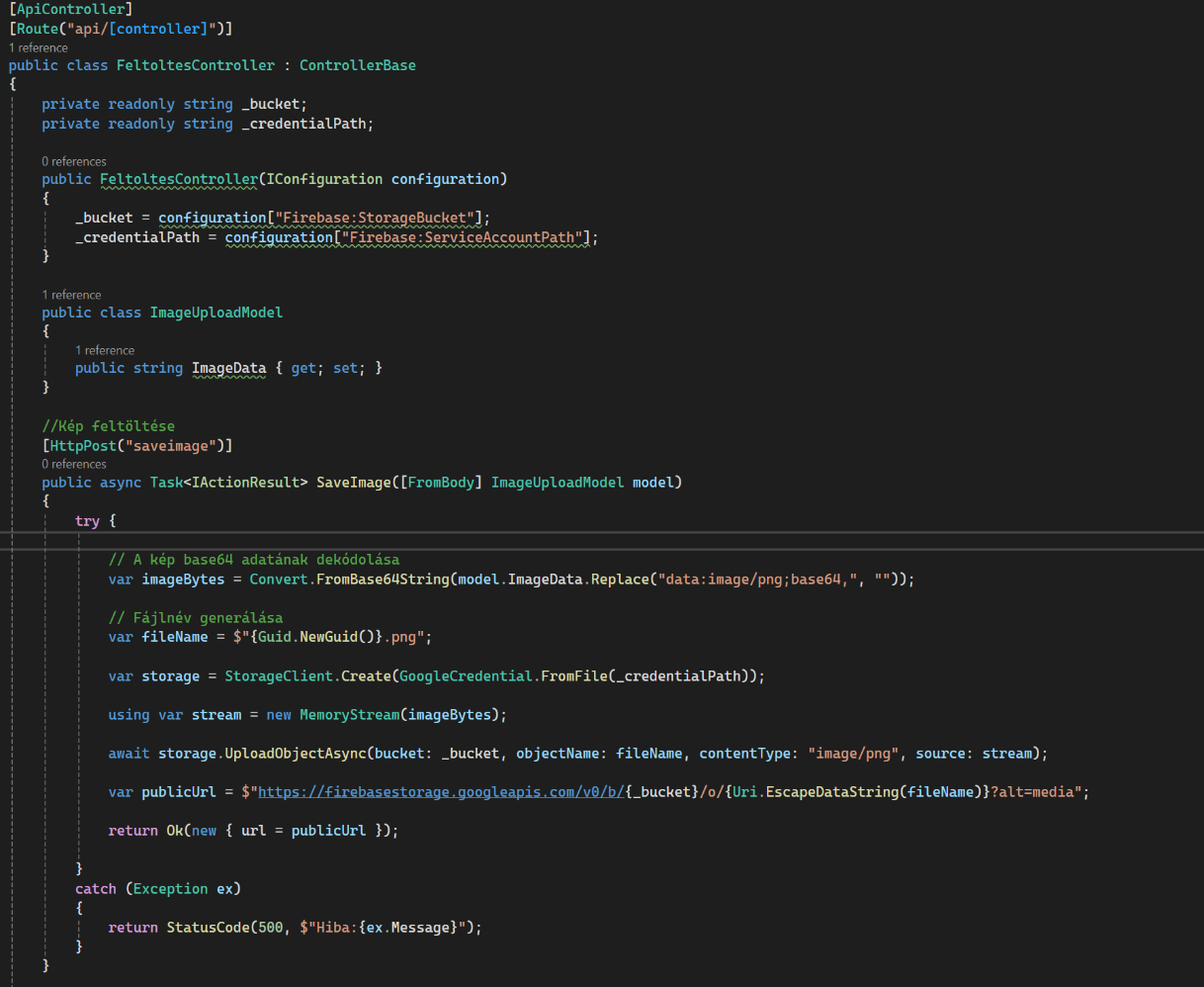
## 3.6. Külső szolgáltatások integrációja



Ábra 3.5.3. Jogosultság ellenőrzés és átirányítás

A képfájlok kezelése az alkalmazás egyik legkritikusabb funkciója, mivel a falak és az utak vizuális megjelenítése nagyméretű állományokat érint. Ezek tárolása nem a webalkalmazás saját tárhelyén történik, hanem a Firebase Cloud Storage szolgáltatásán keresztül, amelyhez a rendszer a Google Service Account hitelesítési mechanizmusát használja. A tároló eléréséhez szükséges hitelesítési adatok a projekt gyökérkönyvtárában található firabase-adminsdk.json fájlban szerepelnek. Ennek a fájlnak az elérési útját és a Firebase Bucket nevét a rendszer a konfigurációs állományból (IConfiguration) olvassa be a FeltoltesController konstruktorába. Ezeket az adatokat felhasználva hozom létre azt a StorageClient példányt, amelyet az összes további fájlművelet alapjaként használunk.

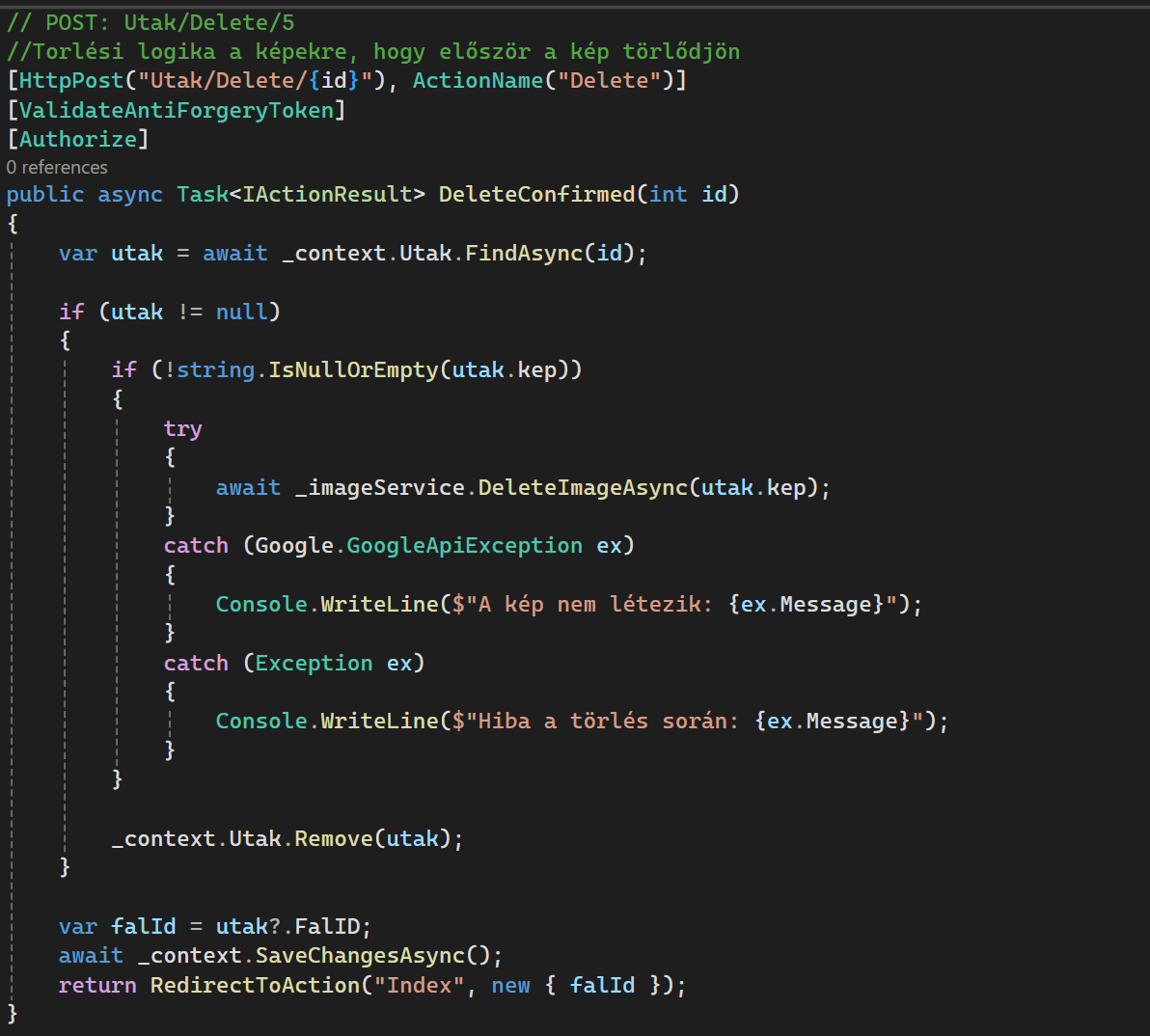
A képfeltöltési folyamat a FeltotesController API-vezérlőn keresztül zajlik, ahol a felhasználó által kiválasztott kép Base 64 kódolt streamként érkezik be. A Controller elvégzi a dekódolást, majd a létrejött adatfolyamot továbbítja a StorageClient UploadObjectAsync metódusába. A feltöltési folyamat szorosan illeszkedik a képkicserélési logikához: a rendszer először feltölti az új képet a Firebase tárhelyére, ezt követően pedig a szolgáltatás által generált publikus URL-t elmenti a megfelelő entitás (út vagy fal) adatbázismezőjébe. Ezzel biztosítjuk, hogy mindig a legfrissebb képet jelenítse meg.



Ábra 3.6.1. FeltotesContoller Kép feltöltése

A törlési folyamat ennél összetettebb, mivel a Firebase letöltési hivatkozásai jellemzően paraméterezett, escapelt URL-ek, amelyek nem tartalmazzák közvetlenül a törlendő fájl nevét. Ezt a problémát a saját implementációmban szereplő ExtracktFileNameFromUrl metódusban szereplő módon oldjuk meg. A metódus a hivatkozás azon részét vágja ki és dekódolja, amely a tárolóban szereplő tényleges állománynévnek felel meg. A fájlnév kinyerése után a törlési kérelem a StorageClient DeleteObjectAsync metódusával hajtódik végre.

A törlési logikát úgy alakítottam ki, hogy az entitás törlése előtt az eredeti URL alapján automatikusan eltávolítjuk az állományt a tárhelyről. Ez a logika megakadályozza, hogy felesleges árva fájlok maradjanak a Firebase-ben és hosszú távon optimalizálja a tárhelyhasználatot.



Ábra 3.6.2. Utak kép törlése először majd a teljes adatbázis rekord törlése

A Firebase Storage integráció ezzel olyan, a szerver saját erőforrásait tehermentesítő, skálázható megoldássá válik, amely teljes mértékben illeszkedik a rendszer működéséhez. A fájlműveletek biztonságos, hitelesített kapcsolaton keresztül zajlanak, a külső tároló használata pedig lehetővé teszi, hogy a rendszer nagy felbontású képekkel is hatékonyan működjön.

# 4. Összegzés

A szakdolgozatomban megfogalmazott célkitűzést – egy olyan webalkalmazás létrehozása, amely rendszerezetten támogatja a falmászó közösséget – sikeresen megvalósítottam. A projekt során kialakítottam egy modern, ASP.NET Core MVC alapú architektúrát, melyben az Entity Framework Core ORM-et, a SQLite adatbázist és a Firebase Cloud Storage szolgáltatást használtam az adatok tárolására és biztonságos kezeléséhez. A webalkalmazás fejlesztésekor kiemelt figyelmet fordítottam a felhasználói élmény optimalizálására, a személyes profilok kezelésére, egy többszintű jogosultsági rendszer kialakítására, valamint a Fabric.js könyvtár beépítésével a dinamikus, képalapú interakciók biztosítására.

A rendszerben kezelem a három különböző felhasználói szerepkört: a Vendég, a Regisztrált felhasználó és az Adminisztrátor számára eltérő hozzáférést biztosítok. Ezzel a szerepkörmodellel biztosítom, hogy a felhasználók kizárólag a számukra releváns funkciókat érhessék el, miközben fennmarad a rendszer biztonsága és adatainak integritása. A regisztrációs, bejelentkezési és jelszó módosítási funkciókhoz a Microsoft PasswordHasher biztonságos hashelési megoldásait rendeltem hozzá, ezzel biztosítottam a felhasználói adatok védelmét. A sütialapú hitelesítési folyamat révén tettem lehetővé, hogy a rendszer folyamatosan azonosítsa a bejelentkezett felhasználót anélkül, hogy érzékeny adatokat tárolnék közvetlenül a kliensoldalon.

Az alkalmazás egyik legfőbb funkciójaként megvalósítottam a képek kezelését és metódusait. A felhasználók által feltöltött képeket a Firebase Cloud Storage-ban tárolom, amely nagy rendelkezésre állást, skálázható és biztonságos objektumtárolási megoldást kínál. A képszerkesztési funkciót a Fabric.js alapokra építettem, lehetővé téve a felhasználók számára a szabadkézi rajzolást, továbbá a rajzolási műveletek Visszavonását (Undo) és Újraalkalmazását (Redo). Mindezen műveleteket teljes egészében a kliensoldalon valósítottam meg, így a felhasználói felület gyors és reszponzív. A képek cseréje, törlése vagy frissítése során az alkalmazás automatikusan kezeli a fájlok Firebase-ből való törlését és az új fájl feltöltését, miközben az adatbázisban frissülnek a megfelelő hivatkozások.

A projekt fejlesztése során a skálázhatóságot, a modularitást és a tiszta architektúra elveinek betartását helyeztem előtérbe. A rendszert a Code First megközelítéssel készítettem el, így a modellekből generált adatbázis-struktúra könnyen bővíthető és karbantartható.

A „Csak mássz fel” alkalmazással a célkitűzésnek megfelelő, működőképes, könnyen bővíthető és stabil alapot biztosítottam a jövőbeli fejlesztésekhez. Továbbfejlesztési lehetőségként szükségesnek látom beépíteni a mászóteljesítmények jelölésének funkcióját, amely lehetővé teszi egyfajta ranglista kialakítását arról, hogy az adott helyen ki mennyi utat mászott meg. Fontos a közösségi interakciók minőségbiztosítása is: implementálnék egy automatikus szűrő funkciót a kommentekhez, amely megakadályozza a sértő és nem megfelelő szavak használatát a hozzászólások és válaszok posztolásakor. Kiemelt feladatként megvalósítanék a webalkalmazás publikus elérését (hosting), valamint az ehhez kapcsolódó automatizált Continuous Deployment (CD) folyamatok kialakítását, amely biztosítja, hogy a GitHub repository frissülésekor az alkalmazás automatikusan kitelepüljön. Ezen túlmenően a térképi funkciók mélyebb Google Maps API integrációval történő bővítését tervezem. A jelenlegi robusztus alap lehetővé teszi, hogy a jövőben az alkalmazást egy, a mászóközösség számára még részletesebb, interaktívabb és funkciógazdagabb rendszerré fejlesszem tovább.

Forráskód: <https://github.com/meszarosmarton1007/szakdolgozatFalmaszas/>

# Irodalomjegyzék

|  |  |
| --- | --- |
| [1] | theCrag, „theCrag,” 2025. [Online]. Available: https://www.thecrag.com/. [Hozzáférés dátuma: 16 10 2025]. |
| [2] | 2. Crags, „27 Crags,” [Online]. Available: https://27crags.com/. [Hozzáférés dátuma: 20 10 2025]. |
| [3] | Természetjáró, „Sziklamászó helyek,” [Online]. Available: https://www.termeszetjaro.hu/hu/list/sziklamaszo-helyek/201908310/. [Hozzáférés dátuma: 20 10 2025]. |
| [4] | S. LLC, „Stōkt Climbing App and Platform,” [Online]. Available: https://www.getstokt.com/. [Hozzáférés dátuma: 29 11 2025]. |
| [5] | Microsoft, „What is Visual Studio?,” Microsoft Learn, 09 09 2025. [Online]. Available: What is Visual Studio?. [Hozzáférés dátuma: 11 11 2025]. |
| [6] | „Geeksforgeeks,” [Online]. Available: https://www.geeksforgeeks.org/c-sharp/what-is-entity-framework-in-net-framework/. [Hozzáférés dátuma: 20 10 2025]. |
| [7] | „Microsoft Learn,” Microsoft, [Online]. Available: https://learn.microsoft.com/en-us/aspnet/core/mvc/views/razor?view=aspnetcore-9.0. [Hozzáférés dátuma: 21 10 2025]. |
| [8] | „Razor Syntax Reference,” [Online]. Available: https://whosnailaspnetcoredocs.readthedocs.io/ko/latest/mvc/views/razor.html. [Hozzáférés dátuma: 29 10 2025]. |
| [9] | „Bootstrap Documentation,” Bootstrap Authors, 2024. [Online]. Available: https://getbootstrap.com/docs/5.3/getting-started/introduction/. [Hozzáférés dátuma: 09 11 2025]. |
| [10] | S. Bhattachergee, „Fabric.js: A Closer Look at Its Core Features,” Medium, 7 11 2023. [Online]. Available: https://medium.com/@syketweb/fabric-js-a-closer-look-at-its-core-features-a23aded4d61f. [Hozzáférés dátuma: 31 10 2025]. |
| [11] | Firebase, „Cloud Storage for Firebase Documentation,” Google, 2024. [Online]. Available: https://firebase.google.com/docs/storage. [Hozzáférés dátuma: 03 11 2025]. |
| [12] | GeeksforGeeks, „Introduction to Firebase Cloud Storage,” 2023. [Online]. Available: https://www.geeksforgeeks.org/firebase/introduction-to-firebase-cloud-storage/. [Hozzáférés dátuma: 03 11 2025]. |
| [13] | Firebase, „Get Started with Cloud Storage on Web,” Google, 2024. [Online]. Available: https://firebase.google.com/docs/storage/web/start. [Hozzáférés dátuma: 03 11 2025]. |
| [14] | Ashutec, „What is Google Firebase Storage and How to Use It for Application Development,” 2023. [Online]. Available: https://www.ashutec.com/blog/what-is-google-firebase-storage-and-how-to-use-it-for-application-development-4f9f462d1487. [Hozzáférés dátuma: 03 11 2025]. |
| [15] | A. P. Automate, „Firebase Cloud Storage Review,” 2023, [Online]. Available: https://www.appypieautomate.ai/blog/reviews/firebase-cloud-storage-review. [Hozzáférés dátuma: 03 11 2025]. |
| [16] | F. &. G. C. Platform, „Firebase and GCP Integration,” Google, 2024. [Online]. Available: https://firebase.google.com/firebase-and-gcp. [Hozzáférés dátuma: 03 11 2025]. |
| [17] | Microsoft, „Overview of ASP.NET Core MVC,” Microsoft Learn, 17 06 2024. [Online]. Available: https://learn.microsoft.com/en-us/aspnet/core/mvc/overview?view=aspnetcore-9.0. [Hozzáférés dátuma: 07 11 2025]. |
| [18] | Microsoft, „Dependency injection in ASP.NET Core,” Microsoft Learn, 2024. [Online]. Available: https://learn.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection?view=aspnetcore-9.0. [Hozzáférés dátuma: 10 11 2025]. |
| [19] | M. Fowler, „Repository,” Patterns of Enterprise Application Architecture, [Online]. Available: http://martinfowler.com/eaaCatalog/repository.html. [Hozzáférés dátuma: 08 11 2025]. |
| [20] | M. Ghanbari, „Code First vs. Database First: A Pragmatic Guide for Modern Software Engineers,” Medium, 12 07 2025. [Online]. Available: https://medium.com/@MatinGhanbari/code-first-vs-database-first-a-pragmatic-guide-for-modern-software-engineers-29ac04f1daa1. [Hozzáférés dátuma: 10 11 2025]. |
| [21] | Microsoft, „Creating and Configuring a Model - EF Core: Use fluent API to configure a model,” Microsoft Learn, 2024. [Online]. Available: https://learn.microsoft.com/en-us/ef/core/modeling/#use-fluent-api-to-configure-a-model. [Hozzáférés dátuma: 10 11 2025]. |
| [22] | Microsoft, „Migrations in EF Core,” Microsoft Learn, 01 12 2023. [Online]. Available: https://learn.microsoft.com/en-us/ef/core/managing-schemas/migrations/?view=efcore-7.0. [Hozzáférés dátuma: 08 11 2025]. |
| [23] | D. Richard, „Distinctive Features of SQLite,” SQLite.org, 31 05 2025. [Online]. Available: https://www.sqlite.org/different.html. [Hozzáférés dátuma: 08 11 2025]. |
| [24] | D. Cookies, „Understanding Object-Relational Mapping (ORM),” Medium, 03 09 2024. [Online]. Available: https://devcookies.medium.com/titleunderstanding-object-relational-mapping-orm-why-and-when-you-should-use-it-a18decc410ba. [Hozzáférés dátuma: 08 11 2025]. |
| [25] | M. Learn, „Model validation in ASP.NET Core MVC,” Microsoft Learn, 2024. [Online]. Available: https://learn.microsoft.com/en-us/aspnet/core/mvc/models/validation?view=aspnetcore-9.0. [Hozzáférés dátuma: 09 11 2025]. |
| [26] | GitHub, „What is Version Control?,” 2024. [Online]. Available: https://github.com/resources/articles/what-is-version-control. [Hozzáférés dátuma: 09 11 2025]. |
| [27] | C. R. S. H. T. Dykstra, „Kestrel web server in ASP.NET Core,” Microsoft Learn, 23 10 2023. [Online]. Available: https://learn.microsoft.com/en-us/aspnet/core/fundamentals/servers/kestrel?view=aspnetcore-9.0. [Hozzáférés dátuma: 11 11 2025]. |
| [28] | Microsoft, „Host and deploy ASP.NET Core,” Microsoft Learn, 2024. [Online]. Available: https://learn.microsoft.com/en-us/aspnet/core/host-and-deploy/?view=aspnetcore-9.0. [Hozzáférés dátuma: 11 11 2025]. |
| [29] | Microsoft, „Password hashing in ASP.NET Core,” Microsoft Learn, 20 07 2022. [Online]. Available: https://learn.microsoft.com/en-us/aspnet/core/security/data-protection/consumer-apis/password-hashing?view=aspnetcore-10.0. [Hozzáférés dátuma: 30 11 2025]. |
| [30] | Microsoft, „Use cookie authentication without ASP.NET Core Identity,” Microsoft Learn, 19 09 2025. [Online]. Available: https://learn.microsoft.com/en-us/aspnet/core/security/authentication/cookie?view=aspnetcore-10.0. [Hozzáférés dátuma: 30 11 2025]. |
| [31] | Microsoft, „Introduction to authorization in ASP.NET Core,” Microsoft Learn, 23 07 2025. [Online]. Available: https://learn.microsoft.com/en-us/aspnet/core/security/authorization/introduction?view=aspnetcore-10.0. [Hozzáférés dátuma: 30 11 2025]. |

# Nyilatkozat

Alulírott Mészáros Márton, Gazdaságinformatika BSc szakos hallgató, kijelentem, hogy a dolgozatomat a Szegedi Tudományegyetem, Informatikai Intézet Szoftverfejlesztés Tanszékén készítettem, Gazdaságinformatikus diploma megszerzése érdekében.

Kijelentem, hogy a dolgozatot más szakon korábban nem védtem meg, saját munkám eredménye, és csak a hivatkozott forrásokat (szakirodalom, eszközök, stb.) használtam fel.

Tudomásul veszem, hogy szakdolgozatomat a Szegedi Tudományegyetem Diplomamunka Repozitóriumában tárolja.

Szeged, 2025.11.30.

Mészáros Márton