Kodolányi János Egyetem

SZAKDOLGOZAT

JUHÁSZ KRISZTIÁN ISTVÁN

ÜZEMMÉRNÖK-INFORMATIKUS

ALAPKÉPZÉSI SZAK

Budapest

2025.

Kodolányi János Egyetem
Informatika Tanszék

Virtuális tárgyakra épülő kereskedelmi platform tervezése és fejlesztése a Steam ökoszisztéma által kínált eszközök és integrációs lehetőségek kihasználásával

Design and Development of a Virtual Item Trading Platform: Leveraging the Steam Ecosystem

Konzulens: Dr. Pitlik László

Készítette: Juhász Krisztián István

ÜZEMMÉRNÖK-INFORMATIKUS

ALAPKÉPZÉSI SZAK

Budapest

2025.

Tartalomjegyzék

1 Bevezetés

„*A digitális gazdaság egyik legdinamikusabban növekvő területe a videojátékokhoz kapcsolódó virtuális tárgyak kereskedelme. A globális virtuális tárgyak piaca 2024-ben 91,66 milliárd USD értéket képviselt, amely 2025-re várhatóan eléri a 112,33 milliárd USD-t, majd 2030-ra 261,36 milliárd USD-ra nő.”* (Mordor Intelligence, 2025) „*Az online piacterek (pl. Dmarket - https://dmarket.com) – különösen a több millió aktív felhasználóval rendelkező nemzetközi platformok – ma már jelentős forgalmat bonyolítanak.*” (LinkedIn/Verified Market Reports, 2025) „*Az online játék eszközök kereskedelmének piaca 2024-ben 3,6 milliárd USD bevételt ért el, amely 2033-ra várhatóan 12,2 milliárd USD-ra növekszik. A rajtuk cserélt vagy értékesített digitális eszközök pedig gyakran kézzel fogható piaci értékkel rendelkeznek.*” (LinkedIn/Verified Market Reports, 2025)

„*A Valve által üzemeltetett Steam a világ egyik legnagyobb digitális játékáruháza és tartalomterjesztő rendszere*” (SQ Magazine, 2025), amely nemcsak játékok értékesítését, hanem a hozzájuk kapcsolódó virtuális tárgyak adásvételét is lehetővé teszi. E tárgyak – amelyek megszerzése sok esetben ritka vagy időszakos – komoly keresletnek örvendenek a játékosközösségben. „*A Team Fortress 2 esetében 2011 augusztusa és 2013 májusa között több mint 70 millió cseretranzakció zajlott, amelyben több mint 300 millió darab virtuális tárgy cserélt gazdát, és 4 267 832 egyedi kereskedő vett részt.*” (Varoufakis, Y., 2012)

1.1 Kutatási és fejlesztési célok

A szakdolgozatom célja egy webalapú tárgycsere-hirdetési platform megtervezése (vö. 3.4 fejezet) és megvalósítása (vö. 3.5 fejezet), amely lehetővé teszi a Steam-felhasználók számára, hogy:

- birtokukban lévő virtuális tárgyakat (pl. sapkák, fegyverek, kozmetikai tárgyak) meghirdethessenek (vö. 3.5.2.5 fejezet)
- ajánlatokat tehessenek más felhasználók hirdetéseire (vö. 3.5.2.11 fejezet)
- böngésszenek mások által meghirdetett tárgyak között, azaz megkeressék az általuk keresett tárgyakat (vö. 3.5.2.7 fejezet)

A rendszer kialakításánál elsődleges szempont a felhasználói élmény, a biztonság (vö. 3.3. fejezet), valamint a Steam API és OpenID autentikáció integrációja (vö. 3.3.1 fejezet). A fejlesztés modern technológiákra épül: vö. a frontend oldal Angular keretrendszert (vö. 3.1.1.1 fejezet), a backend oldal pedig C#-ot használ (vö. 3.1.1.2 fejezet).

1.2 Problémafelvetés és indoklás

A jelenlegi piacterek (pl. backpack.tf) és cserefelületek (pl. posts.tf) gyakran korlátozott testreszabási lehetőséget kínálnak: „*A spell (varázslat) szerinti szűrés 2025 júniusában eltávolításra került a backpack.tf felületéről, ami a felhasználók számára jelentős funkcióvesztést jelentett*” (backpack.tf forums, 2025). Egy Reddit-felhasználó szerint „*az all-class kozmetikai tárgyak szűrése sem megoldott a backpack.tf-en, csak körülményes URL-módosításokkal (pl.? class=all paraméter kézi hozzáadásával az URL végéhez) vagy külön böngészőbővítményekkel (pl. TF2 Trading Enhanced) érhető el.*” (Reddit r/tf2, 2025)

A platformok nem minden esetben biztosítanak könnyen kezelhető felületet a felhasználók számára. Egy 2024-es Reddit-poszt szerint a TF2 felhasználói felülete borzalmasan zsúfolt és zavaros lett: "*a teljes képernyő teljesen átláthatatlan, minden lehetséges opciót bedobtak anélkül, hogy bármilyen gondolat lenne mögötte.*" (Reddit r/tf2 (1), 2024). A poszt kiemeli, hogy az áruház szervezése frusztráló: lehet szűrni osztály és kozmetikai tárgy szerint, de nincs lehetőség az osztály-specifikus sapkák leszűrésére, és nincs szűrési opció a taunts (gúnyolódások) esetében osztály alapján.

1.3 Motiváció

A szakdolgozat témájának választását kettős motiváció vezérelte: személyes tapasztalat a Team Fortress 2 kereskedési közösségben szerzett éveim alapján, valamint a piaci környezet és a virtuális tárgykereskedelmi ökoszisztéma gazdasági jelentősége (lásd 1.1 fejezet). A motiváció bemutatása két alfejezetben kerül részletezésre: a személyes tapasztalatok és problémák ismertetése, majd a piaci környezet és a globális virtuális tárgypiaci trendek elemzése.

1.3.1 Személyes motiváció

Személyes kapcsolatom a Team Fortress 2 játékkal több évre nyúlik vissza. A játék nem csupán szórakozási lehetőséget nyújtott számomra, hanem egy komplex virtuális gazdaságot (vö. 1. fejezet) is feltárt előttem, amelyben aktív résztvevővé váltam. Kezdetben a játékon belüli kereskedési rendszert használtam, ahol közvetlenül a szervereken található játékosokkal cseréltem tárgyakat. Később különböző külső platformokat is kipróbáltam, például a

backpack.tf, tf2outpost (mára már megszűnt...) és egyéb közösségi kereskedési oldalakat. Ezeknek a platformoknak a használata során számos problémával találkoztam. A játékon belüli kereskedés rendkívül körülményes folyamat: meg kell találni egy kereskedőpartnert, hozzá kell adni a Steam-en, kezdeményezni kell egy kereskedési munkamenetet, majd többszöri megerősítés után végrehajtani a cserét. A külső platformok gyakran elavult felhasználói felülettel rendelkeznek, korlátozott szűrési lehetőségeket kínálnak. (vö. 1.2 fejezet) Ezek a tapasztalatok ösztönöztek arra, hogy egy saját megoldást fejlesszek ki, amely modern technológiákra épül és kiküszöböli a meglévő platformok hiányosságait. Fejlesztői szemszögből pedig lehetőséget láttam abban, hogy a több éves Angular (vö. 3.1.1.1 fejezet) és ASP.NET Core (vö. 3.1.1.2 fejezet) tapasztalatomat egy valós problémára alkalmazzam, miközben fejlesztem a full-stack készségeimet.

1.3.2 Piaci motiváció

„*A Team Fortress 2 virtuális tárgyai egy jelentős méretű és aktív piacot alkotnak a Steam ökoszisztémán belül. Kutatási adatok szerint 2011 augusztusa és 2013 májusa között több mint 70 millió cseretranzakció zajlott le a TF2-ben, ami átlagosan több mint 100,000 darab kereskedést jelent naponta, azaz több mint egy tranzakciót másodpercenként.*” (Varoufakis, Y., 2012) Ezekben a tranzakciókban több mint 300 millió darab virtuális tárgy cserélt gazdát, és összesen 4 267 832 fő egyedi kereskedő vett részt a piacon. „*A szélesebb Steam platform is jelentős növekedést mutat. 2024-ben a Steam piaca 4,902 millió USD értéket képviselt, amely 2025-re várhatóan 5,385 millió USD-ra nő, és 2032-re elérheti a 9,195 millió USD-t, 9.6%-os éves növekedési ütemmel.*” (Intel Market Research, 2025) „*A platform 2025 első negyedévében 147 millió fő havi aktív felhasználót ért el, amely jelentős növekedést jelent a 2024-es 132 millió-főhöz képest.*” (SQ Magazine, 2025) A virtuális tárgykereskedelem tehát egy milliárdos piacot mozgató gazdasági tevékenység. A Team Fortress 2 esetében a tárgyak komplexitása - különböző ritkasági szintek, effektek, színezések, tulajdonságok - még érdekesebbé és kihívásokkal telibbé teszi a piacot. Egy jól megtervezett platform, amely hatékonyan kezeli ezt a komplexitást és javítja a felhasználói élményt, valós piaci értéket képviselhet mind a kereskedők, mind az átlagos játékosok számára.

1.4 Célcsoportok

A platform által megcélzott felhasználói csoport az azonosítása elengedhetetlen a rendszer funkcióinak és a felhasználói élmény megfelelő kialakításához. A célcsoport elemzése során figyelembe vettem a demográfiai adatokat, a technológiai jártasságot, a felhasználói szokásokat

és a speciális igényeket. Az Általam definiált célcsoport elsősorban az **„aktív Team Fortress 2 kereskedők”**

Demográfiai jellemzők

„*A Steam platform felhasználói adatai alapján a legnagyobb felhasználói csoport a **25-34 éves korosztály (38%)**, míg a **18-24 évesek 31%-ot** tesznek ki.*” (SQ Magazine, 2025) „*Egy 2024-es Reddit TF2 közösségi felmérés szerint (622 résztvevő) az életkori megoszlása a következő*” (Reddit r/tf2 (2), 2024)*:*

- 14 vagy fiatalabb (43 fő)
- 15-19 éves (246 fő)
- 20-25 éves (223 fő)
- 26-30 éves (66 fő)
- 31-40 éves (28 fő)
- 41+ éves (16 fő)

A felmérésből látható, hogy a megkérdezettek 75,4%-a a 15-25 éves korosztályba tartozik. Fontos megjegyezni, hogy ez egy nem reprezentatív, önkéntes Reddit felmérés, így az eredmények nem általánosíthatók az egész TF2 közösségre.

Aktuális játékos statisztikák (2024-2025)

„*A jelenlegi (2025 október) számok alapján, az elmúlt 30 napban átlagosan 46 602 egyidejű játékos játszik a Team Fortress 2 játékkal.*” (Live Player Count, 2025) Aktív kereskedők pontos számára vonatkozó hivatalos statisztika nem érhető el, azonban van egy jelenleg is aktív csere-igény hirdető felület, ahol fel van tüntetve a regisztrációs szám, amely **4 000-5 000 aktív felhasználó-t** jelent. (posts.tf, 2025), így egy konzervatív becslés alapján kimondható az, hogy nagyjából az aktív játékosok 5-10%-a kereskedik aktívan.

1.5 Hasznosság

A platform elsősorban a Team Fortress 2 virtuális tárgyak kereskedelmével foglalkozó játékosokat, kereskedőket és közösségeket segíti: A vizsgált kereskedési platform és funkcionalitások jellemzően a célcsoportokban (lásd 1.4 fejezet) azonosított felhasználói csoportok igényeit elégítik ki, akik vélhetően az előnyöket-hátrányokat figyelembe véve (vö. 1.2 fejezet) választanak majd a meglévő platformok és az új megoldás között.

1.5.1 Felhasználói előnyök és időhatékonyság

Az aktív kereskedők számára a platform időmegtakarítást és hatékonyságnövekedést kínál:

Időmegtakarítás: Ha egy felhasználó a saját igénye alapján keres egy konkrét tárgyat (vö. 3.5.2.7 fejezet), azt hatékonyan megtalálhatja ezen a platformon, jelentős időt megtakarítva ezzel. Egy optimalizált keresési és szűrési rendszerrel ez az idő jelentősen csökkenthető. Természetesen, a hasznosság előfeltétele, hogy a keresett tárgy meghirdetésre kerüljön a platformon, ami a felhasználói bázis növekedésével párhuzamosan javul.

Hatékonyság növekedése: A jobb szűrési lehetőségek - beleértve a spell-ek (varázslatok), paint-ek (festékek) és effektek alapján történő keresést - lehetővé teszik, hogy a felhasználók alaposabb és célzottabb keresést hajthassanak végre (vö. 3.5.2.7 fejezet). Egy hatékonyabb platform, amely átfogó szűrési lehetőségeket biztosít, felgyorsíthatja a tranzakciókat és növelheti a sikeres kereskedések számát.

Közösségi támogatás: A komment és ajánlattételi rendszer (lásd 3.5.2.11 fejezet) interaktív platformot teremt, amely lehetővé teszi a felhasználók számára - különösen a kezdők számára, hogy tanácsot kérjenek tapasztaltabb kereskedőktől, ajánlatokat tegyenek és közvetlen kommunikációt folytassanak. A platform komment funkciója segíti a biztonságos és átlátható kereskedési gyakorlatok kialakítását.

1.5.2 Fejlesztési költségek és megtérülési modell

Fejlesztési ráfordítások:

A platform kifejlesztése során az alábbi munkaóra ráfordítások merültek fel (1. táblázat).

Fejlesztési fázis	**Becsült munkaóra**	**Junior fejlesztői órabér (Ft)**	**Becsült költség (Ft)**
Tervezés és specifikáció	40 óra	4 166 Ft	166 640 Ft
Backend fejlesztés (ASP.NET Core)	100 óra	4 166 Ft	416 600 Ft
Frontend fejlesztés (Angular)	120 óra	4 166 Ft	499 920 Ft
Microservice fejlesztés (Node.js)	48 óra	4 166 Ft	199 968 Ft
Tesztelés és optimalizálás	50 óra	4 166 Ft	208 300 Ft
Dokumentáció	30 óra	4 166 Ft	124 980 Ft
Összesen	**388 óra**	4 166 **Ft**	**1 616 408 Ft**

1. táblázat - Fejlesztési költségek fázisok szerinti bontásban – Forrás: Saját számítás

A becslés junior full-stack fejlesztői órabér alapján készült (4 166 Ft/óra), amely a magyar piacon jellemző 2025-ben. Az óradíjat a frontend és C#/.NET fejlesztési minimum bruttó havi fizetés mediánjából számoltam ki. (Hays salary guide, 2025) Senior fejlesztő esetén (8 333 Ft/óra) a költség ~3,2 millió Ft körül lenne.

Üzemeltetési költségek (havi):

A kiszámolt költségek a 2. táblázat-ban tekinthetőek meg.

Költségelem	Havi költség (Ft)	Éves költség (Ft)	Forrás
Domain név (.tf)	500 Ft	6 000	Name
Hosting (VPS, 4GB RAM, 2 CPU)	4 058 Ft	48 696 FT	Rackforest
Adatbázis (PostgreSQL)	Ingyenes (VPS-en fut, külön szolgáltatás nélkül, beépített PostgreSQL installációval)	Ingyenes (VPS-en fut, külön szolgáltatás nélkül, beépített PostgreSQL installációval)	-
SSL tanúsítvány	Ingyenes (Let's Encrypt használatával, automatikus megújítással)	Ingyenes (Let's Encrypt használatával, automatikus megújítással)	-
Össz. üzemeltetési költség	4 558 Ft	54 696 Ft	-

2. táblázat - Üzemeltetési költségek havonta és évente – Forrás: Saját számítás

Bevételi modell és megtérülés:

A platform jelenleg nonprofit jellegű, ingyenes szolgáltatásként működik, így közvetlen bevétele nincs. A megtérülés alternatív módjai:

- **Prémium felhasználói funkciók (bump limit emelése, kiemelés)**
  - Becsült havi bevétel: 50 000 – 100 000 Ft
  - Feltétel: 1000+ aktív felhasználó, 5% konverzió
- **Affiliate linkek**
  - Becsült havi bevétel: 20 000 – 40 000 Ft
  - Feltétel: Napi 1000+ fő látogató
- **Összesen (optimista forgatókönyv)**
  - Becsült havi bevétel: 70 000 – 140 000 Ft

Megtérülési idő (optimista forgatókönyv):

- Fejlesztési költség: 1 616 408 Ft
- Havi üzemeltetés: 4 558 Ft
- Éves üzemeltetés: 54 696 Ft
- Havi bevétel (átlag): 105 000 Ft
- Nettó havi bevétel: 105 000 Ft – 4 558 Ft = 100 442 Ft
- Megtérülési idő: 1 616 408 / 100 442 Ft = ~16 hónap

Realisztikus következtetés

Közvetlen pénzügyi megtérülés csak akkor lehetséges, ha a platform eléri a kritikus felhasználói tömeget (5000+ fő aktív felhasználó), ami 12-18 hónapos aktív marketing tevékenységet igényelne. Fontos hangsúlyozni, hogy a platform elsődleges célja nem az **üzleti haszonszerzés**, hanem a szakmai kompetenciák fejlesztése és a közösségi hozzájárulás.

1.6 A szakdolgozat felépítése

A szakdolgozat struktúrája három fő részből áll: a bevezető fejezetek a téma hátterét és indoklását mutatják be, a fő fejezetek a rendszer tervezését és fejlesztését részletezik, míg a záró fejezetek összegzik az eredményeket és jövőbeli fejlesztési irányokat vázolnak fel. A fejezetek egymásra épülő logikai kapcsolatban állnak.

1.6.1 Általános felépítés

A dolgozat elsőként ismerteti a digitális piacterek és a Steam gazdasági modelljének hátterét (vö. 2.2 fejezet), majd bemutatja a rendszer tervezési folyamatait és architektúráját (vö. 3.4 fejezet). Ezt követően részletesen tárgyalja az implementációt (vö. 3.5 fejezet) és a tesztelés eredményeit (vö. 3.6 fejezet). Végül a záró fejezetek összefoglalják a fejlesztés tapasztalatait, kiemelik a rendszer előnyeit, valamint javaslatokat tesznek a jövőbeni bővítési lehetőségekre.

1.6.2 Szakdolgozat korlátjai

A szakdolgozat terjedelmi korlátjai miatt az alábbi témák részletes kifejtésére nem volt lehetőség, azonban felismerem ezek fontosságát a teljeskörű megértéhez:

Biztonsági tesztelés és penetrációs tesztek:

A rendszer és a szakdolgozati dokumentációm alapvető biztonságai kérdéseket megválaszol, implementál (vö. 3.3 fejezet), azonban átfogó biztonság audit és penetrációs tesztelés nem került elvégzésre. Ez különösen fontos lenne a felhasználói adatok védelme szempontjából.

Adatvédelmi és jogi aspektusok:

A GDPR szabályzat, süti politika, felhasználói szerződések jogi hátterének részletes kifejtésére nem került sor, akár önmagában is megállná a helyét egy külön szakdolgozati témának.

Monitoring és statiszika:

A rendszer monitorozása és a teljesítménymutatók folyamatos követése szakmai szempontból fontos, de a dolgozatban csak említés szintjén szerepel. (lásd 3.6.3 fejezet)

1.6.3 Formázási szabályok és jelölések

A szakdolgozat olvashatóságának és konzisztenciájának biztosítása érdekében az alábbi formázási szabályokat alkalmaztam:

Fejezetek és alfejezetek számozása:

- **Főfejezetek:** Arab számozás (1., 2., 3.)
- **Alfejezetek:** Hierarchikus számozás (1.1, 1.1.1, 1.1.1.1)
- Maximum 4 szintű mélység alkalmazása az átláthatóság érdekében

Hivatkozások és kereszthivatkozások:

- **Fejezetre való hivatkozás:** (vö. 3.5.2 fejezet), (lásd 3.5.2 fejezet), (3.5.2 fejezet)
- **Ábra hivatkozás:** (X. ábra) – leírás
- **Forráshivatkozás:** (Forrás: cím, év), + dőlt betűstílus

Hangsúlyozás és kiemelés:

- **Fontos fogalmak:** félkövér betűstílus
- **Technikai rövidítések:** nagybetűs (API, JWT)

Listák és felsorolások:

- **Pontozott lista:** Szövegfolyamban történő felsoroláshoz
- **Számozott lista:** Sorrend vagy lépések esetén

Táblázatok:

- **Cím:** Minden tábláznak van címe

2 Szakirodalmi áttekintés

A szakdolgozat témájának mélyebb megértéséhez elengedhetetlen a virtuális tárgykereskedelem elméleti alapjainak és piaci környezetének áttekintése. Ez a fejezet bemutatja a Team Fortress 2 tárgyrendszerének működését, a Steam platform szerepét a virtuális gazdaságban.

2.1 Virtuális tárgyak

Ezek a tárgyak egy harmadik feles platformon szerepelnek, a Steam-en (vö. 2.3 fejezet). A Steam egy olyan felület, ahol regisztráció után lehetősége van a felhasználónak játékokat vásárolni, amelyek a profiljában kerülnek eltárolásra. Ezek a játékok időnként biztosíthatnak jutalmakat, amelyek valós piaci értékkel rendelkeznek (vö. 1.3.2 fejezet). Az említett virtuális termékekre évről-évre több tényező (pl. ritkaság, limitált időszakú események-halloween, tf2 játékfrissítések, youtube tartalomgyártók által készített tartalmak) hatására egyre nagyobb a kereslet (vö. 1.3.2 fejezet).
Ez betudható annak, hogy bizonyos tárgyak megszerzésére (pl. unusual típusú sapkák – Burning Flames Team Captain, Golden Frying Pan) kevés az esély.
Az alapszintű (unique) tárgyakat játék során lehet megszerezni, minél több időt tölt valaki a játékkal, annál több tárgyat szerezhet meg, bár a megszerezhető tárgyak száma limitálva van (~ 10 db), heti szinten. Az unique típusú tárgyak között több fajta tárgy van, a fajta, mint fogalom azt jelenti, hogy a játék során, milyen tárgyként használható, ezeknek a fajtáknak a listája (3. táblázat – 4. táblázat).

Tárgyfajták

Fő kategória	**Alcsoport**	**Leírás**
Weapon (Fegyver)	Primary Weapon	Főfegyver, pl rakétavető, puska
-	Secondary Weapon	Másodlagos fegyver, pl pisztoly
-	Melee Weapon	Közelharci eszköz, pl balta, baseball ütő
Taunt (Gúnyolódás)	-	Animációk és mozdulatok, pl. tánc, nevetés
Hat (Sapka)	-	Karakter által viselt fejfedő
Cosmetic (Ruha)	-	Karakter megjelenését módosító öltözet
Parts (Alkatrész)	-	Speciális statiszikai funkciókat adhat fegyverekhez/ruhákhoz
Spells (Varázslat)	-	Időszakos eseményeken szerezhetőm kinézetmódosító effekt
Paint (Festék)	-	Ruhák és sapkák színének módosítása

3. táblázat - Team Fortress 2 tárgyak fő kategóriái – Forrás: Saját összegzés

Ritkasági típusok

Típus	**Leírás**
Unique	Általános tárgy, alapértelmezett ritkaság
Strange	Nyilvántartja az eliminált ellenfelek számát
Genuine	Korlátozott mennyiségben kiadott, ritka
Unusual	Látványos, mozgó háttéreffekt, rendkívül ritka
Strange-Unusual	Effekt + statisztikai számláló egyben

4. táblázat - Team Fortress 2 tárgyak ritkasági típusai – Forrás: Saját összegzés

A táblázatok csak bevezetés jelleggel készültek, az elsődleges cél az volt, hogy ismertessem a játék és a tárgyak közötti kapcsolatot, illetve az említett típusok, tárgyak tulajdonságának számosságát szerettem volna röviden bemutatni, mert a megfelelő háttérismeretek nélkül a projekt lényege laikusok számára nem válik magától értetődően érhetővé. Ezek mellett tényleg rengeteg tényező van a piacon, ami egy tárgy árát meghatározzák, ilyen a ritkasága, előtörténete, mennyire népszerű a tárgy a játékosok által, milyen osztályhoz, karakterre való a tárgy stb.
Ez tényleg csak egy bevezető, de azt gondolom, hogy a tárgyak tulajdonságainak számossága és összetétele megfelelő kombinatorikai kihívással rendelkezik egy szakdolgozati témához.

2.2 Online piacterek fogalma

Az online piacterek olyan webalapú platformok, amelyek lehetővé teszik eladók és vásárlók számára, hogy különféle termékeket vagy szolgáltatásokat kínáljanak. Ezek a piacterek közvetítő szerepet töltenek be, biztosítva a piaci szereplők közti kapcsolatokat, valamint a felhasználótárgyak eladási szándékának népszerűsítését. Az online piacterek legfőbb jellemzői közé tartozik a széles kínálat és a sokféle értékesítési forma, amely magában foglalhatja a közvetlen ajánlattételt, aukciókat vagy licitálást. A digitális piacterek előnye, hogy a fizikai korlátokat áthidalva globális közönséget érnek el. A Valve Steam platformja (vö. 2.3 fejezet) kifejezetten videojátékokhoz kapcsolódó virtuális tárgyak kereskedelmét teszi lehetővé. Az online piacterek fejlődése szorosan összefügg az internetes technológiák fejlődésével, a

felhasználói igények változásával, valamint a digitális gazdaság növekedésével. Egyre fontosabb szerep jut az API-alapú integrációknak, amelyek lehetővé teszik, hogy különböző platformok és szolgáltatások összehangoltan működjenek, ezzel egyszerűsítve és bővítve a piaci lehetőségeket (pl. egyedi szoftverek készítése, amelyek felhasználják az API integrációkat – automata csere-végrehajtó botok).

2.3 A steam platform

„*A Steam a Valve Corporation által fejlesztett digitális tartalomterjesztő és -kezelő rendszer, amely 2003. szeptember 12-én jelent meg hivatalosan. A platform eredeti célja a Valve játékainak automatikus frissítése volt, különösen a Counter-Strike esetében, ahol a manuális patch-ek letöltése és telepítése napokra bénította meg a játékot. A Steam fejlesztése 2002-ben kezdődött "Grid" és "Gazelle" munkacímeken, és 2003-ban bétatesztelési fázisba lépett. A platform áttörést 2004 novemberében érte el, amikor a Half-Life 2 megjelenésekor a Steam lett az első magas profilú játék, amely digitális formában is elérhető volt. Ez a lépés jelentős visszhangot váltott ki, mivel a játék retail példányai is Steam aktivációt igényeltek, ami kezdetben szerver túlterheléshez és felhasználói elégedetlenséghez vezetett. A platform azonban folyamatosan fejlődött, és mára a PC játékok legnagyobb digitális terjesztési platformjává vált, 2013-ban a piac 75%-át uralva.*” (Wikipédia, Steam)

2.4 A képzés tantárgyainak és a szakdolgozati témának a kapcsolata

Jelen fejezetben felsorolásra kerültek azok a tárgyak, amelyek a képzés során elméleti/gyakorlati tudást adtak.

2.4.1 Programozás

A programozás tantárgyak (I-III) során lehetőségem volt újra elsajátítani és biztosabb alapra helyezni a C# programozási nyelv és az ASP.NET Core keretrendszer ismereteimet. A tantárgy során gyakoroltam az objektumorientált programozási elveket, amelyek kulcsfontosságúak voltak a backend architektúra kialakításakor (lásd 3.5.3 fejezet).

2.4.2 Adatbázisok

Az adatbázis tantárgyak során elsajátítottam a relációs adatbázisok tervezésének és kezelésének alapjait, beleértve az SQL nyelvet, a normalizálást, az indexelést és a lekérdezés-optimalizálást. A szakdolgozatban implementált adatbázis-struktúra - felhasználók, csere-igények, tárgyak, kommentek közötti relációk (lásd 3.5.3 fejezet) - tükrözi az adatbázis-tervezés során tanult

elveket, különös tekintettel a külső kulcsokra, az egy-sok és sok-sok kapcsolatokra, valamint a lekérdezési teljesítmény optimalizálására.

2.4.3 Adatszerkezetek és algoritmusok

A keresési algoritmus implementálása során (vö. 3.5.3.6 fejezet - TradeService SearchTradesAsync metódus) és lista műveletek (LINQ Where, Select) kerültek alkalmazásra. A pagination algoritmus (Skip/Take) és a szűrési logika összetett adatszerkezeteken alapul.

2.4.4 Szoftverarchitektúrák

A rétegelt (layered) architektúra alkalmazása: Controller → Service → Repository → Database (vö. 3.5.3 fejezet). A mikroszerviz architektúra megértése lehetővé tette a Steam API parser külön Node.js szolgáltatásban való elkülönítését (vö. 3.5.4 fejezet), amely REST API-n keresztül kommunikál a fő alkalmazással.

2.4.5 Hálózatok és számítógép architektúrák

A HTTP protokoll működésének ismerete szükséges volt a RESTful API tervezéséhez (GET, POST, PUT, DELETE metódusok) (vö. 3.5.3.6 fejezet). A CORS konfigurálása, SSL/TLS használata.

2.4.6 Informatikai védelem és biztonság

JWT (JSON Web Token) alapú authentikáció implementálása (vö. 3.3.4 fejezet), a Steam OpenID protokoll (vö. 3.3.1 fejezet) biztonsági szemléleteinek megértése elengedhetetlen volt a felhasználói bejelentkezéshez.

2.4.7 Felhasználói interfészek és vizualizáció

Az PrimeNG komponenskönyvtár használata, reszponzív dizájn kialakítása (vö. 3.5.2 fejezet). A felhasználói élmény optimalizálása (töltési állapotok, hibakezelés, felugró ablakok jelzése) közvetlenül erre a tantárgyra épít.

2.4.8 Szoftvertesztelés

A teljesítménytesztelés során alkalmazott módszertan, warm-state és cold-state tesztelés (vö. 3.6.3 fejezet) és a különböző tesztelési szintek, kliensoldali (vö. 3.6.1 fejezet) és szerveroldali (vö. 3.6.2 fejezet) unit tesztek megértése ezen tantárgy ismeretköréből származik.

2.4.9 Rendszertervezés

A logikai rendszerterv elkészítése (vö. 3.4 fejezet), Use Case diagramok, adatbázis ER diagramok. A követelményelemzés és a rendszerspecifikáció elkészítése során ez a tantárgy adta a módszertani keretet.

2.4.10 Rendszermodellezés

Jelen tárgy ismereti mutatják be azt, hogy hogyan érdemes a legegyszerűbb és a legbeszédesebb módon modellezni az elkészítendő fejlesztést (vö. 3.4.1 fejezet és 3.4.2 fejezet).

2.4.11 Programozási alapelvek és módszertanok

Git verziókezelés használata, pull request-ek, commit message konvenciók. A SOLID elvek (Single Responsibility, Dependency Injection) (vö. 3.5.2 fejezet) gyakorlati alkalmazása a service osztályok tervezésénél.

2.4.12 Matematikai alapok

Statisztikai alapfogalmak (átlag, medián) használata a teljesítményteszt eredmények értékelésénél (vö. 3.6.3 fejezet).

2.4.13 Operációs rendszerek

Jövőbeli üzemeltetési lépésekhez hasznos elméleti/gyakorlati tudásra tehettem szert. Ezért is választottam az üzemeltetésnél a Linux alapú hosting környezet (Ubuntu Server) konfigurálását VPS-en (vö. 3.2.2 fejezet).

2.4.14 Vállalati gazdaságtan

Költség-haszon elemzés elkészítése: fejlesztési költségek (munkaóra × órabér), üzemeltetési költségek, megtérülési idő számítása (vö. 1.5.2 fejezet).

2.4.15 Vezetési és vállalkozási ismeretek

Projektmenedzsment alapok alkalmazása: időbecslés, erőforrás-elosztás (400 munkaóra elosztása), prioritási döntések. A célcsoport elemzése és a piaci igények felmérése (vö. 1.4 fejezet és 1.1 fejezet).

2.4.16 Szoftverüzemeltetés

Deployolási stratégia feltérképezése (vö. 3.2.2 fejezet), illetve logolási (vö. 3.5.3.8. fejezet) beállítások elvégzésre szerver oldalon, amely elősegíti a felmerülő hibák nyomonkövetését, illetve esetleges lassúsági problémák feloldása is egyszerűbbé válik.

2.4.17 Európai civilizáció és identitás

A digitális gazdaság társadalmi hatásainak megértése. A virtuális tárgyak kulturális jelentőségének felismerése (vö. 1.1 fejezet) és a játékos közösségek szerepe az európai digitális kultúrában.

2.4.18 Komplex társadalomtudományi ismeretek

A célcsoport demográfiai elemzése (18-35 éves férfi játékosok) (vö. 1.4 fejezet).

2.4.19 Emberi viselkedés és kommunikáció

UX design során a felhasználói pszichológia figyelembevétele (játékosított rendszer: tapasztalati pontrendszer (vö. 3.5.2.4 fejezet), bump mechanizmus). A komment funkció kommunikációs aspektusai és a platform közösségépítő szerepe (vö. 3.5.2.11 fejezet).

2.4.20 A jog szerepe a modern társadalmakban

A szakdolgozat során fejlesztett webalapú tárgycsere-platform üzemeltetése több jogi aspektust (pl. GDPR adatvédelmi megfelelés, sütik szabályozás, felhasználói adatok tárolásának jogi keretei, szellemi tulajdonjogok tisztázás) is érint, amelyek ugyan nem kerültek részletes kidolgozásra a dolgozatban (vö. 1.6.2 fejezet), de tudatosult bennem azok fontossága a valós üzemeltetés során.

2.4.21 Innovatív információs és kommunikációs technológiák

Modern web technológiák (pl. Angular 18 (vö. 3.1.1.1 fejezet), ASP.NET Core 9 (vö. 3.1.1.2 fejezet)) alkalmazása. Progressive Web App (PWA) lehetőségek alkalmazása a szakdolgozat során.

2.4.22 Mesterséges intelligencia az IT biztonság területén

AI asszisztens (Perplexity AI) használata kódoptimalizálásra, hibakeresésre és szakirodalmi kutatásra (vö. 3.7 fejezet).

2.4.23 IT biztonsági fejlesztések minőség és projektmenedzsmentje

Biztonsági követelmények priorizálása (authentikáció (vö. 3.3.4 fejezet) > authorizáció > adatvédelem).

2.4.24 Tudásmenedzsment az IT biztonság területén

Dokumentáció készítése (jelen szakdolgozat, API dokumentáció Swagger-rel) (vö. 3.5.3 fejezet).

2.4.25 Kultúra, sport, munkahelyi jólét

Work-life balance fenntartása a 400 órás fejlesztési munka során. Időgazdálkodás és stresszkezelés a határidők betartása érdekében.

2.4.26 Az elektronika fizikai alapjai

Bár kevésbé direkten kapcsolódik, a szerverek fizikai működésének (CPU, RAM, SSD) megértése segített a teljesítménytesztek értékelésében és az infrastruktúra kiválasztásában (VPS specifikációk) (vö. 1.5.2 fejezet).

2.4.27 Elektronikus áramkörök

Hasonlóan az előzőhöz, az alapvető hardver-ismeretek hozzájárultak a szerverkonfigurációs döntésekhez (RAM mennyiség, CPU magok száma a konkurens kérések kezelésére) (vö. 1.5.2 fejezet).

2.4.28 Mentori óra

A konzulens professzorral folytatott megbeszélések során kapott visszajelzések alakították a dolgozat struktúráját és tartalmát. A szakmai tanácsok beépítésre kerültek.

3 A saját fejlesztés bemutatása

Ebben a fejezetben részletesen bemutatom a Team Fortress 2 kereskedési platform tervezési és megvalósítási folyamatát. A fejezet célja, hogy átfogó képet adjon a rendszer műszaki megvalósításáról, a választott technológiai megoldásokról és azok indoklásáról. A platform három fő architekturális rétegre épül: a felhasználói felületet kezelő frontend réteg, az üzleti logikát és adatkezelést végző backend réteg, valamint egy specializált microservice komponens, amely a Team Fortress 2 tárgyak formátum-átalakítását végzi. A technológiai stack

kiválasztása során figyelembe vettem a közösségi támogatottságot, a dokumentáció minőségét, a személyes tapasztalatomat és a projekt-specifikus követelményeket.

3.1 Fizikai rendszerterv

A fizikai rendszertervben bemutatom a webalkalmazás fejlesztéséhez választott technológiai stacket, valamint az alternatívákat és a döntések indoklását. A technológiaválasztás során az elsődleges szempontok a skálázhatóság, a közösségi támogatás, a fejlesztési sebesség és a személyes tapasztalataim voltak.

3.1.1 Program specifikáció

Jelen fejezet mutatja be, hogy milyen technológia döntéseket hoztam meg, illetve döntéseimet indoklom.

3.1.1.1 Kliensoldali specifikáció

Választott kliensoldali megoldás: Angular

„*Az angular egy Typescript-alapú ingyenes és nyílt forráskodú egyoldala webalkalmazás-keretrendszer. A Google, valamint magánszemélyek és vállalatok közössége fejlesztette ki. Az Angular az AngularJs csapat által készített nyelv újraírásának az eredménye.*” (Angular Documentation, 2025)

Választás indoklása:

- **Typescript integráció:** Típusbiztonságot nyújt, amely csökkenti a futásidejű hibák számát és átláthatóbb, olvashatóbb kódot eredményez
- **Komponens-alapú architektúra (11. ábra):** Újrafelhasználható UI elemek könnyű fejlesztése
- **Beépített szolgáltatások:** Form kezelés, http kliens, routing, dependency injection
- **Személyes tapasztalat:** 4+ év Angular fejlesztői tapasztalat (vö. 1.3.1 fejezet)
- **RxJS támogatás:** Aszinkron adatkezelés reaktív programozás paradigmával

Megvizsgált alternatívák:

Next.js (react alapú)

A NextJS egy nyílt forráskodú webfejlesztési keretrendszer, amelyet a Vercel magáncég hozott lére, amely React-alapú webes alkalmazásokat kínál szerveroldali megjelenítéssel és statikus

megjelenítéssel. Előnye a gyors kezdeti betöltés és SEO-optimalizáció, hátrányai között szerepel, hogy új keretrendszer tanulását igényelné.

Vue.js

Progresszív JavaScript keretrendszer, amely egyszerű tanulási görbével rendelkezik. Nem választottam, mert kevesebb tapasztalatom van benne, illetve számomra a kód olvashatósága is nehezebb.

3.1.1.2 Szerveroldali specifikáció

Választott szerveroldali megoldás: C# és ASP.NET Core

„*A C# a Microsoft által a .NET keretrendszer részeként kifejlesztett objektumorientált programozási nyelv*" (Microsoft, 2024). Az ASP.NET Core egy cross-platform, nagy teljesítményű keretrendszer modern webalkalmazások építéséhez.

Választás indoklása:

- **Erős típusosság:** Biztonságos és karbantartható kód
- **Entity Framework Core:** ORM támogatás egyszerűsíti az adatbázis műveleteket
- **Aszinkron programozás:** async/await kulcsszavakkal hatékony I/O műveletek
- **JWT támogatás:** Beépített authentikációs middleware-ek

Megvizsgált alternatívák:

Node.js:

Nyílt forráskodú, szerveroldali Javascript futtatókörnyezet, amely a Chrome V8 javascript motorjára épül. Lehetővé teszi, hogy javascript kódokat a szerveren is futtassunk, így egységes nyelvet használhatunk a frontend és backend fejlesztéséhez is. Bár a Node.js-t sikeresen alkalmaztam a projekt microservice részében (lásd 3.5.4 fejezet), a fő backend réteghez a C#-ot választottam. A döntés oka, hogy szerettem volna egy erősen típusos, vállalati környezetben kipróbált nyelvet is elsajátítani, amely mélyebb objektumorientált programozási ismereteket igényel. Emellett az egyetemi tanulmányaim során a C# és a .NET ökoszisztéma részletesebb tanulmányozására is lehetőségem volt, így a szakdolgozat kiváló alkalom volt ezeknek a készségeknek a gyakorlati alkalmazására és elmélyítésére. (vö. 3.5.3 fejezet)

NestJS:

Node.js-alapú progresszív keretrendszer TypeScript támogatással, MIT licenc alatt. Az Angular-hez hasonló architektúrát követ, de a C# ökoszisztéma gazdagabb volt a projekthez.

Java:

Általános célú, objektumorientált nyelv. Nem választottam, mert a .NET ökoszisztéma modernebb eszközöket (pl. LINQ, async/await) kínál.

Választott adatbázis technológia megoldás: PosgreSQL

„*A PostgreSQL egy nyílt forráskódú, relációs adatbázis-kezelő rendszer, amelyet közösségi alapon fejlesztenek.*" (PostgreSQL Global Development Group, 2024) Ismert stabilitásáról, ACID-kompatibilitásáról és fejlett funkcióiról (JSON támogatás, full-text search).

Választás indoklása:

- **Ingyenes és nyílt forráskódú:** Nincs licencdíj
- **JSON támogatás:** Hibrid adatmodell lehetősége
- **Közösségi támogatás:** Nagy fejlesztői közösség
- **Entity Framework Core integráció:** Kiváló .NET támogatás

Megvizsgált alternatívák:

Microsoft SQL Server:

A microsoft SQL Server egy szabadalmaztatott relációs adabázis-kezelő rendszer, amelyet a Microsoft fejlesztett ki úgynevezett Structured Query Language használatával. Hátránya a licencdíj, ezért nem választottam.

MySQL:

Egy nyílt forráskódú relációs adatbázis-kezelő rendszer, amelyet széles körben használnak webes alkalmazásokhoz. A MySQL inkább kisebb és közepes méretű alkalmazásoknál, illetve gyors fejlesztéseknél kedvelt. Nem választottam, mert a PostgreSQL fejlettebb funkciókat (pl. JSONB, window functions) biztosít.

3.1.2 Technológiai stack összefoglalása

Készítettem egy összefoglalót a technológiai stack-ek összefoglalásáról (5. táblázat).

Réteg	**Technológia**	**Verzió**	**Indoklás**
Frontend	Angular	19.2.0	Komponens-alapú, Typescript, személyes tapasztalat
Backend	ASP.NET Core	9.0	JWT támogatás, EF Core
Adatbázis	PostgreSQL		Nyílt forráskód, JSON támogatás
ORM	Entity Framework Core	9.0.9	Code-first megközelítés, LINQ támogatás
Auth	JWT + Steam OpenID	-	Külső authentikáció

5. táblázat - A rendszer technológiai architektúrája és fő komponensei – Forrás: Saját összegzés

3.2 Üzemelési terv

Az üzemelési terv célja, hogy bemutassa a webalkalmazás éles környezetben történő futtatásának technikai követelményeit, a telepítési folyamatot és az üzemeltetési feladatokat. A fejezet fontos része a szakdolgozatnak, mivel megmutatja, hogy a fejlesztett rendszer valós környezetben is működőképes.

3.2.1 Hardver- és szoftverkövetelmények

Szerver oldali követelmények:

Az ASP.NET Core backend futtatásához szükséges környezet:

- **Operációs rendszer:** Linux (Ubuntu 22.04 LTS vagy újabb) vagy Windows Server 2019+

- **Processzor:** Minimum 2 CPU mag (ajánlott 4 mag)
- **Memória:** Minimum 4 GB RAM (ajánlott 8 GB)
- **Tárhely:** 50 GB SSD (adatbázis méretétől függően)
- **Hálózat:** Minimum 100 Mbps internet kapcsolat

Adatbázis szerver követelményei:

PostgreSQL adatbázishoz szükséges erőforrások:

- **Verzió:** PostgreSQL 15.x vagy újabb
- **Memória:** Minimum 2 GB dedikált RAM
- **Tárhely:** Minimum 20 GB (növekedési lehetőséggel)

Node.js mikroszerviz követelményei:

A tárgyformátum-átalakító mikroszervizhez (lásd 3.5.4 fejezet):

- **Node.js verzió:** 18.x LTS vagy újabb
- **Memória:** Minimum 1 GB RAM
- **Processzor:** 1-2 CPU mag

Kliens oldali követelmények:

Az Angular frontend használatához:

- **Böngészők:** Chrome 100+, Firefox 90+, Edge 100+, Safari 15+
- **JavaScript:** Engedélyezett és aktív
- **Internetkapcsolat:** Minimum 5 Mbps (ajánlott 10 Mbps)

3.2.2 Telepítés és beüzemelés

A rendszer telepítése három fő komponens egyidejű konfigurálását igényli: a backend API szerver, az adatbázis, valamint a Node.js microservice üzembe helyezését. A telepítési folyamat alapvető lépései az alábbiak szerint zajlanak.

Backend API telepítés

Az ASP.NET Core (vö. 3.5.3 fejezet) alkalmazás telepítéséhez először a forráskódot a VPS szerverre kell másolni Git repository klónozásával. A „dotnet publish” parancs használatával a projektet kiadásra kész build-állapotba kell helyezni, majd a „dotnet run” vagy systemd service konfigurációval éles módban futtatható. Az alkalmazás konfigurációs fájljában

(appsettings.json) szükséges beállítani a PostgreSQL connection string-et, JWT secret kulcsot, valamint a Steam API kulcsot a külső authentikáció működéséhez.

Adatbázis inicializálás

A PostgreSQL adatbázis üzembe helyezéséhez a szerveren telepített PostgreSQL szerveren létre kell hozni egy új adatbázist dedikált felhasználóval. Az Entity Framework Core migration-ök (dotnet ef database update parancs) futtatásával az adatbázis séma automatikusan létrejön a modell definíciók alapján.

Node.js mikroszerviz telepítés

A Node.js alapú tárgyformátum-átalakító mikroszerviz telepítéséhez (vö. 3.5.4 fejezet) a forrás könyvtárat szintén a szerverre kell másolni, majd az „npm install" paranccsal telepíteni a függőségeket. A szolgáltatás „npm start" segítségével futtatható háttérfolyamatként. A microservice alapértelmezetten a 3000-es porton figyel, amelyet a backend API-ból lehet elérni HTTP kérésekkel.

Frontend telepítés

Az Angular alkalmazás (vö. 3.5.2 fejezet) build-eléséhez (ng build --configuration production) a projekt fordított JavaScript bundle-ként kerül előállításra. Az így kapott dist/ könyvtár tartalmát egy webszerverre (pl. Nginx vagy Apache) kell feltölteni, amely statikus fájlokat szolgál ki. Fontos, hogy a webszerver konfigurálja az Angular routing-ot támogató URL átirányítást, hogy a SPA navigáció megfelelően működjön.

3.3 IT biztonsági terv

3.3.1 OpenId

„*Az OpenId egy nyílt, decentralizált, ingyenes internetes szolgáltatás, ami lehetővé teszi a felhasználók számára, hogy egyetlen digitális identitással lépjenek be különböző oldalakra. Maga a szolgáltatás az OAuth 2.0 keretrendszer az (IETF RFC 6749 and 6750)*" (datatracker, 2025) specifikációi alapján. Az OpenId szolgáltatás azért hasznos, hogy már egy meglévő felületen, amelyen bejelentkezik a felhasználó (steam), továbbítja a bejelentkezett, authentikált adatokat a szerverről, így mellőzve az új bejelentkezést.

1. A végfelhasználó böngészőn keresztül **navigál egy weboldalra vagy webes alkalmazásba .**
2. A végfelhasználó **a Bejelentkezés gombra kattint,** és beírja a felhasználónevét és jelszavát.
3. Az RP (kliens) **kérést küld** az OpenID-szolgáltatónak (OP).
4. Az OP **hitelesíti a felhasználót** és megszerzi az engedélyt.
5. Az OP **egy azonosító tokennel** és általában egy **hozzáférési tokennel** válaszol .
6. Az RP **küldhet egy hozzáférési tokent tartalmazó kérést** a felhasználói eszköznek.
7. A UserInfo végpont a végfelhasználóval kapcsolatos **jogcímeket ad vissza .**

1. ábra - OpenID működésének ábrázolása – Forrás: openid.net

3.3.2 Third party login – SteamLogin

A steam mint tartalomtovábbító és -kezelő rendszer (vö. 2.3 fejezet) ezek a tevékenységek mellett biztosít OpenId szolgáltatói lehetőséget is (1. ábra). Így a szakdolgozatban érintett platform használatához nem szükséges úgymond fiókkezelést, szerepkezelést biztosítani a felhasználók számára, hiszen maga az authentikáció és a bejelentkezés a steam által biztosított oldalon fog megtörténni. A bejelentkezés után már csak a felhasználó bejelentkezéséhez társított, generált azonosítót szükséges tárolni és később felhasználni azt lásd az 2. ábra és 3. ábra. A Valve vállalat a szolgáltatás használatáért cserébe csak annyit kér, hogy az ő általuk tervezett bejelentkezési nyomógomb kerüljön felhasználásra:

2. ábra - Steam bejelentkező gomb UI elem - verzió 1 – Forrás: Steam Web API Documentation

3. ábra - Steam bejelentkező gomb UI elem - verzió 2 – Forrás: Steam Web API Documentation

3.3.3 Steam API

A Steam vállalat a fentebb felsorolt szolgáltatások mellett, nyílt API végpontokat is biztosít a fejlesztők számára. Ennek használatához már egy meglévő steam fiókkal kell rendelkeznie a

fejlesztőnek, amely segítségével megigényelheti a saját, egyedi API kulcs azonosítóját. Ennek az azonosító használatával kérdezhető le különböző végpontokon, különböző adatok. Az adatok az alábbi struktúrában érkezhetnek meg a válaszüzenetekben:

- json - The output will be returned in the JSON format
- xml - Output is returned as an XML document
- vdf - Output is returned as a VDF file.

A helyes működéshez és az egyértelmű használathoz a Steam biztosít egy fejlesztői dokumentációt, amelyben példa üzenetekkel (küldendő és érkező) demonstrálja a nyílt, API végpontok működését.
A dokumentáció az alábbi linken érhető el: **https://steamcommunity.com/dev**

3.3.4 JWT Bearer hitelesítés

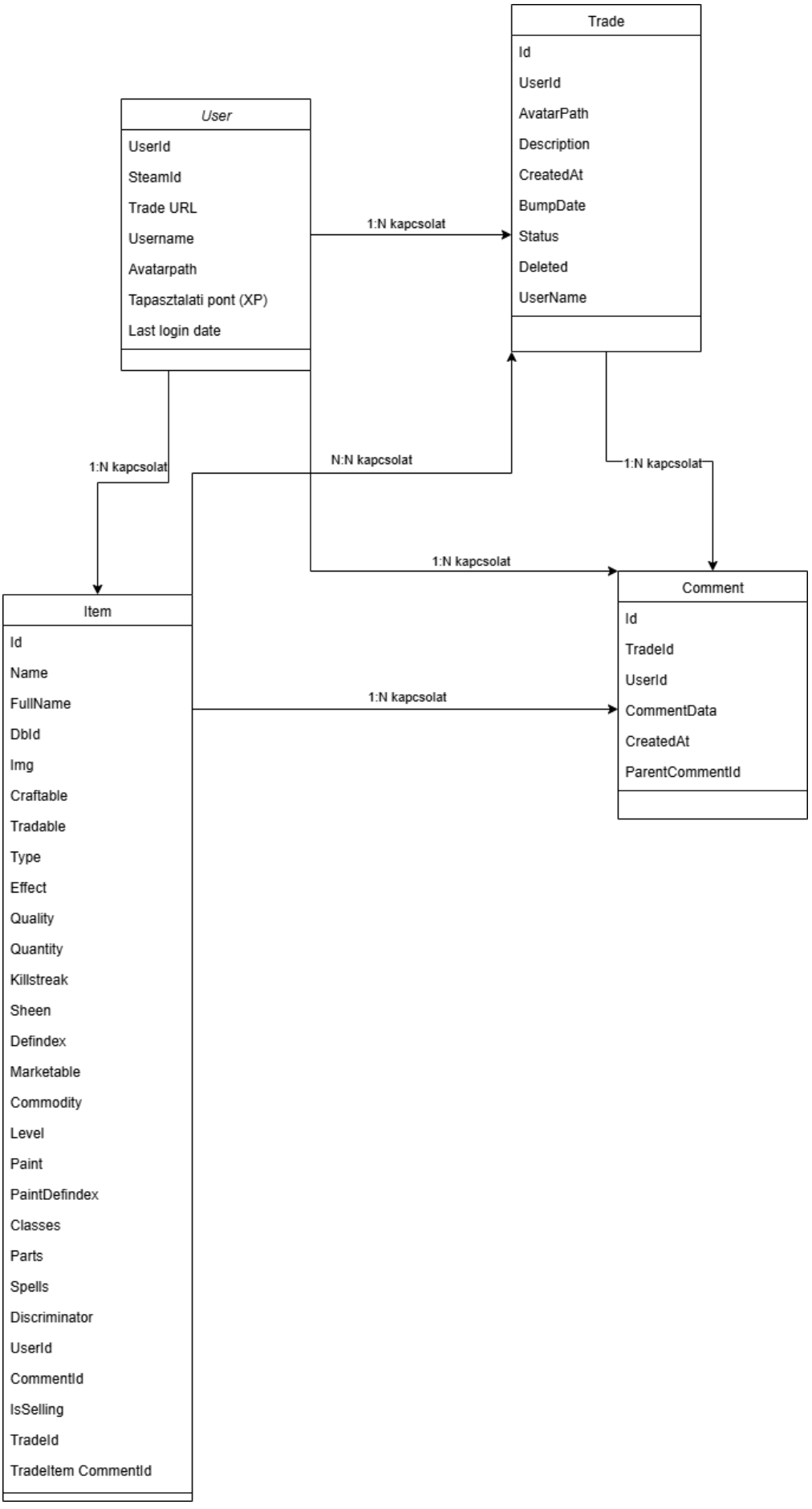
„*A JSON Web Token (JWT) egy nyílt szabvány (RFC 7519)*" (datatracker, 2025) amely egy kompakt, URL-ben is továbbítható formátumban tárol hitelesítési és jogosultsági adatokat. A JWT lehetővé teszi a szerver és a kliens közötti biztonságos információcserét, ahol a token aláírással védett, így a hitelessége ellenőrizhető.

3.4 Logikai rendszerterv

A logikai rendszertervben kifejtem, hogy a webalkalmazásom miként terveztem meg, illetve kifejtem azt is, hogy miért az alábbi architektúrákat választottam alkalmazásom elkészítéséhez. A rendszer fő komponenseit az ismertetéshez a legalkalmasabb, folyamatábrákat, illusztrációkat használom.

3.4.1 Adatstruktúrák logikai szinten - Entitások, attribútumok, kapcsolatok

Készítettem egy osztálydiagrammot, amely bemutatja, hogy a webalkalmazás alap entitásai milyen kapcsolatban állnak egymással. Az osztálydiagram segítségével gyorsan és egyszerűen szemléltethető akár összetettebb, bonyolultabb kapcsolat is.

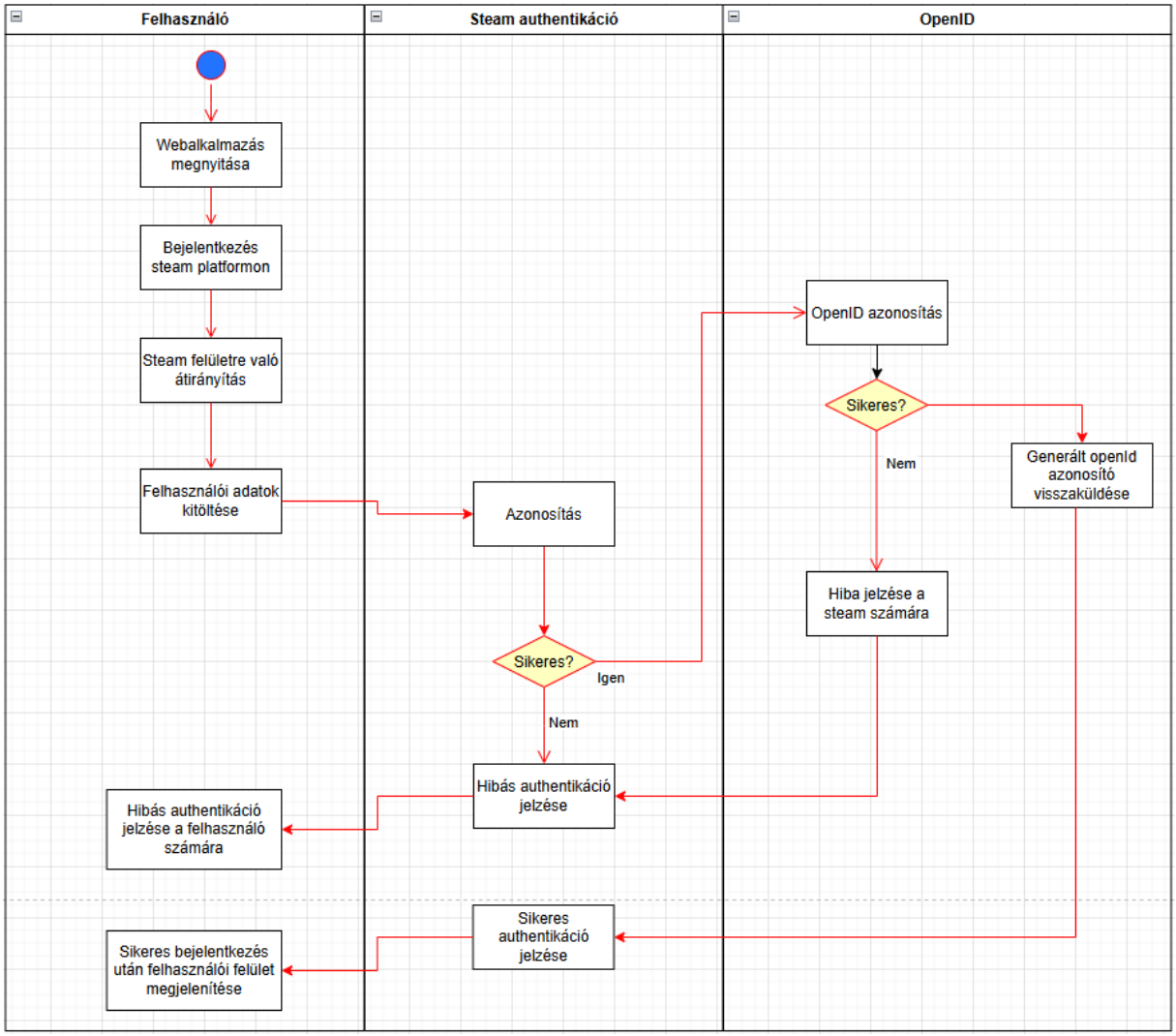

Trade
Id
UserId
AvatarPath
Description
CreatedAt
BumpDate
Status
Deleted
UserName
User
UserId
SteamId
Trade URL
Username
Avatarpath
Tapasztalati pont (XP)
Last login date
1:N kapcsolat
1:N kapcsolat
N:N kapcsolat
1:N kapcsolat
1:N kapcsolat
Item
Id
Name
FullName
DbId
Img
Craftable
Tradable
Type
Effect
Quality
Quantity
Killstreak
Sheen
Defindex
Marketable
Commodity
Level
Paint
PaintDefindex
Classes
Parts
Spells
Discriminator
UserId
CommentId
IsSelling
TradeId
TradeItem CommentId
Comment
Id
TradeId
UserId
CommentData
CreatedAt
ParentCommentId

4. ábra - Osztálydiagram - fő entitások közötti kapcsolatok – Forrás: Saját képernyőfotó

Az entitások közötti kapcsolatok kulcsfontosságúak egy megbízható, skálázható adatbázis létrehozásában. Silberschatz és munkatársai (2010) hangsúlyozzák, hogy „*az Entitiy-Relationship (ER) modell alapvető eszköz az adatbázis-tervezésben, mivel lehetővé teszik az egyszerű vizualizációt. Az ER diagramból egyszerűen leolvashatók a relációs séma táblái, attribútumai és az idegen kulcsok, amelyek biztosítják a robusztus működést.*" (Silberchatz et al, 2010)

3.4.2 Felhasználói modul

A felhasználói modulhoz készített folyamatábra ismerteti a bejelentkezés – authentikációs folyamatot a webalkalmazás és a steam platformja között.



5. ábra - Aktivitás diagram – Authentikáció – Forrás: Saját képernyőfotó

A webalkalmazás regisztrációt és bejelentkezést nem kezel, nem tárol authentikációs adatokat. A steam által biztosított openId provider lehetőséget kihasználva csak a sikeres authentikáció után a webalkalmazás szerver oldalon JWT token-t generál (vö. 3.3.4 fejezet), amely a kliens böngészőjében eltárolásra kerül, sikeres authentikáció esetén.

Felhasználói fiók kezelése

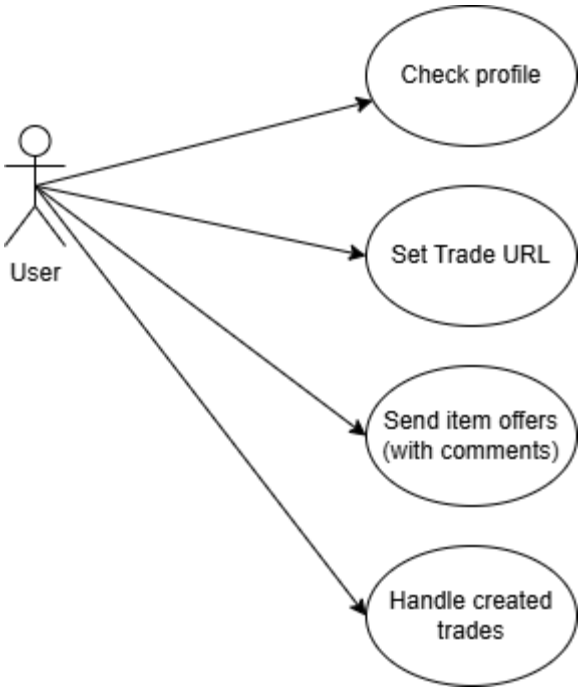
Az alkalmazásba való bejelentkezés után a felhasználó számára elérhetővé válik a **profilom** menüpont (vö. 3.5.2.2 fejezet), ahol személyre tudja szabni a beállításait, illetve a profiljáról készült statisztikákat tudja megtekinteni. A statisztikák a publikus ajánlattételek számáról, aukciós ajánlatok/tranzakciók számából és a bejelentkezés számáról készül.

Trade URL beállítás

A rendszer használatához szükséges a Steam-en (vö. 2.3 fejezet) keresztüli elkérhető kereskedési link (trade URL) megadása. A felhasználó ezt a saját profiljában adhatja meg, amelyet később bármikor módosíthat. A beállított URL ellenőrzésen megy keresztül (formai érvényesség), mielőtt eltárolásra kerülne az adatbázisban.

Tapasztalati pont (XP) és szintlépési rendszer

Az alkalmazás tartalmaz egy játékosított tapasztalati pontrendszert, amely a felhasználók aktivitását díjazza. A felhasználó különböző interakciókért (pl. ajánlattétel, trade URL beállítása) pontokat (XP-t) szerezhet (vö. 3.5.3.5 fejezet), amely segítségével szintet léphet. A szintlépés fő célja, hogy fenntartsa a felhasználó számára a motivációt.



6. ábra - Felhasználói fiók funkciók – Forrás: Saját képernyőfotó

3.4.3 Csere-igény modul

Csere-igény létrehozása

A felhasználó csere-igényt tud meghirdetni, amelynek a módja a következő:

1. A felhasználó kitudja választani az Általa birtokolt (steam fiókban található) tárgyak listájából az eladni, meghirdetni kívánt tárgyakat.
2. Ezután a felhasználó megadhatja, milyen típusú vagy konkrét tárgyakat keres cserébe. Ezt egy szűrhető listán keresztül teheti meg:
    a. Tárgy típusa (sapka, unusual, strange stb.)
    b. Színezés
    c. Ritkaság
3. Opcionálisan szöveges megjegyzést is fűzhet a cseréhez (vö. 3.5.2.11 fejezet), például: „Csak hasonló értékű festéket keresek”, „Unusual érdekel” stb.
4. Végül a felhasználó közzéteszi a csere-igényt, amely ezután megjelenik a nyilvános cserepiacon, ahol más felhasználók ajánlatot tehetnek rá (vö. 3.5.2.8 fejezet).

Csere-igény módosítása

A felhasználó a már korábban létrehozott csere-igényeit tudja kezelni egy külön menüponton belül (vö. 3.5.2.9 fejezet). A bejegyzéseket a következő módon állíthatja át:

1. Teljes csere-igény bejegyzés törlése (a létrehozott bejegyzés törlésre kerül, nem lesz többé publikus mások számára)
2. A publikált csere-igényben egy eladni kívánt tárgyat törölhetünk, státuszát eladott-á állíthatjuk, megvásárolni kívánt tárgyak módosítása, törlése, hozzáadása
3. A meghirdetett csere-igényre érkező ajánlatok módosítása, törlése
4. Csere-igényhez írt kommentek törlése, módosítása.

Csere-igény státusz állítása

A csere-igény státuszának állítása a felhasználó számára lehetőséget biztosít, hogy az általa korábban létrehozott csere-igények státuszát dinamikusan módosítsa anélkül, hogy azokat véglegesen törölné. Ez a funkcionalitás egy kapcsoló (toggle) mechanizmusként működik, amely lehetővé teszi a csere-igény "aktív" vagy "inaktív" állapotba helyezését. Ennek az az előnye, hogy a felhasználó ideiglenesen felfüggesztheti a csere-igény bejegyzéseket, például, ha átmenetileg nem szeretne új ajánlatokat kapni, de nem kívánja elveszíteni a csere-igény adatait.

Csere-igény törlése

A csere-igény törlése a felhasználó által létrehozott csere-igény bejegyzés eltávolítását jelenti a felületről. Ez a művelet végleges, ezért a rendszer biztosítja, hogy a törlés csak a bejelentkezett és az adott csere-igényhez jogosultsággal rendelkező felhasználó számára legyen elérhető. Az adatbázisból **nem** kerül törlésre a rekordja, csupán az úgynevezett „deletedFlag” mező kerül állításra, majd idővel automatikusan törlésre kerül az adatbázisból.

Publikus csere-igény piac böngészése

A felhasználóknak lehetőségük van az összes, nyilvánosan elérhető csere-igény bejegyzések között böngészni egy dedikált felületen (vö. 3.5.2.7 fejezet). A csere-igények oldalanként kerülnek listázásra, a megjelenített bejegyzések száma választható (10, 15 vagy 20 elem oldalanként).

Ajánlattétel egy csere-igényre

A felhasználónak lehetősége van ajánlatot tenni egy másik felhasználó által létrehozott, publikus, aktív hirdetésen belül. Az ajánlattétel során csak az ajánlattevő meglévő tárgyai közül

tud választani és a kommenteléshez hasonló módon, ajánlatot tehet (vö. 3.5.2.11 fejezet). Fontos kiemelni azt, hogy a webalkalmazás kizárólag egy interaktív platformként működik, amely a Steam-fiókokhoz kapcsolódó csere-igény hirdetések kezelését és népszerűsítését támogatja. A rendszer nem bonyolítja le ténylegesen a tárgyak cseréjét, nem kér hozzáférést a felhasználók tárgyaihoz, és nem tárol semmilyen virtuális tárgyat. A felhasználók közötti cserefolyamat teljes mértékben a Steam hivatalos platformjain keresztül történik (vö. 2.3 fejezet), a webalkalmazás csupán a hirdetések közzétételét, kereshetőségét és az ajánlattétel lehetőségét biztosítja.

Microservice komponens – tárgyformátum átalakítás / alapértelmezett tárgyak

A tf2-node alapú microservice egy dedikált Node.js szolgáltatás, amely kifejezetten a TF2 játék virtuális tárgyainak komplex feldolgozására lett tervezve (vö. 3.5.4 fejezet). Ennek oka, hogy a Steam API által szolgáltatott adatok rendkívül részletesek és összetettek, rengeteg attribútummal rendelkeznek. A microservice célja, hogy ezt az összerendelési, formázási, specifikus feldolgozást leválassza a fő backend rendszertől, így:

- Megtartja az ASP.NET Core backend tiszta architektúráját és könnyen kezelhető szolgáltatásait.
- A specifikus TF2 tárgylogikát egy külön komponens kezeli.
- Gyorsabbá és hatékonyabbá teszi az adatok lekérését, mivel a tf2-node könyvtárra épül, amely a TF2-specifikus tárgyak elemzését és konvertálását támogatja.
- Biztosítja a TF2 játékban szereplő tárgyak és stílusokat.

3.5 Implementáció

Az előző fejezetekben bemutatott tervezési döntések és architektúrális megoldások ebben a fejezetben kerülnek gyakorlati megvalósításra. Az implementáció részletezi a platform frontend és backend komponenseinek konkrét technikai megvalósítását, a választott technológiai stack elemeit.

3.5.1 Bevezető

Az implementáció fejezet célja bemutatni a szakdolgozat során fejlesztett webalapú tárgycsere-platform technikai megvalósítását. Itt mutatom be részletesen a használt technológiákat, fejlesztési környezeteket, valamint az alkalmazás működésének kulcsfontosságú elemeit. Az implementáció vizsgálata során kitérek a frontend és backend összetevőkre, a biztonsági

megoldásokra (vö. 3.3 fejezet), a külső rendszerekhez (Steam OpenID, Steam API) történő kapcsolódás módjára és a funkcionalitás gyakorlati megvalósítására. A fejezet bemutatja, milyen eszközökkel és módszerekkel valósult meg a projekt terve, továbbá rámutat a fejlesztési kihívásokra, azok megoldásaira, miközben útmutatóként szolgál a rendszer használatát illetően.

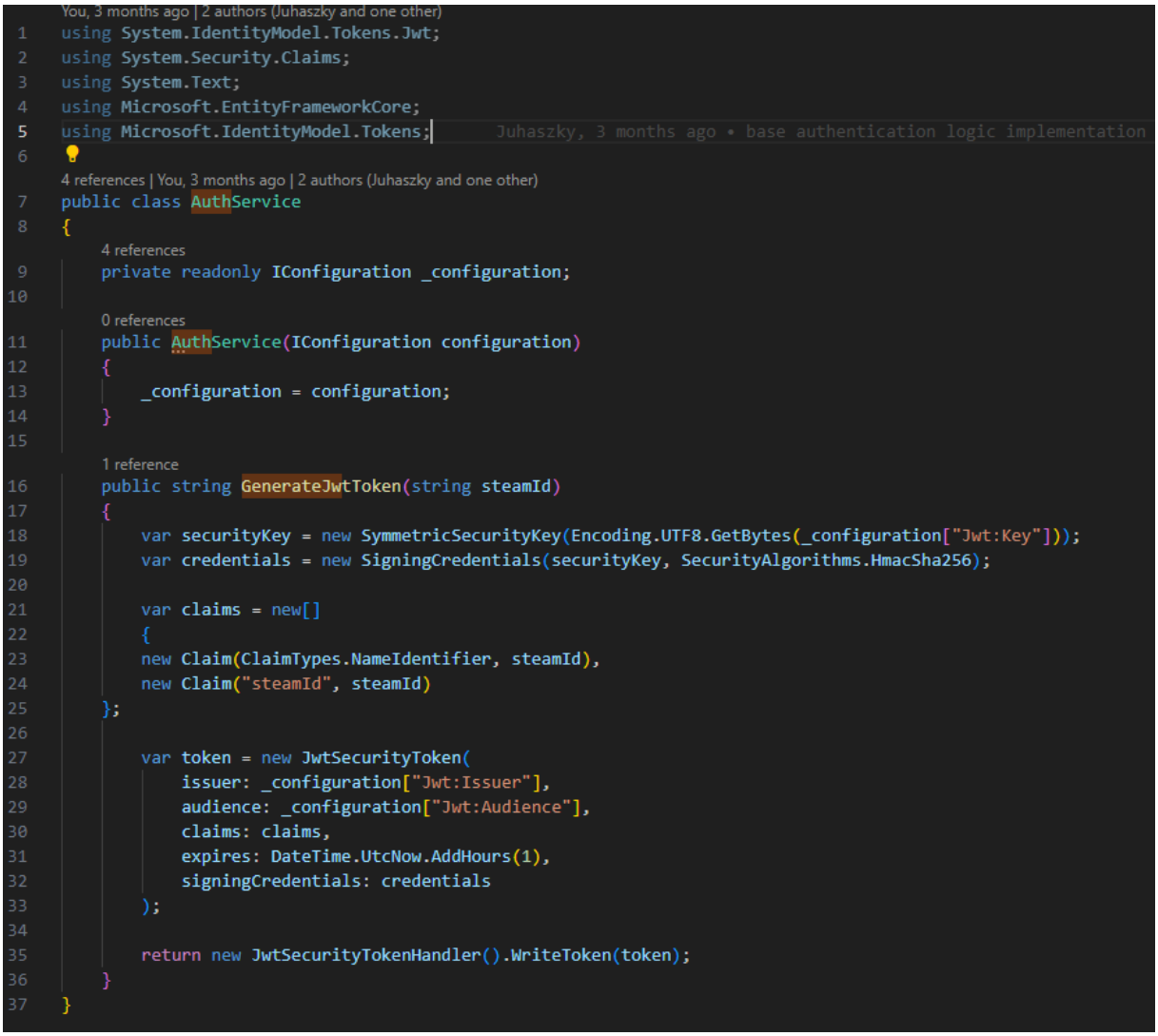
A fejlesztés során az alábbi főbb technológiákat alkalmaztam:

- **Frontend:** Angular keretrendszer (vö. 3.1.1 fejezet), amely modern, komponens alapú struktúrát ad a kliens oldali megvalósításhoz. Angular-t választottam a széles körű támogatás, stabilitás és beépített szolgáltatások (pl. routing, formok kezelése, HTTP kommunikáció) miatt.
- **Backend:** C# (vö. 3.1.1 fejezet), amely a támogatja a modern aszinkron programozást, és könnyű integrációt biztosít a JWT alapú authentikációhoz.
- **Entity Framework Core:** Az adatbázis műveletekhez (pl. select, insert) használt ORM eszköz, amely megkönnyíti az adatszerkezet kezelését és lekérdezéseket.
- **Steam API:** A Valve által szolgáltatott nyilvános API-k, melyek segítségével lekérdezhetők a játékosokhoz tartozó virtuális tárgyak adatai (vö. 3.3.3 fejezet).
- **Steam OpenID:** Autentikációs rendszer a felhasználók biztonságos beazonosításához és hitelesítéséhez, amely a Steam fiók hitelesítését végzi (vö. 3.3.1 fejezet).
- **JWT (JSON Web Token)**: A kliens és a szerver közti további hitelesített kommunikáció biztosítására szolgáló token alapú megoldás, amely megőrzi a bejelentkezett felhasználó állapotát (vö. 3.3.4 fejezet).
- **Fejlesztői eszközök:** Visual Studio Code, Git verziókezelés, Swagger az API tesztelésére.

Ez a technológiai környezet biztosítja a rendszer megbízhatóságát, skálázhatóságát és biztonságos működését.

3.5.1.1 Főbb elemek bemutatása

Az ASP.NET backend fogadja a felhasználó bejelentkezési kérelmét, majd továbbítja a Steam OpenID szolgáltatásra (vö. 3.3.1 fejezet). A Steam hitelesíti a felhasználót, majd a válasz alapján a backend létrehozza vagy frissíti a felhasználói adatot és generál egy JWT tokent. Az alábbi vázlat bemutatja egy JWT token létrehozását ASP.NET Core-ban:



```
You, 3 months ago | 2 authors (Juhaszky and one other)
using System.IdentityModel.Tokens.Jwt;
using System.Security.Claims;
using System.Text;
using Microsoft.EntityFrameworkCore;
using Microsoft.IdentityModel.Tokens;      Juhaszky, 3 months ago • base authentication logic implementation

4 references | You, 3 months ago | 2 authors (Juhaszky and one other)
public class AuthService
{
    4 references
    private readonly IConfiguration _configuration;

    0 references
    public AuthService(IConfiguration configuration)
    {
        _configuration = configuration;
    }

    1 reference
    public string GenerateJwtToken(string steamId)
    {
        var securityKey = new SymmetricSecurityKey(Encoding.UTF8.GetBytes(_configuration["Jwt:Key"]));
        var credentials = new SigningCredentials(securityKey, SecurityAlgorithms.HmacSha256);

        var claims = new[]
        {
        new Claim(ClaimTypes.NameIdentifier, steamId),
        new Claim("steamId", steamId)
    };

        var token = new JwtSecurityToken(
            issuer: _configuration["Jwt:Issuer"],
            audience: _configuration["Jwt:Audience"],
            claims: claims,
            expires: DateTime.UtcNow.AddHours(1),
            signingCredentials: credentials
        );

        return new JwtSecurityTokenHandler().WriteToken(token);
    }
}
```

7. ábra - Authentikációs szolgáltatás – kódrészlet – Forrás: Saját implementáció

3.5.1.2 Hitelesítés konfiguráció ASP.NET Core-ban

Az authentikáció hitelesítés beállítása Program.cs fájlban:

```
builder.Services.AddAuthentication(options =>      Juhaszky, 3 months ago • base authentication logic implementation
{
    options.DefaultScheme = "Cookies";
    options.DefaultChallengeScheme = "Steam";

})
.AddCookie("Cookies")
.AddSteam(options =>
{
    options.Events.OnAuthenticated = context =>
    {
        var url = context.Identity?.FindFirst(ClaimTypes.NameIdentifier)?.Value;
        var steamId = new Uri(url).Segments.Last();
        if (!string.IsNullOrEmpty(steamId))
        {

            context.Identity?.AddClaim(new Claim("steamId", steamId));
        }
        return Task.CompletedTask;
    };
}).AddJwtBearer(options =>
{
    var key = Encoding.UTF8.GetBytes(configuration["Jwt:Key"]);
    options.Events = new JwtBearerEvents
    {
        OnAuthenticationFailed = context =>
        {
            Console.WriteLine("Authentication failed: " + context.Exception.Message);
            return Task.CompletedTask;
        },
        OnTokenValidated = context =>
        {
            Console.WriteLine("Token validated successfully.");
            return Task.CompletedTask;
        },
        OnChallenge = context =>
        {
            Console.WriteLine("JWT challenge: " + context.Error + " - " + context.ErrorDescription);
            return Task.CompletedTask;
        }
    };
    options.TokenValidationParameters = new TokenValidationParameters
    {
        ValidateIssuer = true,
        ValidateAudience = true,
        ValidateLifetime = true,
        ValidateIssuerSigningKey = true,
        ValidIssuer = configuration["Jwt:Issuer"],
        ValidAudience = configuration["Jwt:Audience"],
        IssuerSigningKey = new SymmetricSecurityKey(key),
        ClockSkew = TimeSpan.Zero
    };
});
```

8. ábra - Authentikáció hitelesítés – kódrészlet – Forrás: Saját implementáció

3.5.1.3 API vezérlők (Controllers)

A szakdolgozati projekt során az API vezérlők felelősek a szerveroldali logika és a kliens közötti kommunikáció kezeléséért. Ezek végpontokat biztosítanak, amelyekhez a kliensoldali Angular alkalmazás különböző HTTP kérésekkel tud kapcsolódni. Minden fő funkcióhoz külön vezérlő tartozik (pl. felhasználói profil (vö. 3.5.3.2 fejezet), cserék (vö. 3.5.3.6 fejezet))

Swagger dokumentáció használata

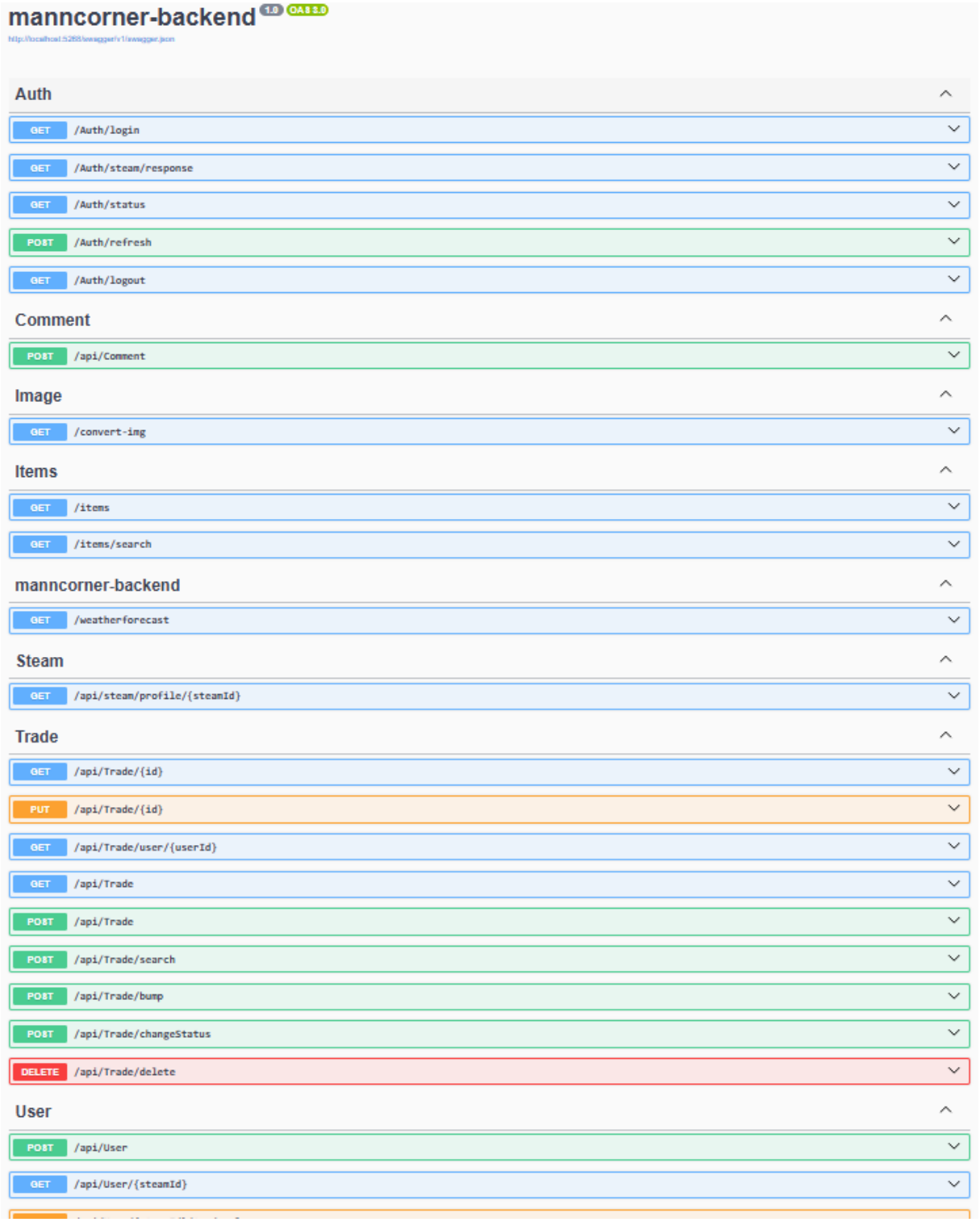
A backend szolgáltatásaim dokumentálására a Swagger nevű, integrált eszközt alkalmaztam. A Swagger automatikusan generálja a webes API dokumentációt az ASP.NET Core projektekben. A Swagger nagyon hasznos tud lenni:

- lehetővé teszi a végpontok teljes körű vizuális áttekintését
- Megjeleníti minden végponthoz tartozó elvárt paramért, minta kérés és választ, illetve kipróbálható a felületen.

A vezérlők példái a projekten belül, néhány fontosabb controller:

- AuthController
  Kezeli a Steam OpenId bejelentkezési folyamatot, valamint a JWT token generálást.
    - Válasz: Sikeres hitelesítés esetén JWT token
- UserController
  Felhasználói adatokat, profilmódosítást, statisztikákat kezel
    - Végpontok:
        - /api/User (POST) felhasználó létrehozása
            - Elvárt paraméterek: steamId, tradeUrl
        - /api/User/{steamId} (GET) felhasználó lekérdezése
            - Elvárt paraméterek: steamId
        - /api/User/{steamId}/tradeurl (PUT) felhasználó csereUrl módosítása
- TradeController
  A cserék létrehozását, módosítását, ajánlatok kezelését biztosítja
    - Végpontok:
        - /api/Trade/{id} (GET) Egy csere részletes adatainak a lekérdezésére szolgál
        - /api/Trade (GET) Visszaadja a cseréket
            - Elvárt paraméterek: mettől, mennyit adjon vissza (oldalanként kérdezhető le)

A következő képen látható a Swagger UI főoldala, amelyen áttekinthetőek a projektben elérhető REST végpontok és azok metódusai.



9. ábra - Swagger dokumentáció – Forrás: Saját képernyőfotó

3.5.1.4 Entity Framework Core: Cold vs Warm Start jelenség

„Az Entity Framework Core cold start jelenség akkor jelentkezik, amikor az alkalmazás első lekérdezése egy adott DbContext típussal történik. Ebben az esetben az EF Core számos

háttérműveletet végez - betölti és validálja a modellt, metadata loading-ot hajt végre, view generation-t futtat, majd lefordítja a LINQ kifejezéseket SQL utasításokká és cache-eli ezeket az információkat. Ez az első "cold" lekérdezés akár 5-10-szeresen lassabb lehet, mint a későbbi "warm" lekérdezések, mivel azok már a cache-elt fordítási eredményeket használják." (medium.com (1), 2025)

3.5.2 Frontend megvalósítás

Az angular keretrendszert választottam a kliens oldal elkészítéséhez, mivel:

- **Komponens alapú felépítés**: Könnyű újra felhasználható komponenseket készíteni (51. ábra), ezzel egyszerűsítve a fejlesztői feladatokat
- **Kétirányú adatkövetés (data binding)**: Lehetővé teszi a modell és a megjelenítés szoros szinkronját
- **Beépített szolgáltatások**: Form kezelés, validáció, routing, HTTP kliens és interceptors használata gyorsítja és egységessé teszi a fejlesztést
- **Typescript alap**: Biztonságosabb kódírást és könnyebb hibakeresést tesz lehetővé

Az alkalmazás felépítése különböző komponensekre tagolódik. A komponensek csomópontonként kerültek szeparálásra, tehát egy adott „modul” pl, authentikációhoz tartozó komponensen, authentikációs mappába kerültek tárolásra. A komponensek szolgáltatásokon (services) keresztül kommunikálnak a backenddel. A routing biztosítja az oldalváltásokat és a hozzáférési jogosultságokat.

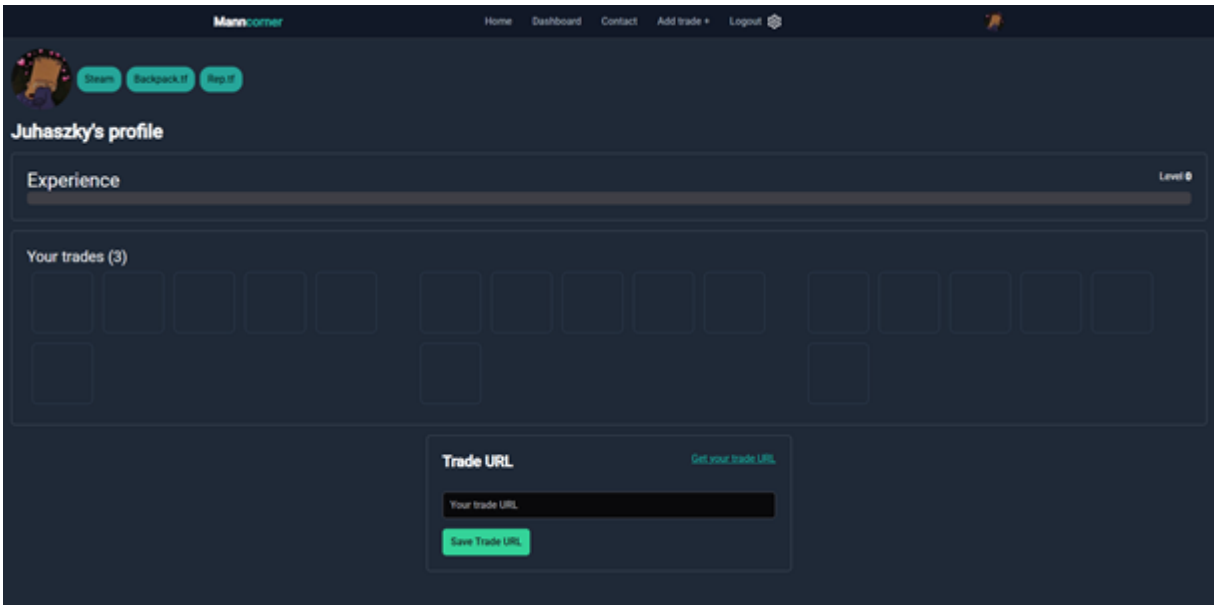
3.5.2.1 Felhasználói authentikáció kezelése

A legenerált JWT token szolgál a további API hívások hitelesítésére. Az angular alkalmazásban a JWT eltárolásra került a böngészőben sütiként és minden HTTP kéréshez automatikusan hozzáadódik, így biztosítja a szerveroldali jogosultságellenőrzést. A kliens oldalon a JWT kezelése egy dedikált authentikációs szolgáltatásban történik, amely felelős a token tárolásáért, lekéréséért és törléséért. Ezzel együtt egy http interceptor gondoskodik róla, hogy minden API hívás tartalmazza a megfelelő hitelesítést.

3.5.2.2 Profil megjelentése és szerkesztése

Az alfejezetben bemutatom az alkalmazás felhasználói profilkezelési funkcióit. Azt, hogy hogyan jelennek meg és hogyan szerkeszthetők a felhasználói adatok, különös tekintettel a Steam platformról származó Trade URL beállítására. Bemutatom a játékosított tapasztalati pont

(XP) és szintlépési rendszer megjelenítését. A bejelentkezett felhasználó megtekintheti személyes adatait, valamint konfigurálhatja a Steam fiókjához kapcsolódó Trade URL-t, amely elengedhetetlen a tárgycserék lebonyolításhoz.



10. ábra - Saját profil nézet – Forrás: Saját képernyőfotó

A 10. ábra fejlécében megjelennek úgynevezett „chip” elemekben a felhasználói fiókhoz tartozó egyéb, külső oldalon szereplő profiladatok. Ilyen a „Backpack.tf”, amely egy hasonló platform, illetve a „Rep.tf”, amelyen a Steam felhasználói fiókodhoz tartozó értékelések jelennek meg. Az említett oldalakra nem szükséges regisztrálni, amennyiben valaki rendelkezik steam fiókkal az automatikusan megfog jelenni az említett oldalakon. Kilistázásra kerül a felhasználóhoz tartozó legutolsó három, legfrissebb csere, amely általa került létrehozása, illetve a képernyő alsó részén megjelenik a Trade URL beállítási lehetőség, egy linkkel beágyazva, amely a Steam-es saját profil oldalára továbbít minket, ahonnnan kimásolható a Steam által legenerált egyedi tradeUrl. Illetve a felületen megjelenítjük a felhasználóhoz tartozó tapasztalati pontokat és a szintjét.

3.5.2.3 Tapasztalati pont (XP) és szintlépési rendszer megjelenítése

Az alkalmazás játékosított elemet is tartalmaz, amely aktivitás alapján tapasztalati pontokat (XP)-t oszt ki a felhasználóknak

- Az XP gyűjtése az ajánlattételektől, bejelentkezésektől és egyéb funkciók használatól függ

- A megszerzett XP alapján a felhasználó szintet lép, amely a profilon megjelenik későbbiekben stílusokat vásárolhat belőle (vö. 6.2 fejezet)
- Szintlépés motiválja és ösztönzi a felhasználókat az aktív részvételre

3.5.2.4 Csere-igény hirdetési felület

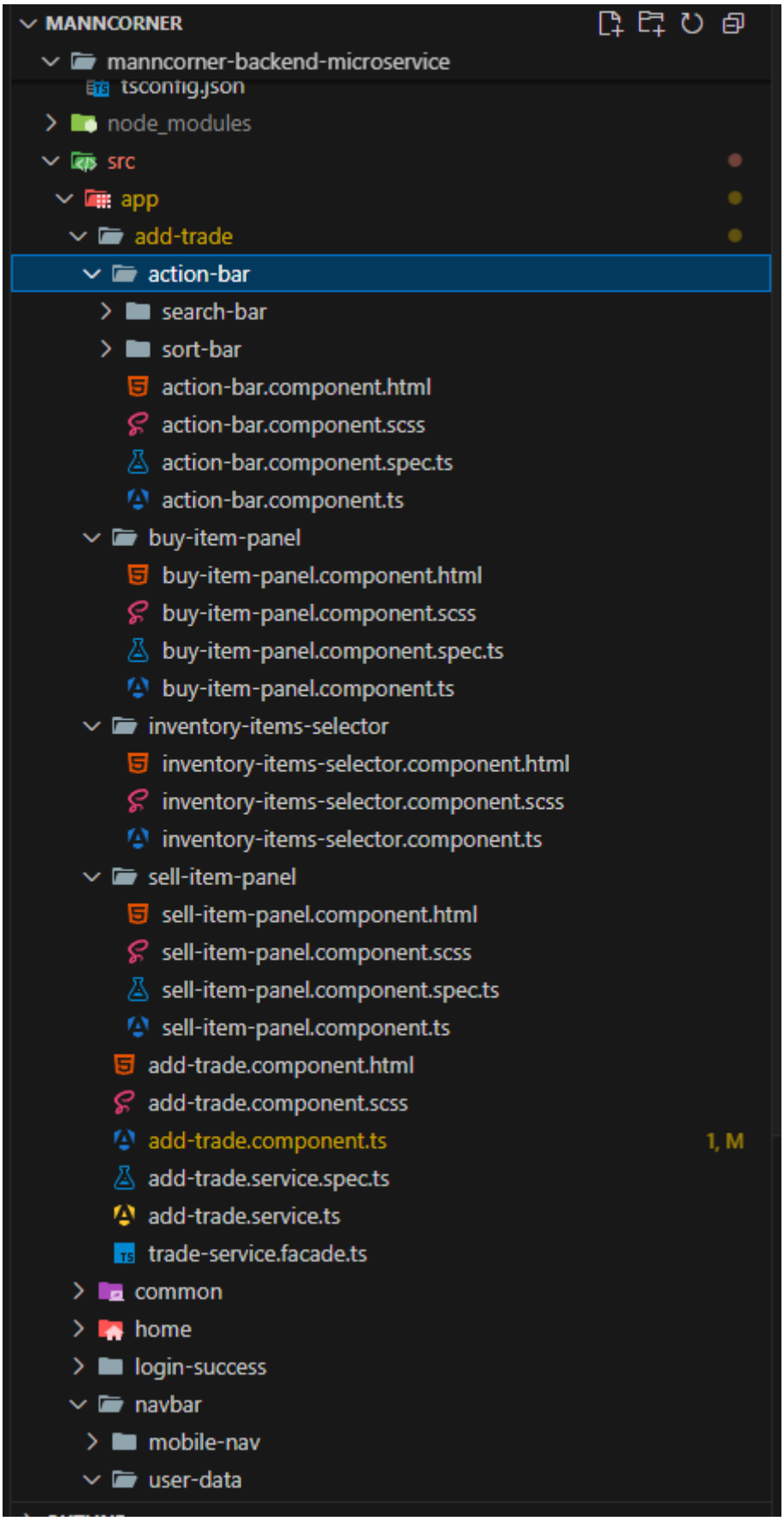
A csere-igény hirdetési felület fő célja, hogy a bejelentkezett felhasználó az „Add trade+" menüponton belül egyszerűen és áttekinthetően meghirdethesse az általa eladni kívánt virtuális tárgyakat, valamint beállítson egy cserét-igényt az általa keresett tárgyakra.

Komponensek szétbontása

- Cél: Minden komponens csak egy jól körül határolt feladattal foglalkozzon — például, vagy csak a megjelenítéssel, vagy csak a konkrét logikai műveletekkel.
- Ez megkönnyíti a tesztelést, és az újra felhasználhatóságot.

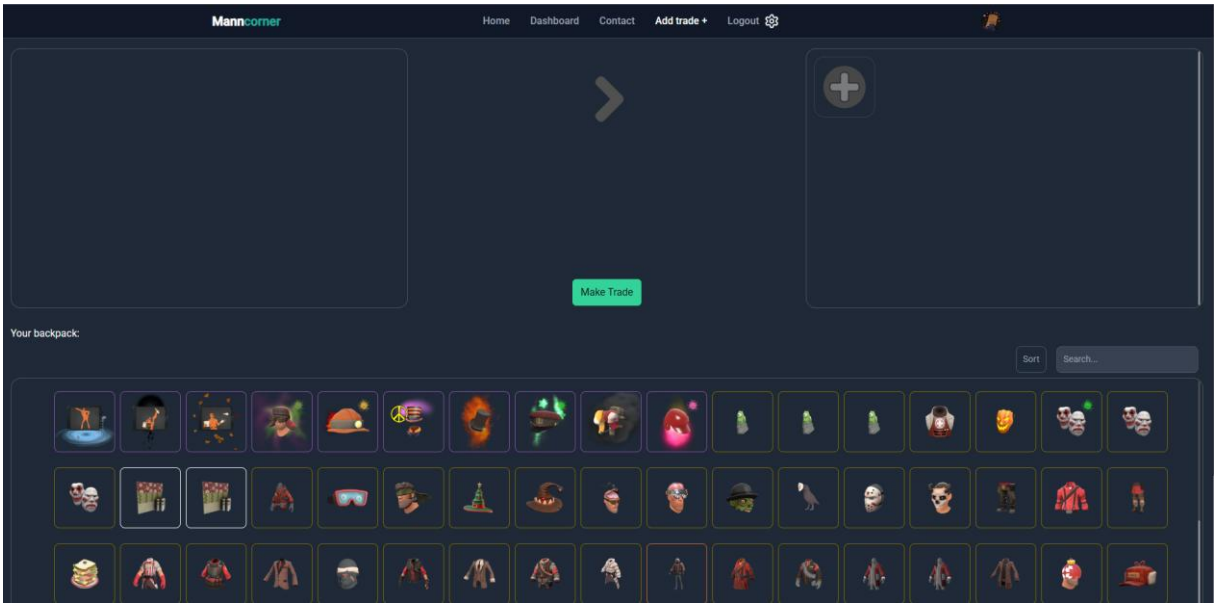
Facade pattern használata

Az „item-selector-facade" (48. ábra) egy külön szolgáltatásként (service) működik, amely az összetettebb logikát egy egységes, egyszerű felületen keresztül kezeli. Feladata, hogy összefogja azokat a műveleteket, melyek a tárgyak kiválasztásához kapcsolódnak (pl. választott tárgyak tárolása, állapot kezelése).
A facade elrejti a komplex belső működést a komponensek elől, így azok csak a megjelenítésre és interakciókra koncentrálhatnak. Ez javítja az alkalmazás skálázhatóságát és karbantarthatóságát, mert a változtatások nagy része a facade-ben kezelhető anélkül, hogy az összes komponenst módosítani kellene. A következő képen bemutatom, hogy az architektúra, amelyet alkalmaztam az, hogyan néz ki a csere-igény hirdetési felület esetén:



11. ábra - Komponens alapú architektúra - csere hirdetési felület – Forrás: Saját képernyőfotó

3.5.2.5 Grafikus felület felépítése



12. ábra - Csere-igény hirdetési felület – Forrás: Saját képernyőfotó

A 12. ábra bal oldalán a „tárgyak eladása” doboz elembe a felhasználó által kijelölt, eladni kívánt tárgyak kerülnek, jobb oldalán „tárgyak beszerzése” doboz elembe a felhasználó által kijelölt, beállított, vásárolni kívánt tárgyak kerülnek. A felület alsó részében megjelennek a Steam végponton keresztül lekérdezhető, felhasználóhoz tartozó tárgyak. A komponens megjelenítésekor automatikusan huszonöt tárgy kerül lekérdezésre, a rövid válaszidő miatt. A többi tárgy lekérdezésére az alábbi két lehetőség adott a felhasználó számára:

- Legörgetés esetén, amennyiben a doboz, amelyben szerepelnek a tárgyak görgetésre kerül és majdnem a doboz aljára kerül a csúszka, abban az esetben újabb lekérdezés indul, amely lekérdezi a következő huszonöt elemet a felhasználó tárgyai közül.
- Keresés esetén, lekérdezés történik az alkalmazás szerveroldalától, ahol a keresési mezőbe beírt paraméter alapján szűr a felhasználó által birtokolt tárgyak közül.

A lekérdezés szerveroldali módja egy másik fejezetben kifejtésre kerül, (vö. 3.5.3.6 fejezet). A szűrési eredmények a 13. ábra alapján kerülnek megjelenítésre. A felhasználónak lehetősége van szűrni a lekérdezett tárgyai között az alábbi módon:

- Szöveges keresés, a felhasználó tárgyai között szöveges módon keres
- Rendezés, felhasználói tárgyai között az alábbi szempontok alapján rendez:
  - Név – ABC sorrendben való listázás
  - Ritkasági típusok (vö. 3.1.1 fejezet)

13. ábra - Szűrési eredmény – Forrás: Saját képernyőfotó

„Tárgyak eladása” elem felépítése

Amennyiben hozzáadásra került egy elem, akkor a felhasználó hátizsákjából ideiglenesen kivételre kerülnek a tárgyak, ezt a felület egy halvány stílussal jelzi ezt a felhasználó számára,

lásd 14. ábra.

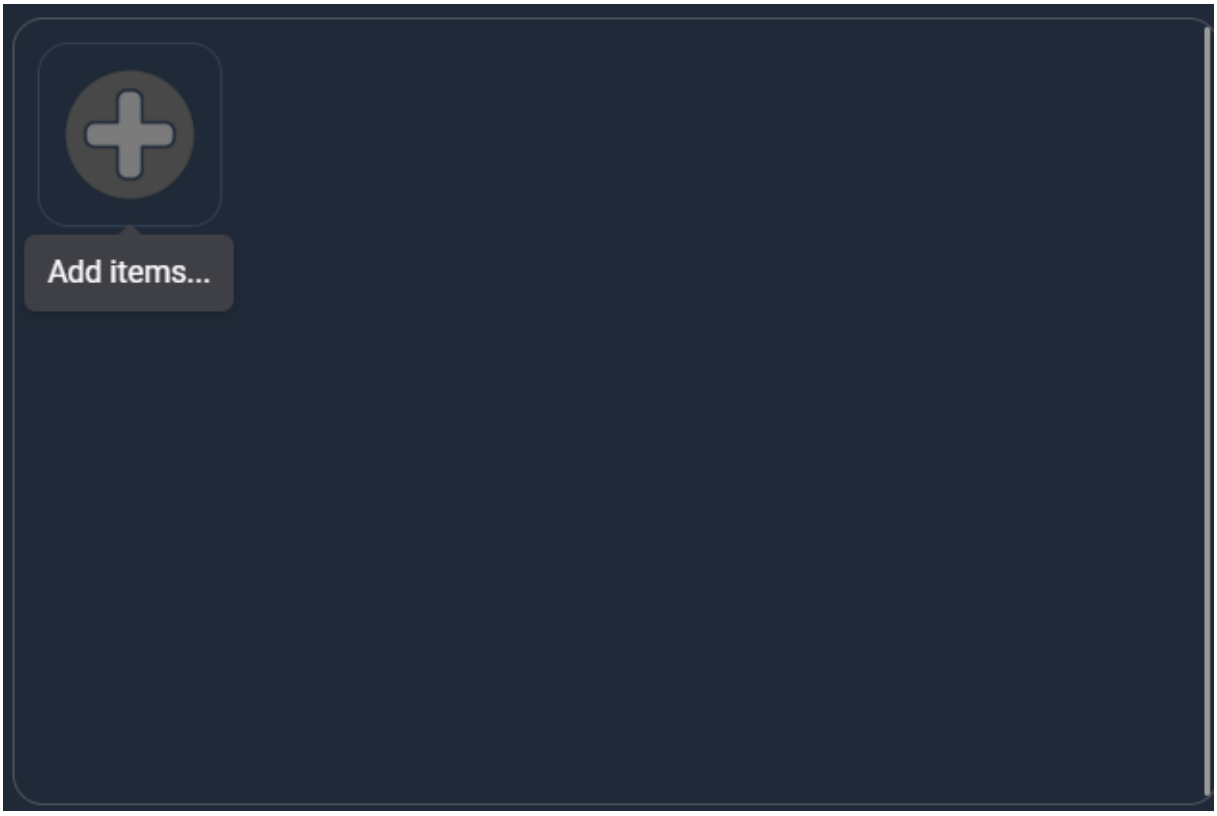
14. ábra - Kiválasztott elem ábrázolása – Forrás: Saját képernyőfotó

A kiválasztott elemeket a „tárgyak eladása” dobozon belül törölni tudja felhasználó (15. ábra), ilyenkor megszűnik a kijelölés és visszakerül a tárgy az eredeti helyére.

15. ábra - Törlés funkció – Forrás: Saját képernyőfotó

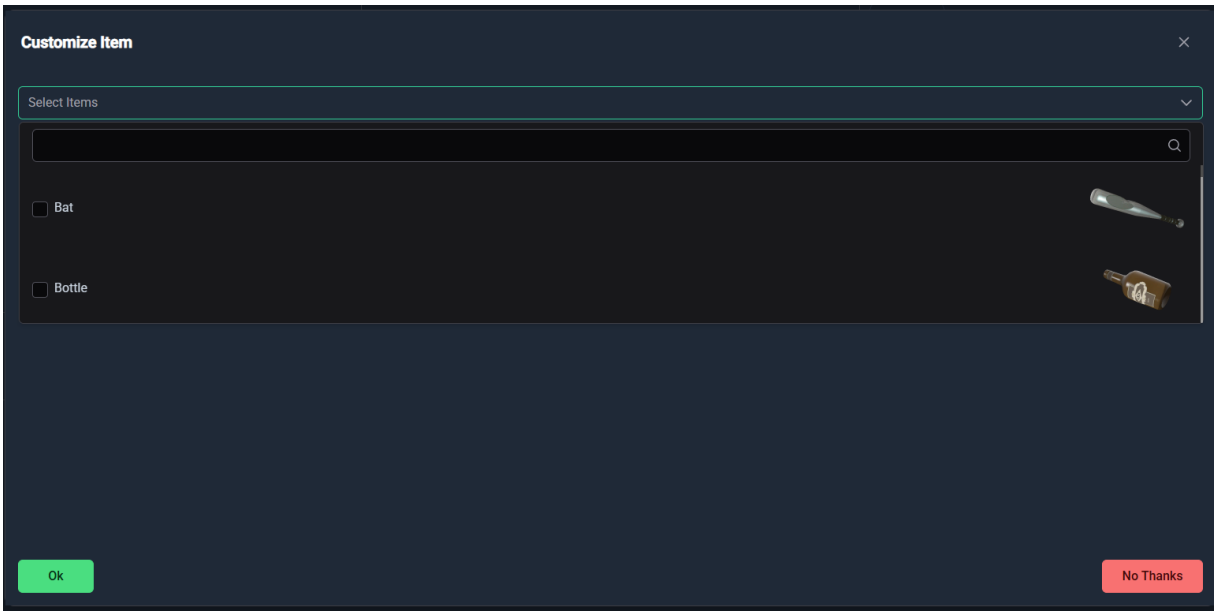
„Tárgyak vásárlása” elem felépítése

A tárgyak vásárlása dobozban alapértelmezetten a tárgyak hozzáadása lehetőség érhető el (16. ábra). A kliens a háttérben automatikusan lekérdezi a microserviceként használt nodejs alapú szerverről (vö. 3.5.4 fejezet) a játékban szereplő összes tárgyat.



16. ábra - Tárgy vásárlása funkció – Forrás: Saját képernyőfotó

A felhasználó ki tudja választani azokat a tárgyakat, amelyekre ő nyitott, azaz, amit szeretne kapni az eladni kívánt tárgyaiért cserébe. Az alap tárgyat kell kiválasztaniuk a felületen (17. ábra) majd ezután lehetőségük van a tárgy módosítására, típus, színezék, hatás kiválasztására is (19. ábra). Teljesen személyre szabhatóak a tárgyak és pontosan meghatározható az a tárgy, amelyet a felhasználó szeretne vásárolni.



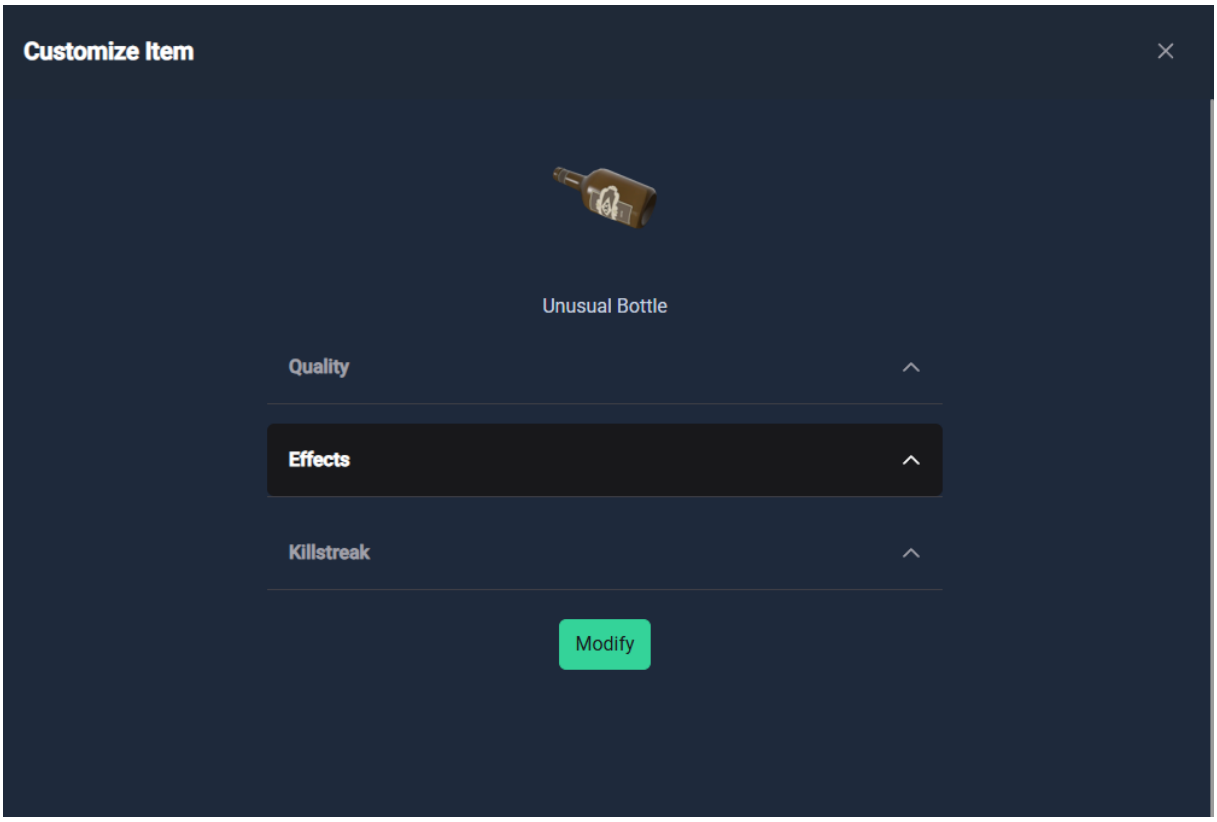
17. ábra - Alapértelmezett tárgy kereső felület – Forrás: Saját képernyőfotó

A kiválasztott tárgyak megjelennek egy listát nézetben is, ahol lehetőségük van a tárgyak módosítására vagy törlésére, elemenként (18. ábra).

18. ábra - Tárgyak listás nézete – Forrás: Saját képernyőfotó

A módosítani kívánt tárgyakon a következő beállítások érhetőek el:

- Ritkasági típus
- Hatás
- „killstreak”

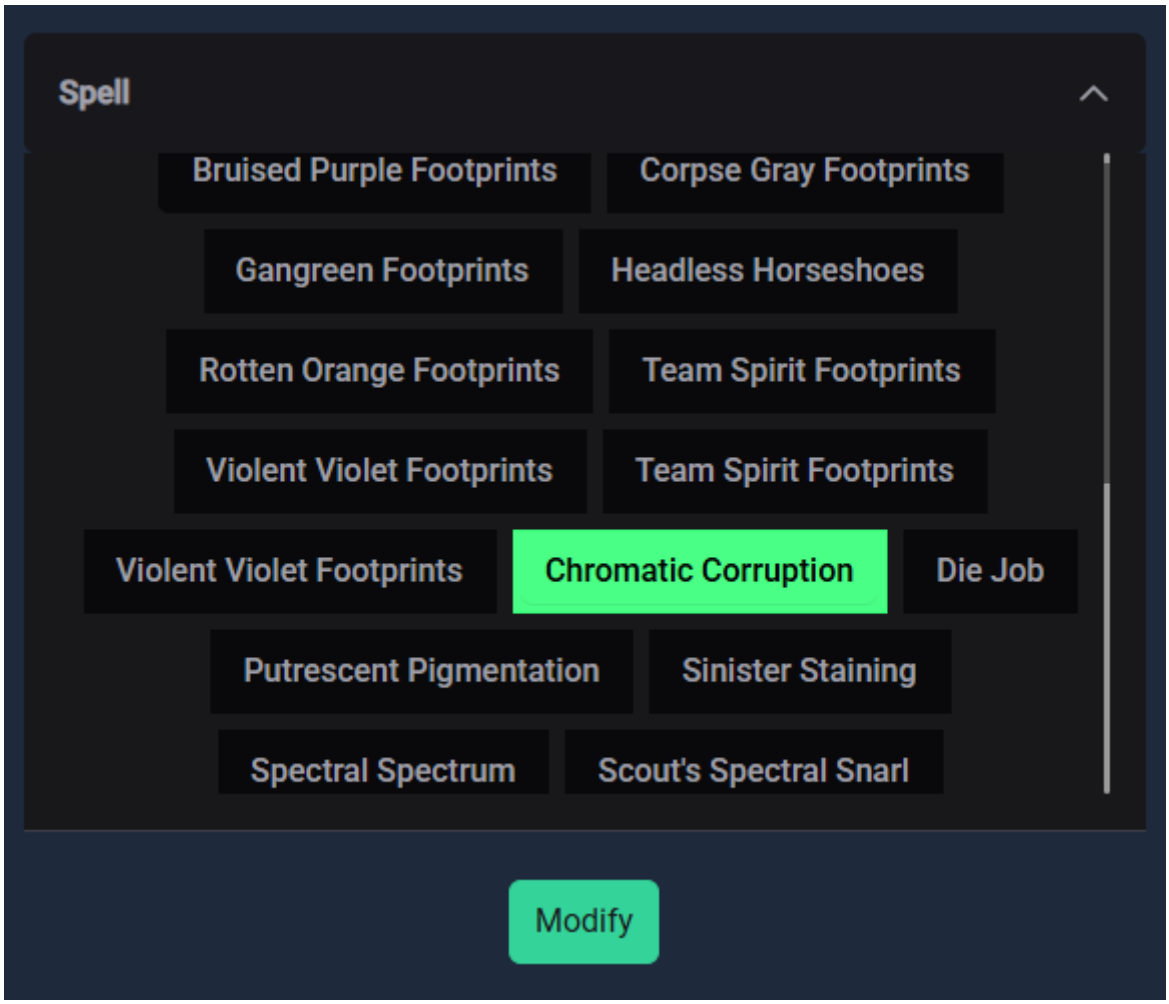


19. ábra - Tárgy személyre szabása – Forrás: Saját képernyőfotó

A konfigurációs felületen az alábbi tulajdonságokat állíthatja be a felhasználó a kiválasztott tárgyra:

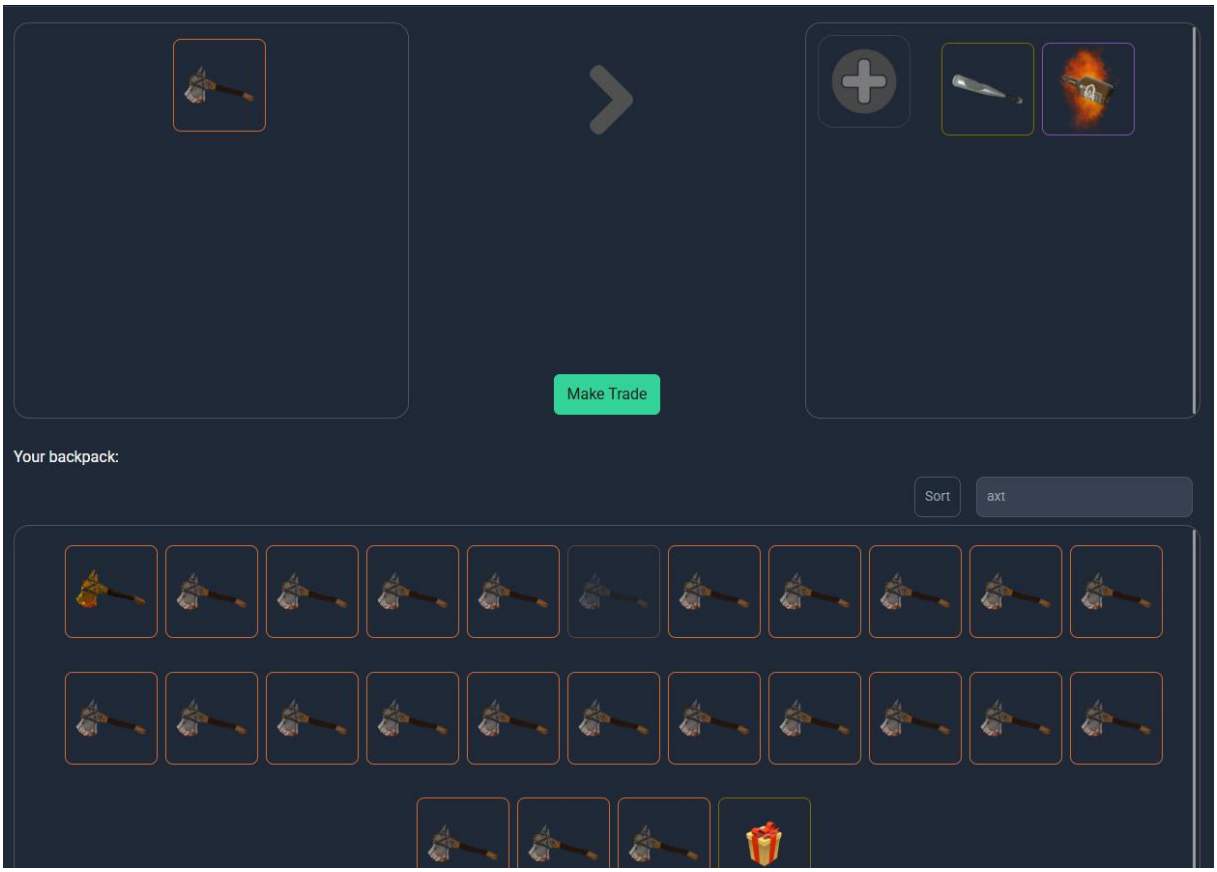
- Quality – típus
- Effects – hatások
- Killstreak

Mindegyik tulajdonság-választó menüpont egy úgynevezett „accordion-panel” -ben helyezkedik el, ezt lenyitva érheti el a felhasználó a kiválasztható értékeket (20. ábra).



20. ábra - Varázslatok választó felület – Forrás: Saját képernyőfotó

A személyre szabott tárgy elmentésre kerül a „Modify” gomb megnyomásakor. Ezután megjelenik a felületen a létrehozott tárgy (21. ábra).



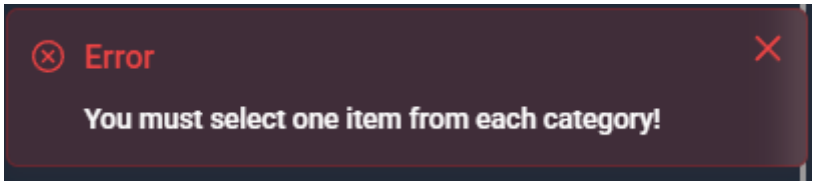
21. ábra - Egyszerű csere-igény bemutatása – Forrás: Saját képernyőfotó

Validáció

Egy csere-igény létrehozásánál az alábbi szabályoknak kell érvényesülniük:

- Legalább egy eladni kívánt tárgyat szükséges kiválasztani
- Legalább egy vásárolni kívánt tárgyat szükséges kiválasztani
- Típusonként maximum tíz tárgy választható

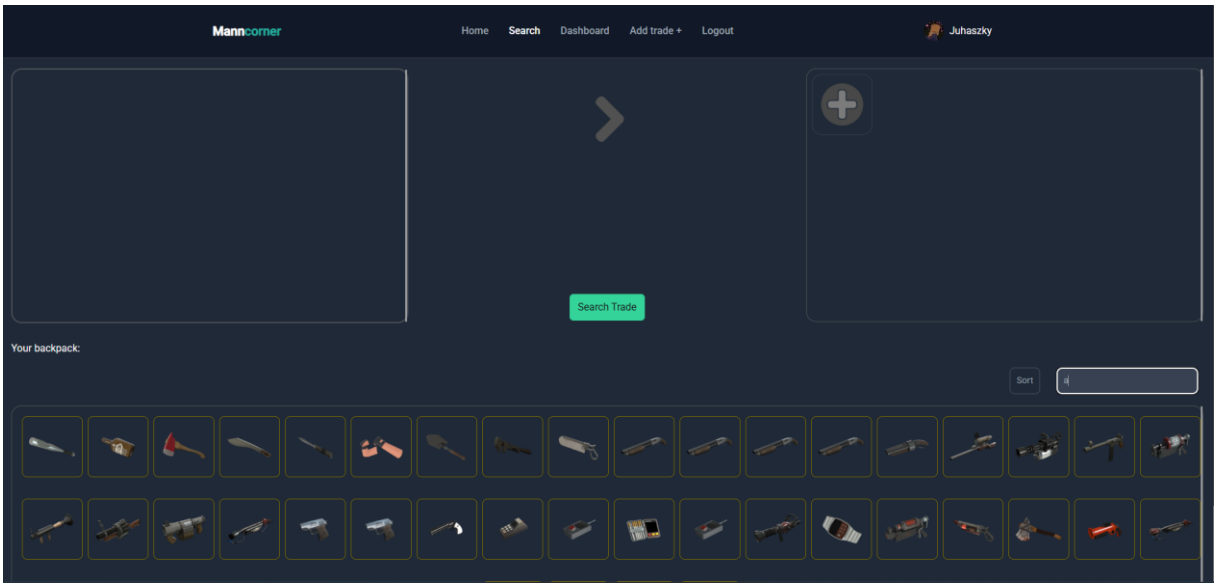
A szabályok megsértéséről a felhasználó figyelmeztetést kap a rendszertől (22. ábra), amelyért az alkalmazásban található primeng könyvtárban szereplő üzenet szolgáltató osztály felel.



22. ábra - Figyelmeztető üzenet – Forrás: Saját képernyőfotó

3.5.2.6 Csere-igény keresési felület

A csere keresési felület nagyon hasonló a korábban leírt csere-igény hirdetési felülettel (vö. 3.5.2.5 fejezet). A felületen megjelenő komponensek és a használat megegyezik, azonban a megjelenített tárgyak forrása más. A felületen minden esetben a játékban fellelhető összes, alapértelmezett tárgy megjelenik és a felhasználó ezek közül választva, egyedileg konfigurálva tudja megkeresni az általa beállított csere-igényt/csere-igényeket (23. ábra). A nagyobb teljesítmény érdekében az oldal betöltésekor csak az első ötven tárgy kerül lekérdezésre, majd a további görgetés vagy szűrés alapján a többi tárgy is.



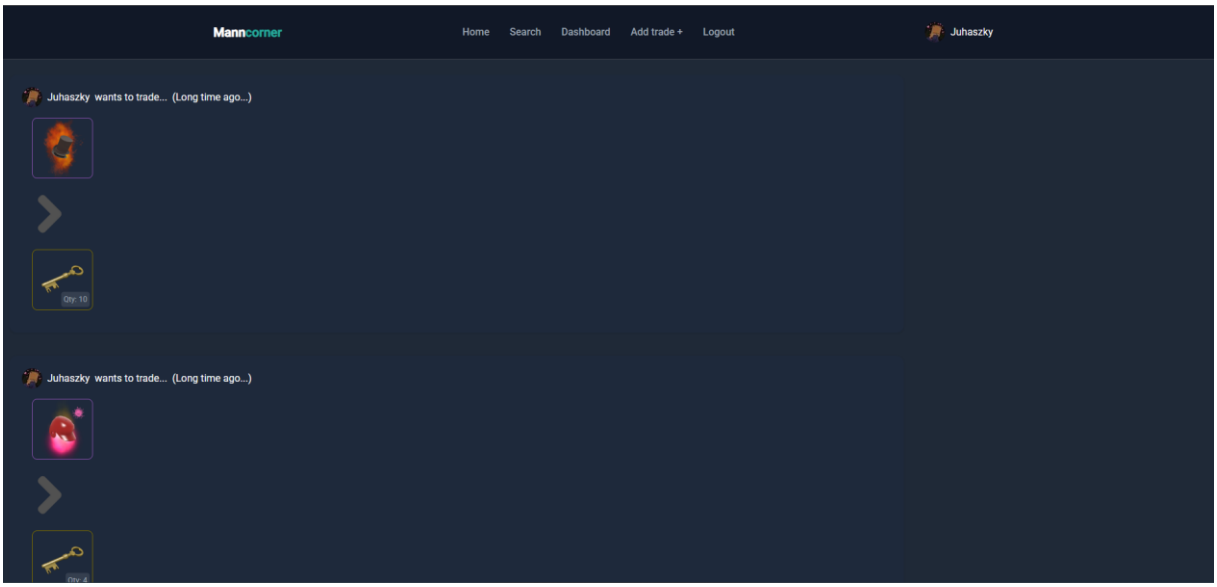
23. ábra - Csere-igény keresési felület – Forrás: Saját képernyőfotó

Validáció

A keresési felület validációja teljesmértékben megegyezik a korábban részletezett, csere hirdetési felületen szereplő logikával. (vö. 3.5.2.4 fejezet)

3.5.2.7 Keresési eredmények

A keresési találatokat a felhasználó által megadott keresési paraméterek alapján jeleníti meg a felület. Amennyiben a felhasználó kiválasztott egy eladó tárgyat, azaz a (23. ábra) bal oldalára került felvételre, akkor abban az esetben az alkalmazás azokat a keresési eredményeket fogja visszaadni, ahol a felhasználók szintén eladó tárgyként jelölték meg az adott tárgyat. A részletesebb keresési logika a szerveroldali megoldásoknál kerül kifejtésre, az alábbi fejezetben (vö. 3.5.3.6 fejezet). A keresési eredményeket (24. ábra) az alkalmazás az alábbi elérési útvonalon jeleníti meg: „/results”



24. ábra - Keresési eredmények felület – Forrás: Saját képernyőfotó

A megjelenített keresési eredmények között a felhasználónak lehetősége van a lapozásra. Alapértelmezetten tíz eredmény kerül megjelenítésre. A skálázható és modern szerveroldali megoldásnak köszönhetően a keresési sebesség több felhasználó esetében is gyors (<30 ms) és optimalizált. Szakdolgozatom során fokozottan figyeltem arra, hogy az alkalmazásom minél gyorsabb és a lehető legkevesebb erőforrást vigyék el egy-egy lekérdezések. Ezért úgynevezett stress-test-nek tettem ki az alkalmazásomat és néhány főbb funkció mögötti szerver és kliens oldalt teszteltem. A tesztelési eredményeket, amelyek a keresési funkcióhoz kapcsolódnak, az alábbi fejezetben kerül kifejtésre (vö. 3.6.3 fejezet).

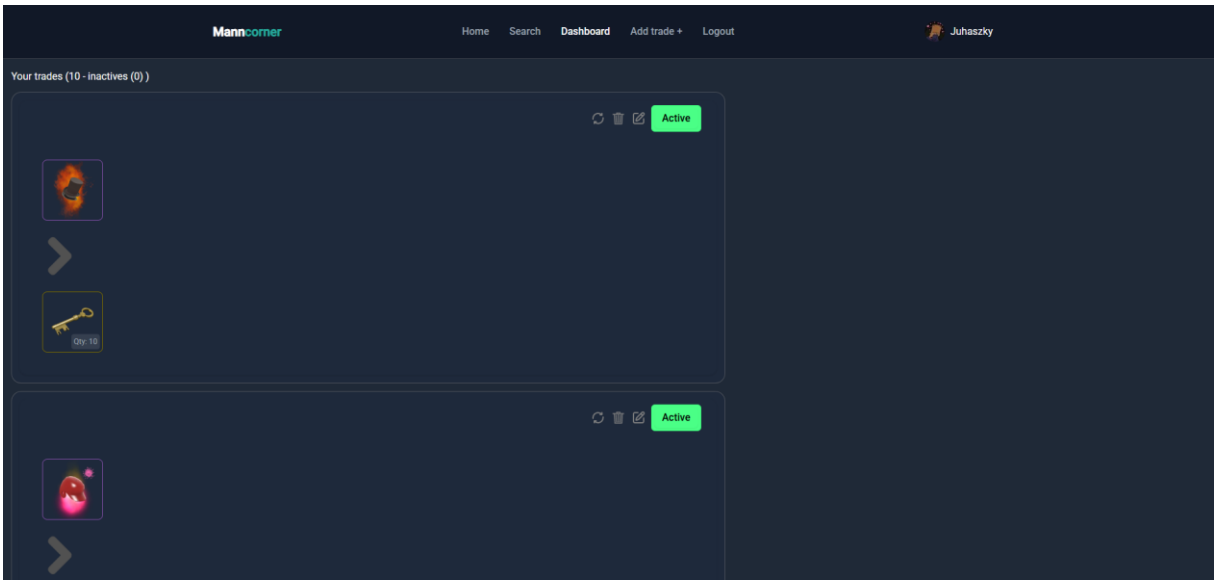
3.5.2.8 Csere-igény kezelő felület

A csere-igény kezelő felületen (25. ábra) az aktuális bejelentkezett felhasználó által létrehozott cserék jelennek meg lista-nézet formátumban. Jelen alfejezetben kifejtett felületen a felhasználó az alábbi interakciókat hajhatja végre:

- **Csere-igény frissítése:** a frissítési dátumot állítja át a frissítés gomb megnyomás idejére, ezzel a felhasználó csere-igénye előre kerül a listázásban.
- **Csere-igény törlése:** A felhasználó véglegesen törölni tudja a csere-igényét, ez a művelet végleges, így ez a művelet fokozott figyelemmel hajtható csak végre.
- **Csere-igény módosítása:** A korábban létrehozott csere-igényt módosítani lehet, tárgyakat törölhetünk, inaktívvá tehetünk, illetve, akár teljesen módosíthatjuk is azt, a meglévő kommentek és ajánlatok meghagyásával!

- **Csere-igény státusz állítása:** Lehetőséget nyújt a meglévő csere-igények felfüggesztésére. Ilyen esetben a csere-igény nem kerül törlése, viszont nem is fog megjelenni mások számára sem.

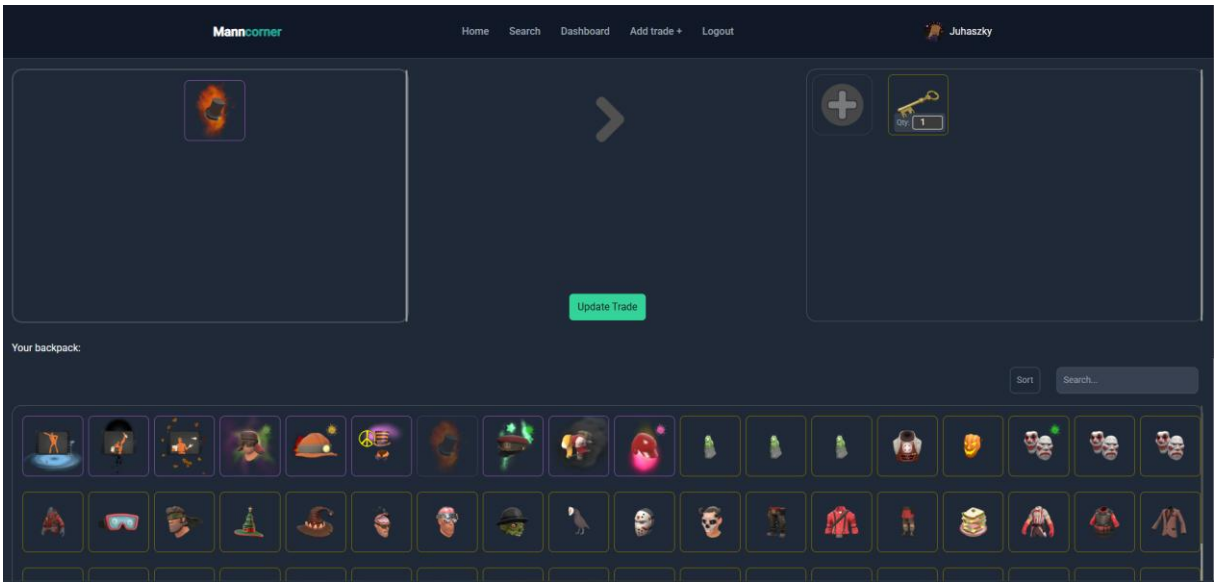
A felsorolt műveletek (frissítés, törlés, módosítás, felfüggesztés) kizárólag bejelentkezett és hitelesített felhasználók számára érhetők el, így biztosítva, hogy csak a saját csere-igény ajánlatok kezelhetők. Ez megakadályozza, hogy más felhasználók jogosulatlanul módosítsák, töröljék vagy állítsák át mások által létrehozott csere-igényeket. Az ilyen védelem az adatok és a felhasználói fiókok biztonsága érdekében elengedhetetlen, és minden művelet végrehajtása szigorúan a felhasználó azonosításához kötött.



25. ábra - Csere-igény kezelő felület – Forrás: Saját képernyőfotó

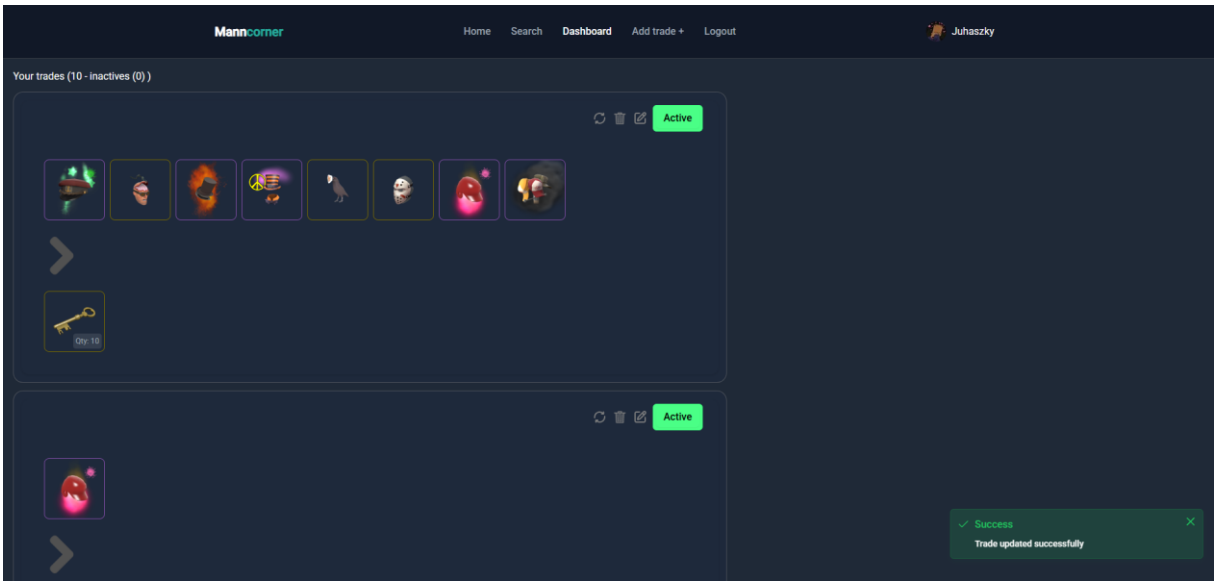
3.5.2.9 Csere-igény módosítási felület

A bejelentkezett felhasználó a csere-igény kezelési felületen kitudja választani a már korábban létrehozott csere-igény módosítását. Ebben az esetben az alkalmazás a felhasználót átirányítja egy másik felületre, ahol betöltésre kerül a kiválasztott, módosítani kívánt csere-igény tárgyai. A felület működése megegyezik a csere-igény hirdetési felülettel (vö. 3.5.2.4 fejezet), azzal a különbséggel, hogy ebben az esetben a csere-igény nem újként jön létre, hanem a meglévő csere-igény kerül módosításra. Illetve amennyiben az utolsó dátum frissítés óta eltelt öt perc, abban az esetben a csere-igény módosításakor a dátum frissítés is megtörténik, tehát a legfrissebb csere-igények között fog megjelenni a módosított csere-igény.



26. ábra - Csere-igény módosítási felület – Forrás: Saját képernyőfotó

A módosítás tényéről a primeng könyvtárban implementált üzenet szolgáltatás osztály jelzi a felhasználónak (27. ábra).



27. ábra - Sikeres csere módosítás üzenet – Forrás: Saját képernyőfotó

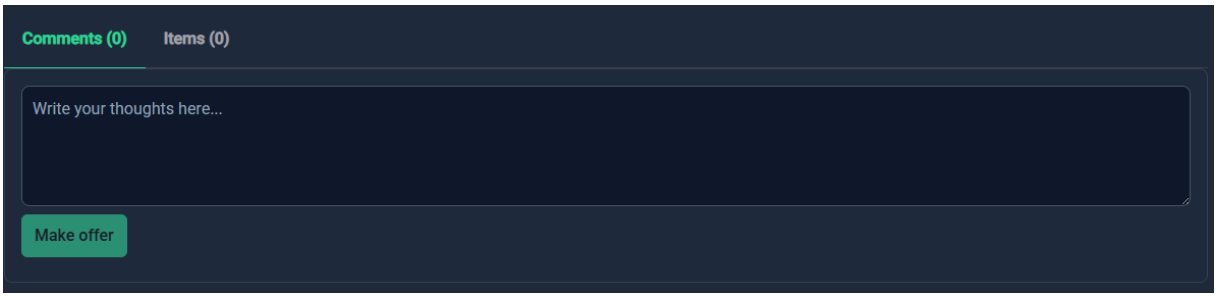
3.5.2.10 Komment és Ajánlattétel felület

A felhasználók ajánlatot tehetnek más felhasználók által létrehozott csere-igényekre komment formájában (28. ábra). Amikor egy felhasználó megnyit egy másik felhasználó által létrehozott cserét, lehetőséget kap komment írására és tárgyak felajánlására. A felület két fül (tab) köré épül:

- Comments (Kommentek) fül: Szöveges komment megírására szolgál

- Items (Tárgyak) fül: Konkrét Steam-tárgyak kiválasztására és felajánlására szolgál

Kommentek fül funkciói



28. ábra - Kommentek felület – Forrás: Saját képernyőfotó

A kommentek fül aktív állapotát mutatja. A felületen (28. ábra) egy szövegmező található, ahol a felhasználó megírhatja ajánlatát vagy észrevételét. A zöld "**Make offer**" (Ajánlat küldése) gomb segítségével rögzíthető a komment. A felület bal felső sarkában két navigációs elem található:

- "Comments (0)": A kommentek fül, aktív állapotban zöld színnel kiemelve
- "Items (0)": A tárgyak fül, zárójelben a kiválasztott tárgyak száma (jelen esetben 0)

Tárgyak fül funkciói

Amennyiben a **tárgyak** fül aktív (29. ábra) abban az esetben a megszokott módon (vö. 3.5.2.5 fejezet) lekérdezi az alkalmazás a felhasználó steam platformon birtokolt tárgyait. Ezek közül kiválasztva a kommenthez menti a rendszer a tárgyakat és a komment rögzítésekor megjeleníti a kommenthez a tárgyakat, így lehet konkrét tárgyakat felajánlani más felhasználók által létrehozott csere-igényekhez.

29. ábra - Ajánlatadás - tárgy választó felület – Forrás: Saját képernyőfotó

3.5.3 Backend megvalósítás

Az alkalmazás backend részét az ASP.NET Core keretrendszerrel valósítottam meg C# nyelven, mert:

- **Modern és skálázható keretrendszer:** Lehetővé teszi az aszinkron műveletek hatékony kezelését és nagy teljesítményű RESTful API-k fejlesztését.
- **Beépített támogatás az authentikációhoz:** Könnyedén integrálható a Steam OpenID hitelesítés, valamint a JSON Web Token (JWT) alapú stateless authentikáció, amely biztonságos kliens-szerver kommunikációt biztosít.
- **Entity Framework Core használata:** Az ORM elősegíti az adatbázis műveletek egyszerű kezelését, az adatok lekérdezését és módosítását objektum-orientált módon.
- **Könnyen dokumentálható API:** A Swagger eszköz segítségével automatikusan generált dokumentáció teszi átláthatóvá és tesztelhetővé a backend végpontokat fejlesztés és integráció során.
- **Microservice architektúra támogatása:** A TF2-specifikus üzleti logika elkülönítése külön Node.js alapú microservice-re növeli a rendszer moduláris felépítését és karbantarthatóságát.

A backend logikai egységei jól elkülönült controller-ekből állnak, például: authentikációért felelős AuthController, felhasználói adatokat kezelő UserController, valamint a cserék

kezelését és ajánlatok adminisztrációját végző TradeController. A backend logika felhasználja és implementálja a szakdolgozat során bemutatott (4. ábra) entitás kapcsolatokat. Az API végpontok szigorú jogosultságkezelés mellett működnek: az authentikációt követően a kliens oldalról érkező kérések a JWT token ellenőrzésével biztosítottak. Az adatok továbbítása JSON formátumban történik, ami elősegíti a könnyű kommunikációt az Angular frontend és a backend között. A backend moduláris és jól átlátható felépítése elősegíti a további fejlesztéseket és a tesztelést (vö. 3.6.3 fejezet).

3.5.3.1 Szerveroldali authentikáció kezelése

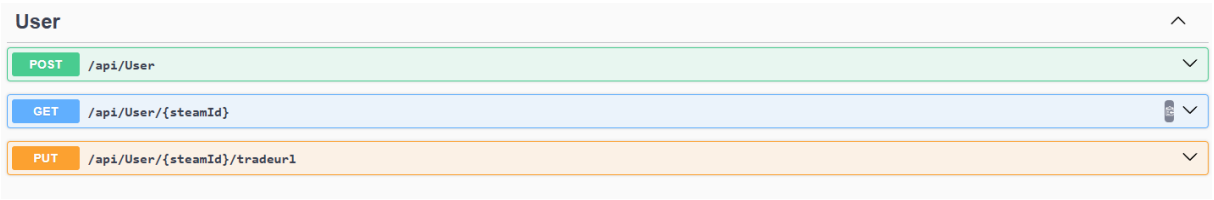
A webalkalmazás szerveroldali authentikációját a Steam OpenID (vö. 3.3.1 fejezet) protokollra építve valósítottam meg ASP.NET Core környezetben. A Steam közösségi fiókkal történő bejelentkezés lehetővé teszi, hogy a felhasználók egyszerűen, külső hitelesítő rendszer segítségével azonosítsák magukat (8. ábra), így a platform nem tárol saját jelszavakat vagy privát belépési adatokat. A bejelentkezési folyamat (5. ábra) során a felhasználó egy OpenID-átirányítással a Steam hivatalos authentikációs felületére kerül, ahonnan sikeres hitelesítés után visszairányítás történik az alkalmazásba, kliens felületére. Ekkor a backend begyűjt néhány általános, publikus adatot a bejelentkezett felhasználó steam fiókjáról. Például: profilnév, profilkép, profilazonosító és új felhasználói rekordként mentésre kerül az adatbázisba. A sikeres authentikáció után (7. ábra) a backend szerver egy JSON Web Token (JWT) hozzáférési tokent generál (vö. 3.3.4 fejezet), melyet HttpOnly sütiként helyez el a felhasználó böngészőjében (30. ábra). Ez a token igazolja a felhasználói státuszt a szerver felé, amikor további védett API végpontok elérésére kerül sor. Emellett egy frissítő (refresh) token kerül generálásra, amely biztonságosan hosszabbítja meg az authentikáció élettartamát anélkül, hogy a felhasználónak újra be kellene jelentkeznie. Ez a felépítés megfelel a modern biztonsági elvárásoknak: az érzékeny adatok csak szerveroldalon vannak kezelve, a sütik HttpOnly megoldást használnak (így JavaScript-ből nem hozzáférhetők), amivel csökken a Cross-site Scripting vagy token-lopás esélye. A Steam OpenID integráció (vö. 3.3.1 fejezet) emellett nagyfokú felhasználói kényelmet és adatvédelmet biztosít.

```
    [AllowAnonymous]
    [HttpGet("login")]
    0 references
    public IActionResult Login()
    {
        var properties = new AuthenticationProperties
        {
            RedirectUri = Url.Action("SteamResponse")
        };
        return Challenge(properties, SteamAuthenticationDefaults.AuthenticationScheme);
    }
    [AllowAnonymous]
    [HttpGet("steam/response")]      Juhaszky, 5 months ago • base authentication logic implementation
    0 references
    public async Task<IActionResult> SteamResponse()
    {
        var result = await HttpContext.AuthenticateAsync("Cookies");
        if (!result.Succeeded)
        {
            return BadRequest("Authentication failed.");
        }
        var claims = result.Principal.Claims;


        var url = claims.FirstOrDefault(c => c.Type == ClaimTypes.NameIdentifier)?.Value
                  ?? claims.FirstOrDefault(c => c.Type == "sub")?.Value
                  ?? claims.FirstOrDefault(c => c.Type.Contains("nameidentifier"))?.Value;
        var steamId = new Uri(url).Segments.Last();
        if (string.IsNullOrEmpty(steamId))
        {
            return BadRequest("SteamID not found in claims.");
        }
        var existingUser = await _userService.GetUserBySteamIdAsync(steamId);
        if (existingUser == null)
        {
            var externalData = await _steamService.GetPlayerSummary(steamId);

            await _userService.CreateUserAsync(steamId, "", externalData.response.players[0].personaname, externalData.response.players[0].avatarmedium);
        }
        var token = _authService.GenerateJwtToken(steamId);
        Response.Cookies.Append("accessToken", token, new CookieOptions
        {
            HttpOnly = true,
            Secure = true,
            SameSite = SameSiteMode.None,
            Expires = DateTime.UtcNow.AddMinutes(15)
        });

        var frontendUrl = $"http://localhost:4200/home";
        return Redirect(frontendUrl);
    }
```

30. ábra - Szerveroldali authentikáció – kódrészlet – Forrás: Saját implementáció

3.5.3.2 Profilkezelés

A szerveroldali profilkezelést a **UserController** és a **UserService** osztályok valósítják meg, amelyek együttműködve biztosítják a felhasználói adatok kezelését (6. ábra) és a Steam API integrációját.



31. ábra - Swagger dokumentáció – profilkezelés – Forrás: Saját képernyőfotó

A (31. ábra) mutatja be, hogy milyen API végpontok kerültek implementálásra a felhasználói kezeléssel kapcsolatban.

3.5.3.3 Felhasználó létrehozása

A **CreateUser** endpoint fogadja a felhasználó Steam ID-ját és a Trade URL-jét. A controller első lépésként lekérdezi a Steam API-ból a felhasználó publikus adatait (név, profilkép), majd ezeket az adatokat elmenti az alkalmazás az adatbázisba. Ez a folyamat biztosítja. hogy a platform mindig friss és hiteles adatokkal dolgozzon.

3.5.3.4 Trade URL frissítése

A **SetTradeUrl** endpoint különös figyelmet érdemel, mivel két szintű authentikációt és authorizációt valósít meg. Először ellenőrzi a JWT tokent, majd biztosítja, hogy a felhasználó csak a saját Trade URL-jét módosíthassa (token-ben lévő steamId egyezik-e a kérésben szereplővel). Ha a felhasználó először állítja be a Trade URL-jét, a rendszer automatikusan 25 tapasztalati pontot jutalmaz az **ExpService** segítségével, ösztönözve ezzel a platform teljes körű használatát.

3.5.3.5 Tapasztalati pont (XP) és szintlépési rendszer implementálása

Az ExpService osztály felelős a felhasználók tapasztalati pontjainak kezeléséért. Az increaseExp() metódus kap egy pontmennyiséget és egy Steam ID-t, majd a UserService segítségével lekérdezi a felhasználót és növeli az XP értékét. Ez a játékosított elem motiválja a felhasználókat a platform aktív használatára különböző tevékenységek elvégzésével (pl. Trade URL beállítás, csere létrehozás, kommentelés). A service dependency injection-nel van csatolva a UserService-hez, így újrafelhasználja annak lekérdezési logikáját, követve a DRY (Don't Repeat Yourself) elvet.

3.5.3.6 Csere szolgáltatás implementálása

A TradeService és TradeController implementálja a platform központi üzleti logikáját, amely a csere-igény hirdetések teljes életciklusát kezeli. A szolgáltatás réteg dependency injection elvén keresztül kap AppDbContext és ILoggerService példányokat. A CRUD műveletek között szerepel a csere-igény létrehozása (CreateTradeAsync), amely automatikusan beállítja a létrehozási időpontot és a kezdeti bump dátumot. A lapozási logika OFFSET-alapú pagination-nel (Skip, Take) valósul meg, amely 10-100 darab elem közötti oldalméreteket támogat és visszaadja a teljes találati számot, oldalszámot és az aktuális adatokat. A komplex keresési funkció (SearchTradesAsync) dinamikus LINQ query builder-t alkalmaz, amely lehetővé teszi többféle szűrési feltétel együttes alkalmazását - defindex, effect, paint, killstreak, sheen paraméterek alapján. Az authorizáció JWT Bearer token alapú hitelesítést használ a védett

műveleteknél - létrehozás, módosítás, törlés, bump és státusz váltás. A bump mechanizmus időkorlátot alkalmaz - egy hirdetés csak 5 percenként "bump"-olható fel, amely frissíti a BumpDate értéket és ezáltal a hirdetést a lista elejére helyezi az OrderByDescending(t => t.BumpDate) rendezés miatt.

3.5.3.7 Képek optimalizálása

A platform teljesítményének egyik kritikus eleme a képoptimalizálás, mivel minden Team Fortress 2 tárgy rendelkezik Steam-ről származó képpel, amelyek nagy felbontásban és PNG formátumban érkeznek. Az ImageService és ImageController ezt a problémát oldja meg on-the-fly konverzióval, amely a Steam képeket WebP formátumra alakítja át. A SixLabors.ImageSharp library használatával a szolgáltatás először HTTP kéréssel letölti a képet a Steam szerverről, majd memóriában betölti és feldolgozza. Az optimalizálási lépések között szerepel a 128x128 pixel-es átméretezés, amely egységes megjelenítési méretet biztosít a tárgylistákban, valamint a WebP formátumra konvertálás 75%-os minőséggel, amely jelentősen csökkenti a fájlméretet a vizuális minőség minimális romlása mellett.

3.5.3.8 Naplózási szolgáltatás

A backend alkalmazás központi naplózási infrastruktúrája a Serilog library-re épül, amely strukturált és szintek szerint kategorizált log bejegyzéseket tesz lehetővé. A LoggerService wrapper osztály az ILoggerService interface-t implementálja, amely három fő naplózási szintet biztosít: Information, Warning és Error. A konfigurációs beállítások a program.cs-ben kerültek beállításra. A beállítások lehetővé teszik azt, hogy a logolási szolgáltatás elkülönülve napi bontásban hozzon létre új log fájlokat (app-YYYY-MM-DD.log) formátumban a **Logs** könyvtárban. Ezzel biztosítva a hosszútávú működést.

3.5.4 Microservice szerver oldal megvalósítás

Külön microservice komponensként került kialakításra, amely egy nodejs alapú szerver. Célja az, hogy egy publikus könyvtár felhasználásával a legoptimálisabb és a leggyorsabb módon, általános sémát alakítson ki a különböző team fortress 2 tárgyakból és a hozzátartozó tulajdonságokból. Az alábbi okok miatt került kialakításra, külön microservice komponensként a rendszer ezen része:

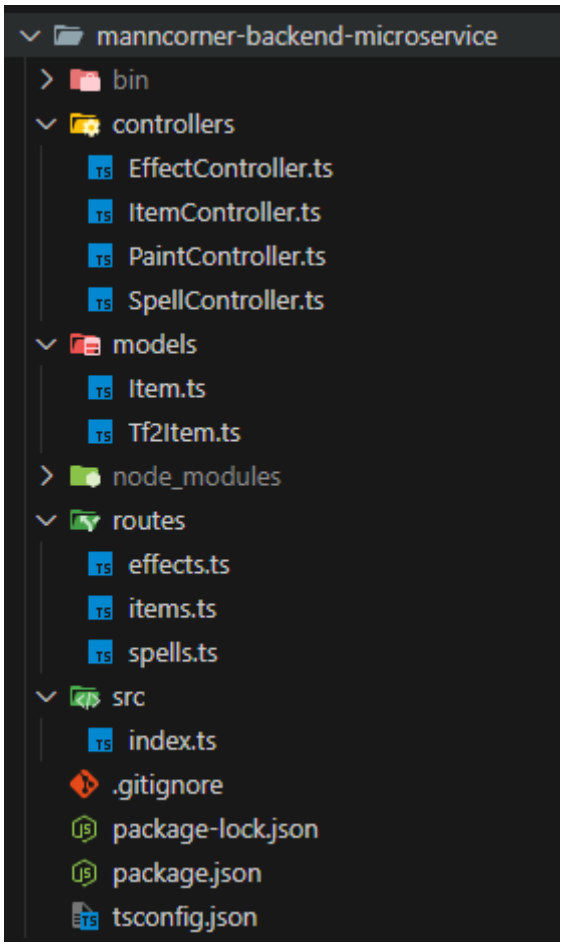
- **Generalizált adatmodell kialakítása:** A tf2-node segítségével az egyedi TF2 tárgyakból és tulajdonságokból egy egységes, optimalizált sémát épít fel, amely elősegíti a gyors és hatékony adatfeldolgozást.

- **Teljesítmény és skálázhatóság:** A különálló microservice architektúra lehetővé teszi a backend fókuszált fejlesztését és skálázását a TF2 tárgyak kezelésére, anélkül, hogy a fő alkalmazás teljesítményét befolyásolná.
- **Publikus könyvtár használata:** A tf2-node (https://github.com/Nicklason/node-tf2-item-format) kihasználja a TF2 közösség által létrehozott megbízható és naprakész adatokat, valamint segít a tárgyak részletes jellemzőinek feldolgozásában (pl. minőség, effektek, sapkák).
- **Funkcionális modulok:** A szerver moduláris felépítésű, támogatja a gyors lekérdezéseket, szűréseket, kereséseket és a tárgyakhoz kapcsolódó metaadatok kezelését.

Műszaki megvalósítás

A tárgyfeldolgozó komponens nodejs környezetben fut, Express.js keretrendszerrel. A tf2-node könyvtár biztosítja a TF2-specifikus tárgyak és tulajdonságok feldolgozását és kezelését. Az adatfeldolgozás REST API-kon keresztül érhető el a backend szerveren keresztül. Így kimondható az, hogy az alkalmazás kliens oldali része csak az ASP.NET alapú szerverrel kommunikál ezzel különválasztva az alkalmazás tárgyfeldolgozó részét. A node.js microservice rétegelt architektúrát követ (31. ábra), amely három fő rétegre osztja a komponenst:

- **Routing réteg:** Express.js router segítségével definiálja a http végpontokat.
- **Controller réteg:** Üzleti logikát valósít meg (formátum-átalakítás, validáció, hibakezelés)
- **Model réteg:** Adatstruktúrat és típusdefiníciót tartalmaz



32. ábra - microservice architektúra – Forrás: Saját képernyőfotó

3.6 A rendszer tesztelése

Ebben a fejezetben bemutatom a fejlesztett webalkalmazás tesztelésének folyamatát, módszereit, valamint a tapasztalatokat és eredményeket. A rendszer megbízhatóságának biztosítása érdekében több szinten végeztem tesztelést:

- Egységtesztek a backend kritikus funkcióira
- Egységtesztek a kliens funkcióira
- Kliens oldali kézi tesztelés
- Integrációs teszteket az adatbázis és az API végpontok helyes együttműködésére
- Kézi tesztelést Swagger (9. ábra) és Postman segítségével a teljes folyamatok ellenőrzésére

A cél az volt, hogy minél korábban észlelhessem az esetleges hibákat, valamint biztosítsam az egyes komponensek és a teljes rendszer stabil működését.

3.6.1 Kliensoldali tesztelés módszertana és megvalósítása

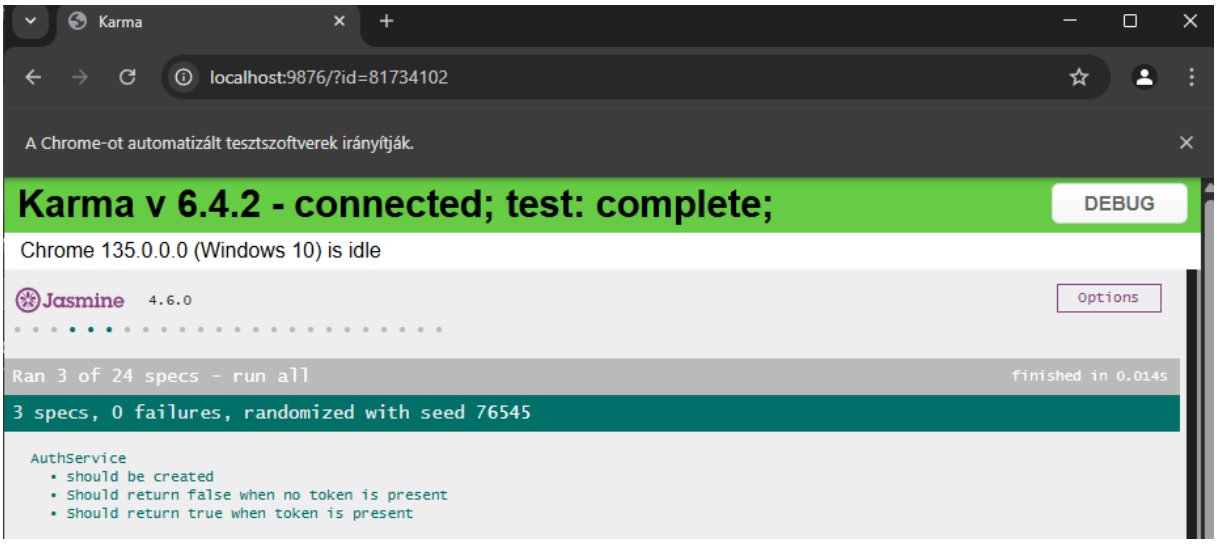
A kliensoldali tesztelés célja, hogy biztosítsuk az Angular alapú webalkalmazás helyes, stabil és felhasználóbarát működését. A gyorsan változó front-end fejlesztési környezetben elengedhetetlen a funkcionális hibák korai felismerése, a regressziós hibák elkerülése és a felhasználói élmény minőségének fenntartása. A teszteléssel minimalizálható a hibák éles rendszerbe kerülése, és lehetőség nyílik a karbantarthatóság növelésére. Az Angular projektekben jellemzően két fő tesztelési szintet különítünk el:

1. Egységtesztelés (Unit Testing): Az egyes komponensek, szolgáltatások, pipe-ok, direktívák viselkedésének izolált tesztelése.
2. Integrációs tesztelés (Integration Testing): Több komponens együttműködésének, adatok átadásának és felhasználói folyamatoknak a tesztelése.

Mindkét típus Jasmine tesztkeretrendszert és Karma futtatókörnyezetet használ Angular CLI projektekben. Az angular projektben alapértelmezetten támogatott a Jasmine tesztelési keretrendszer és a Karma futtató. Az említett technológiák segítségével az „ng test” parancs automatikusan elindítja a teszteseteket az adott projekten belül. Továbbá az „ng test –main -fájl elérési útvonala (projekten belül)” lehetőségünk van egy-egy komponenshez tartozó teszteket futtatni (33. ábra). Példa egy szolgáltatás – authentikációs szolgáltatás tesztelésére: (authService):

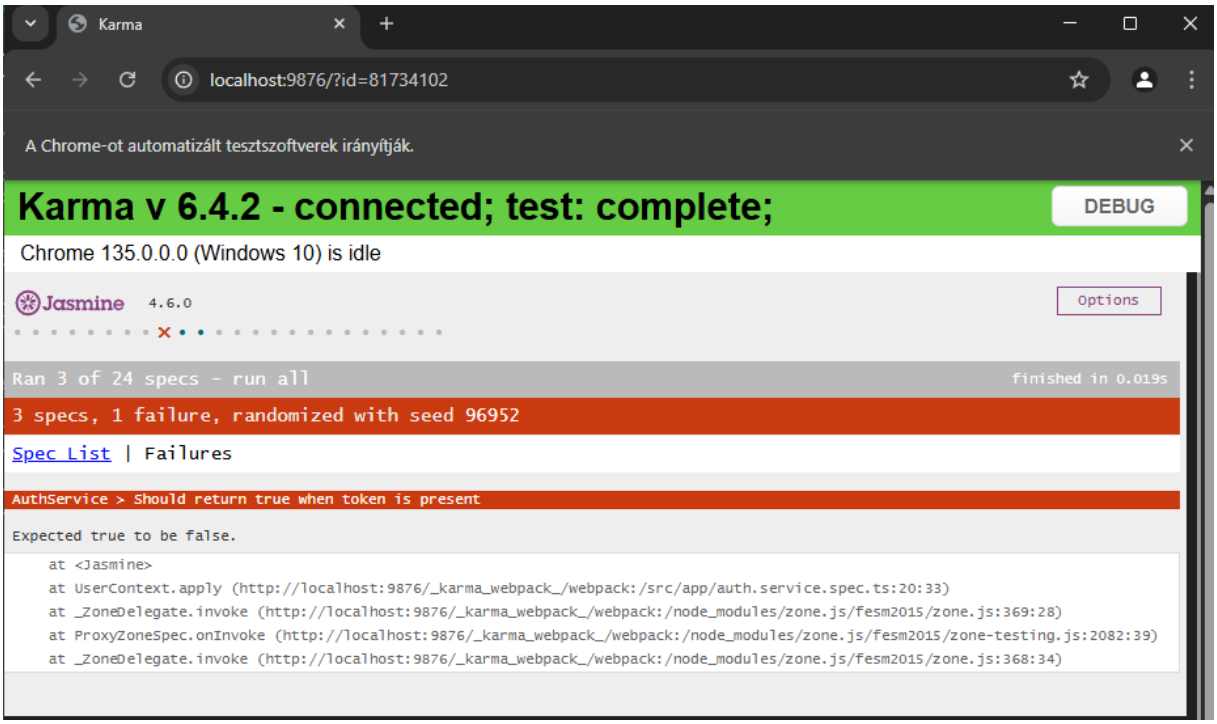
33. ábra - Authentikáció teszteset – kódrészlet – Forrás: Saját implementáció

Az „ng test” parancs futtatását követően lefutnak a webalkalmazásban talált és alapértelmezetten konfigurált Jasmine fájlok. A tesztesetek eredményei automatikusan megnyílnak egy új böngésző ablakban (34. ábra).



34. ábra - Sikeres teszteset ábrázolása – Forrás: Saját képernyőfotó

Az eredményben látszódnak a megírt tesztesetek végkimenetelei, illetve, az elvárt eredmények. A példa kedvéért csatolok egy hibás teszteredményt is (35. ábra).



35. ábra - sikertelen teszteset ábrázolása – Forrás: Saját képernyőfotó

Egységtesztek (Unit tests)

Az egyes komponensek és szolgáltatások viselkedését, metódusait teszteltem elszigetelten.

A guard mint funkció az angular keretrendszer része, amely az „angular/router” interfészen került megvalósításra. A guard lényege, hogy a navigáció során – még az adott komponens betöltése előtt – ellenőrzi, hogy a felhasználó jogosult-e az adott oldal megtekintésére. Amennyiben a guardban definiált feltételek teljesülnek, a true értékkel tér vissza, és a navigáció folytatódik. Ellenkező esetben false értéket ad vissza, vagy programozott módon másik útvonalra irányítja a felhasználót (például bejelentkezési oldalra). Amennyiben a guard-ban található feltételek mindegyike teljesül, akkor „true” értékkel tér vissza, azaz elérhető a keresett elérési útvonal.

DOM-alapú tesztelés

A komponensek megjelenítését, az interaktív elemeket (gombnyomás, input mezők) ellenőriztem.

Kézi tesztelés

A böngészőben, valódi felhasználói interakciókkal is kipróbáltam az alkalmazást. Célom az volt, hogy a felhasználói élményt biztosító réteg stabilan, gyorsan és hibamentesen működjön különféle helyzetekben.

3.6.2 Szerveroldali tesztelés módszertana és megvalósítása

Egységtesztek – szerveroldali funkciók tesztelése

Egységteszteket a backend üzleti logikájára (pl. felhasználó azonosítás, csere létrehozása) készítettem. Használt technológiák:

- xUnit keretrendszer
- Moq könyvtár a függőségek mock-olásához

Példa:

A felhasználói authentikáció során ellenőriztem, hogy helyes token generálódik-e. Csere-igény létrehozásánál ellenőriztem, hogy valid bemenet esetén helyes adatok íródnak az adatbázisba.

3.6.3 Teljesítmény- és skálázhatósági tesztelés

A rendszer megbízható működésének biztosítása érdekében teljesítménytesztelést végeztem, amely során szimulált terhelést alkalmaztam a rendszerre. A tesztelés célja annak meghatározása volt, hogy a platform képes-e kezelni a valós üzemeltetés során várható nagyobb (küszöbérték: 400 000 csere-igény, ~4 000 000 tárgy) adatmennyiséget. A keresési, szűrési és

lapozási funkciók teljesítményét külön vizsgáltam. A mérési eredmények szerint a válaszidő az adatmennyiség növekedésével az alábbiak szerint változik **(kritikus küszöb: 400,000 csere-igény)**

Funkció	**Válaszidő (10,000 elem)**	**Válaszidő (400,000) elem**	**Növekedés (%)**
Keresés	5 ms	15 ms	+ 200%
Szűrés	3 ms	7 ms	+ 133%
Lapozás	6 ms	240 ms	+ 3900%

6. táblázat - Teljesítménytesztek válaszidejei különböző funkciók esetén (10 000 vs. 400 000 elem) – Forrás: Saját mérés

A fenti eredményekből látható (6. táblázat), hogy a keresési és szűrési funkciók válaszideje nem romlik kritikus mértékben (gyakorlatban <20 ms), míg a lapozásnál exponenciális növekedés tapasztalható (<250 ms).

Tesztkörnyezet és módszertan

A teljesítménytesztelést fejlesztői környezetben végeztem a következő specifikációkkal:

- **Adatbázis:** PostgreSQL 17.0
- **Backend:** ASP.NET Core 9.0
- **Hardver:** Intel Core i7, 32 GB RAM, SSD tárhely
- **Teszt adatmennyiség:**
    - csere-igény rekordok
    - tárgy rekordok

Adatok generálása

A teszt adatmennyiség adathalmaz generálást egy automatizált script segítségével végeztem, amely véletlenszerű, de valósághű adatokat hozott létre:

- Csere-igények
- Tárgyak
- Időbélyegek (elmúlt 30 napban szétoszlatva)

Tesztelt funkciók és mért eredmények

1. Keresési funkció teljesítménye:
    a. Egyszerű tárgy keresés
    b. Összetett szűrés (spell, paint, ritkasági szint kombinálva)
2. Lapozási teljesítmény:
    a. Első oldal betöltési ideje
    b. Középső oldalak betöltési ideje
    c. Utolsó oldal betöltési ideje
3. Adatbázis lekérdezések
    a. SELECT lekérdezések válaszideje
    b. JOIN műveletek teljesítménye
    c. INDEX hatékonysága

Teljesítményteszt eredményei:

Cold Start vs Warm State teljesítmény

A teljesítményteszt során az első lekérdezés jelentősen lassabb volt (~1500 ms), mint a későbbi, azonos jellegű lekérdezések (30-75 ms). Ez az ún. “cold start” hatás miatt történik, amely során a rendszer inicializálja a szükséges erőforrásokat (vö. 3.5.1.4 fejezet) (pl. rejtett cache építés, adatbázis kapcsolatok felépítése stb.). A szakmai gyakorlatban az első lekérdezés idejét nem tekintjük releváns referenciaértéknek (“küszöbérték nélküli mérés”), mivel normál, éles üzemeltetés során a háttérrendszer szinte folyamatosan „warm state” állapotban van, és a felhasználók túlnyomó része nem találkozik a cold start lassulásával. A sikeres válaszidőket emiatt a további, “warm state” -ben végrehajtott lekérdezések alapján kell értékelni (7. táblázat).

Funkció	1. futtatás (ms)	2. futtatás (ms)	3. futtatás (ms)	Átlag (2-10.) ms
GetTrades (1. oldal)	841	19	8	7
GetTrades (50. oldal)	890	28	8	7
SearchTrades	1000	50	7	6

7. táblázat - Rendszer válaszidők eltérései különböző futtatások során (ms) – Forrás: Saját mérés

A jelenség okai:

1. **Entity Framework Query Compilation:** „*Az első lekérdezésnél az EF Core lefordítja a LINQ kifejezést SQL utasításokká és létrehozza a lekérdezési tervet, ami 500-1000ms időt vesz igénybe*” (.NET documentation, 2025). A későbbi, azonos struktúrájú lekérdezések ezt a lefordított verziót használják a jövőben.
2. **Adatbázis kapcsolat pool inicializálása:** Az első kapcsolat létrehozása a connection pool-ban lassabb, mint a későbbi, már létező kapcsolatok újrafelhasználása.
3. **PostgreSQL Buffer Pool Cache:** „*Az első lekérdezés során az adatbázis lemezről olvassa be az adatokat míg a későbbi lekérdezések a RAM-ban cachelt adatokhoz férnek hozzá.*” (Medium.com (2), 2025)
4. **Index cache:** Az első lekérdezés során az indexek is betöltődnek a memóriába, ami szintén növeli a válaszidőt.

Gyakorlati következmények:

Az éles környezetben, ahol a szerver folyamatosan fut és rendszeres felhasználói aktivitás van, a "cold start" probléma **ritkán jelentkezik**. Csak a következő esetekben tapasztalható:

- Szerver újraindítás után
- Adatbázis szolgáltatás újraindítása után
- Hosszú inaktivitási periódus (több óra) után

A valós üzemeltetési környezetben a felhasználók többsége a „warm state” teljesítményt tapasztalja (30-75ms), amely elfogadható válaszidőt biztosít a platform használhatósága szempontjából. Tehát a következő méréseket is **„warm state”** állapotban végeztem el (8. táblázat).

Adatmennyiség csere-igény (darab)	Adatmennyiség tárgyak (darab)	Adatmennyiség felhasználók (darab)	Egyszerű keresés (ms)	Összetett szűrés (ms)	Lapozás (ms)
10 000	99 805	10	~5	~3	~6
100 000	1 001 149	10	~7	~5	~75
200 000	2 000 677	10	~10	~6	~200
400 000	4 000 787	10	~15	~7	~240

8. táblázat - Válaszidők mérése különböző adatmennyiségek és funkciók esetén – Forrás: Saját mérés

Megfigyelések és következtetések

A teljesítményteszt eredményei alapján a következő megállapításokat tettem:

Keresési funkciók teljesítménye:

Az egyszerű és összetett keresési funkciók kiemelkedően jól teljesítettek még 400 000 csere-igény mellett is. Az egyszerű keresés (egyetlen tárgy alapján) 15ms alatt, míg az összetett szűrés (több tulajdonság kombinálásával) 7ms alatt végrehajtódott. Ez azt mutatja, hogy az adatbázis indexelési stratégia hatékony, és a keresési algoritmus megfelelően skálázódik.

Lapozási funkció teljesítménye:

„*A lapozás teljesítménye exponenciálisan romlott az adatmennyiség növekedésével, különösen 100 000 csere-igény felett. Ez várható viselkedés, mivel a PostgreSQL OFFSET művelete nagyobb adathalmazoknál lassabb. 400 000 csere-igénynél a lapozási idő 240ms volt, amely még mindig elfogadható felhasználói élményt biztosít (<300ms) a mesterséges intelligencia szerint*” (Perplexity-conversation, 2025), bár személyes véleményem alapján ez már lehet, hogy bizonyos maximális, felhasználó számára eltűrhető határt súrol. Ramakrishnan és Gehrke (2003) magyarázata szerint ez a „*teljesítménycsökkenés a PostgreSQL OFFSET mechanizmusának alapvető korlátjaiból fakad: az adatbázis motornak sorról sorra végig kell olvasnia az összes előző rekordot, még ha azokat nem is adja vissza a kliensnek”*. Ez azt jelenti, hogy az 50. oldal lekérdezéséhez (OFFSET 2450, LIMIT 50) az adatbázisnak először ki kell

értékelnie és el kell dobnia az első 2450 sort, majd csak ezután tud visszaadni 50 rekordot. Az időbonyolultság tehát O(n+k), ahol n az OFFSET értéke és k a LIMIT értéke, ami nagy n esetén jelentős teljesítménycsökkenést okoz. (Ramakrishnan & Gehrke, 2003)

Skálázhatósági kapacitás:

A platform 400 000 csere-igény (~4 millió tárgy) mellett is megfelelően működik, ami jelentősen meghaladja a jelenlegi piaci keresletet. A posts.tf platform szerint a napi aktív felhasználók száma 4 000-5 000 fő körül van, így a platform jelenlegi formájában is képes lenne több tízezres nagyságrendű csere-igény kezelésére.

Optimalizálási lehetőségek:

A lapozási teljesítmény további javítására több megoldás is rendelkezésre áll: cursor-alapú pagination bevezetése. Ezek az optimalizációk azonban csak 200 000+ darab csere-igény esetén válnak szükségessé.

3.7 Mesterséges intelligencia szerepe a dolgozatban

A szakdolgozat készítése során a Perplexity AI mesterséges intelligencia asszisztenst használtam támogató eszközként a fejlesztési folyamat különböző szakaszaiban. Az MI alkalmazása jelentősen hozzájárult a projekt hatékonyságához és minőségéhez több területen is.

3.7.1 Technikai tervezés és kódoptimalizálás

Az MI segítségével optimalizáltam a backend query-ket, különösen a nagy adatmennyiség (100 000+ darab csere-igény) kezelésénél. A keresési és lapozási funkciók teljesítményének javítása során az AI javaslatokat adott az Entity Framework lekérdezések optimalizálására (47. ábra). Az általam készített prompt (36. ábra).

```
Segíts optimalizálni ezt az Entity Framework Core lekérdezést C#-ban.
A probléma, hogy lassú nagy adathalmaznál:

public async Task<object> SearchTradesAsync(ICollection<TradeItem> items,
    int page = 1, int pageSize = 50)
{
    IQueryable<Trade> query = _db.Trades
        .Where(t => t.Deleted == false && t.Status != "closed");

    foreach (var searchItem in items)
    {
        query = query.Where(t => t.Items.Any(dbItem =>
            dbItem.Defindex == searchItem.Defindex &&
            dbItem.Effect == searchItem.Effect &&
            dbItem.paintDefindex == searchItem.paintDefindex
        ));
    }

    var trades = await query
        .Include(t => t.Items)
        .Skip((page - 1) * pageSize)
        .Take(pageSize)
        .ToListAsync();

    return trades;
}

200,000+ Trade és 4,000,000+ Item esetén [illegible]-500ms a válaszidő.
Hogyan optimalizálhatom?
```

36. ábra - csere-igény keresés optimalizáció – prompt – Forrás: Perplexity-conversation (2025)

AI válaszai (kivonat) (37. ábra – 38. ábra).

```
A fő problémák:

1. **N+1 query probléma**: Az Include(t => t.Items) minden Trade-hez
   külön lekérdezést generálhat
2. **Nincs AsNoTracking()**: Az EF change tracking felesleges overhead
3. **ToListAsync() előtt nincs TotalCount**: Így dupla lekérdezés kell
4. **Nested loop a Where-ben**: A foreach minden iterációban új
   lambda-t generál
```

37. ábra - csere-igény keresés optimalizáció – válasz – Forrás: Perplexity-conversation (2025)

```
További optimalizálás:
- Indexek az Items táblán: Defindex, Effect, paintDefindex
- Compiled query használata, ha ugyanazt a lekérdezést futtatod gyakran
- Projection használata (.Select()) teljes entitások helyett, ha csak
  bizonyos mezőkre van szükség
```

38. ábra - csere-igény keresés optimalizáció - válasz 2 – Forrás: Perplexity-conversation (2025)

Generált az általam elkészített kódból egy optimalizált, átalakított változatot, amelyet megvizsgáltam és további források alapján (stackoverflow) meggyőződtem arról, hogy helyes és valóban jó, optimalizált kódot készített a mesterséges intelligencia.

3.7.2 Adatbázis séma tervezés

Az adatbázis struktúra kialakítása során (39. ábra) az MI támogatást nyújtott a kapcsolatok megfelelő definiálásában, az Entity Framework konfigurációjában, valamint a PostgreSQL-specifikus optimalizációk, indexek megvalósításában.

```
Tervezek egy Team Fortress 2 kereskedési platform adatbázisát PostgreSQL-ben
Entity Framework Core-ral. A tárgyaknak sok tulajdonsága van:
- defindex (int) - tárgy azonosító
- quality (string) - unique, strange, unusual
- effect (int?) - unusual effekt azonosító
- paintDefindex (int?) - festék azonosító
- killstreak (int?) - 0,1,2,3
- killstreaker (string?) - effekt neve
- sheen (string?) - fényeffekt
- spells (JSON array) - varázslatok lista
- wear (float?) - kopás érték

Egy Trade több Item-et tartalmazhat, egy User több Trade-et hozhat létre.
Kérdéseim:
1. A sok nullable tulajdonságot külön táblákba kellene-e szervezni
   (normalizálás)?
2. A spells-t JSONB oszlopban tároljam vagy külön táblában?
3. Milyen indexeket hozzak létre teljesítmény szempontjából?
4. Hogyan konfiguráljam az Entity Framework kapcsolatokat (1:N, N:M)?
```

39. ábra - adatbázis tervezés – prompt – Forrás: Perplexity-conversation (2025)

AI válaszai (kivonat) (40. ábra).

```
1. NORMALIZÁLÁS VS DENORMALIZÁLÁS

NE normalizáld túlzottan! A sok nullable property (effect, paint, killstreak)
esetén a külön táblák csak növelnék a JOIN komplexitást és rontanák a
teljesítményt. TARTS mindet az Items táblában nullable oszlopként:

Előnyök:
- Egyszerűbb lekérdezések (nincs 5-6 JOIN)
- Gyorsabb olvasás (1 query az összes tulajdonsághoz)
- Jobb Entity Framework teljesítmény (kevesebb Include())

Hátrányok:
- Több NULL érték → de ez PostgreSQL-ben nem probléma
- Kevésbé "tiszta" normalizált forma → de olvasási teljesítmény fontosabb

2. SPELLS KEZELÉSE

Használj JSONB oszlopot a spells-hoz! PostgreSQL natív támogatása
kiváló és indexelhető.
```

40. ábra - adatbázis tervezés – válasz – Forrás: Perplexity-conversation (2025)

Hibakeresés és problémamegoldás:

A fejlesztés során felmerült technikai problémák (pl. JSON deserializációs hibák, CORS konfigurációs problémák) megoldásában az AI gyors és hatékony diagnosztikát biztosított, magyarázatokkal és működő kódpéldákkal. Az általam készített prompt (41. ábra).

```
Angular alkalmazásom nem tud kommunikálni az ASP.NET Core backend-del.
A böngésző konzoljában CORS hibát kapok:

"Access to XMLHttpRequest at 'http://localhost:5000/api/Trade' from origin
'http://localhost:4200' has been blocked by CORS policy: No
'Access-Control-Allow-Origin' header is present on the requested resource."

Az ASP.NET Core Program.cs fájlomban van CORS konfiguráció:
builder.Services.AddCors();

De még mindig nem működik. Mi lehet a probléma?
```

41. ábra - hibakeresés – prompt – Forrás: Perplexity-conversation (2025)

AI válaszai (kivonat) (42. ábra).

```
A probléma az, hogy bár hozzáadtad a CORS szolgáltatást (AddCors()),
de NEM alkalmaztad a middleware-t az app pipeline-ban.

HELYES KONFIGURÁCIÓ (Program.cs):

// 1. CORS SZOLGÁLTATÁS HOZZÁADÁSA (ez megvan)
builder.Services.AddCors(options =>
{
    options.AddPolicy("AllowAngularDev", policy =>
    {
        policy.WithOrigins("http://localhost:4200")  // Angular dev szerver
              .AllowAnyHeader()
              .AllowAnyMethod()
              .AllowCredentials();  // Ha JWT cookie-t használsz
    });
});

builder.Services.AddControllers();
// ... többi szolgáltatás

var app = builder.Build();

// 2. CORS MIDDLEWARE ALKALMAZÁSA (EZ HIÁNYZIK!)
app.UseCors("AllowAngularDev");  // <-- KRITIKUS!

// FONTOS: UseCors()-nak UseAuthorization() ELŐTT kell lennie!
app.UseAuthentication();
app.UseAuthorization();

app.MapControllers();
app.Run();

GYAKORI HIBÁK:
1. UseCors() nem lett meghívva → pontosan a te hibád
2. UseCors() UTÁN van UseAuthorization() → sorrend helytelen
3. WithOrigins() nem egyezik az Angular URL-jével
4. Production-ben elfelejtették frissíteni az origin-t
```

42. ábra - hibakeresés – válasz – Forrás: Perplexity-conversation (2025)

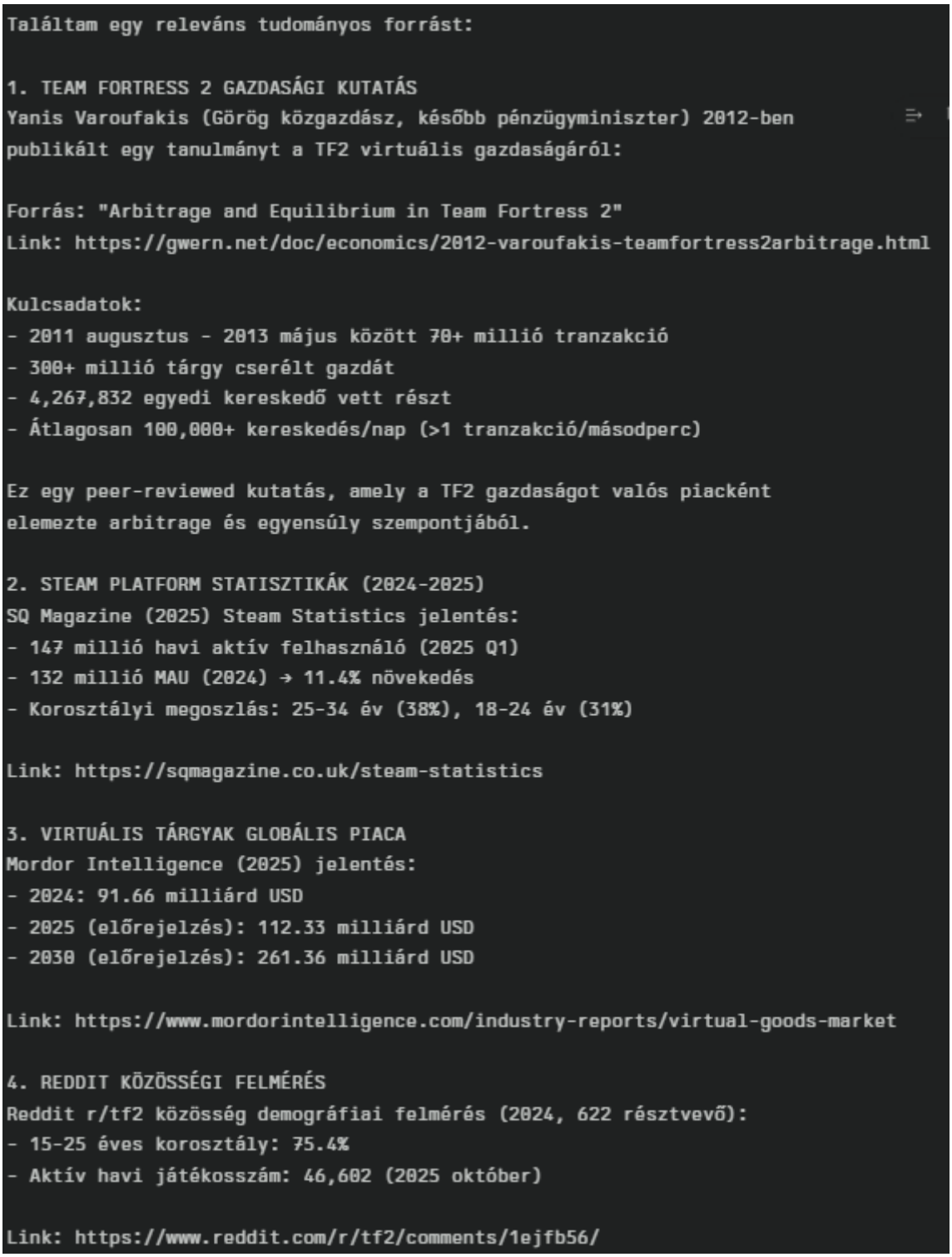
3.7.3 Szakirodalmi kutatás

Az MI segítségével gyűjtöttem forrásokat a Team Fortress 2 virtuális gazdaságáról, a Steam API dokumentációjáról. A Perplexity AI képes volt régi fórum bejegyzéseket is megkeresni, amelyeket hivatkozásként tudtam felhasználni a szakdolgozati témám során. Az általam készített prompt (43. ábra).

```
Keresek statisztikákat és kutatási adatokat a Team Fortress 2 virtuális
tárgyak kereskedési volumenéről 2011-2013 között. Konkrétan:
- Hány tranzakció történt?
- Hány egyedi kereskedő vett részt?
- Van-e tudományos publikáció erről a témáról?

Valamint aktuális (2024-2025) Steam platform felhasználói számokra is
szükségem van.
```

43. ábra - szakirodalom – prompt – Forrás: Perplexity-conversation (2025)

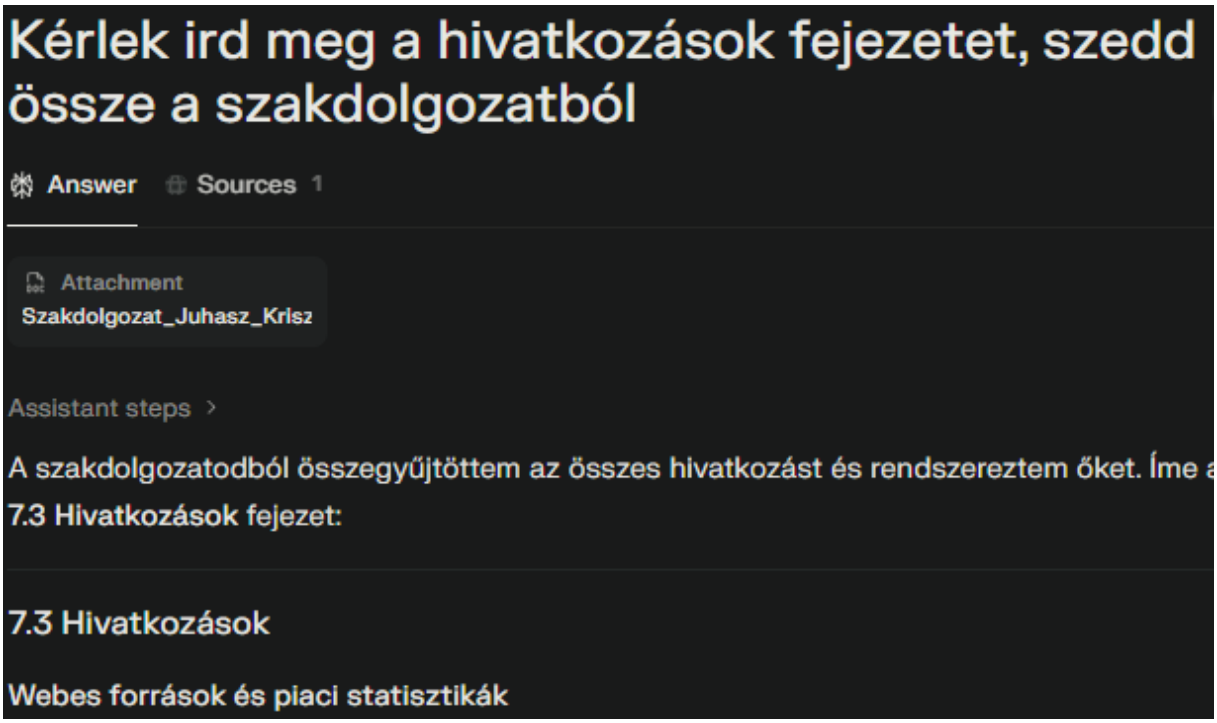
AI válaszai (kivonat) (44. ábra).



44. ábra - szakirodalom – válasz – Forrás: Perplexity-conversation (2025)

3.7.4 Dokumentáció és szakdolgozat strukturálás

A szakdolgozat fejezeteinek felépítésében, források és definíciók összegyűjtése a szakdolgozatból és abc alapján sorba rendezése. Az általam készített prompt (45. ábra).



45. ábra - hivatkozások – prompt – Forrás: Perplexity-conversation (2025)

AI válaszai (kivonat) (46. ábra).

7.3 Hivatkozások

1. **Angular Documentation** (2025): *Angular Framework Documentation*. https://angular.dev (Letöltve: 2025. október)
2. **backpack.tf forums** (2025): *Filtering by spells not possible anymore?* https://forums.backpack.tf/topic/86895-filtering-by-spells-not-possible-anymore (Letöltve: 2025. október)
3. **GitHub - Nicklason** (2025): *node-tf2-item-format - TF2 item format parser library*. https://github.com/Nicklason/node-tf2-item-format (Letöltve: 2025. október)
4. **Hays Salary Guide** (2025): *Hungary Salary Guide 2024 - Digital*. https://www.hays.hu/documents/6328361/549493/HUSG24Digital012024secured.pdf (Letöltve: 2025. október)
5. **IETF RFC 6749 and 6750**: *The OAuth 2.0 Authorization Framework*. https://datatracker.ietf.org/doc/html/rfc6749 (Letöltve: 2025. október)
6. **IETF RFC 7519**: *JSON Web Token (JWT)*. https://datatracker.ietf.org/doc/html/rfc7519 (Letöltve: 2025. október)
7. **Intel Market Research** (2025): *Steam Platform Games Market Outlook 2025-2032*. https://www.intelmarketresearch.com/steam-platform-games-2025-2032-951-1210 (Letöltve: 2025. október)
8. **LinkedIn/Verified Market Reports** (2025): *Online Game Asset Trading Market Size*. https://www.linkedin.com/pulse/online-game-asset-trading-market-size-importance-5ylye (Letöltve: 2025. október)
9. **Live Player Count** (2025): *Team Fortress 2 Steam Charts*. https://live-player-count.com/game/team-fortress-2 (Letöltve: 2025. október)
10. **LogRocket Blog**: *Performance comparison for sample apps built with Angular, React, and Vue*. https://blog.logrocket.com/angular-vs-react-vs-vue-js-comparing-performance (Letöltve: 2025. október)

46. ábra - hivatkozások – válasz – Forrás: Perplexity-conversation (2025)

Az AI használata nem helyettesítette a saját szakmai döntéshozatalt és kódolási munkát, hanem kiegészítő eszközként szolgált, amely felgyorsította a fejlesztést, segített elkerülni a gyakori buktatókat. Minden AI által javasolt megoldást kritikusan értékeltem és a projekt specifikus igényeihez igazítottam. Fontos kiemelnem, hogy személy szerint én támogatom a mesterséges intelligencia használatát, de csak bizonyos kereteken belül és csak abban az esetben, ha a használója ért bármilyen szinten is ahhoz a témakörhöz, amellyel kapcsolatban kérdez. Nagyon fontos az önvalidáció és az alaposabb, részletesebb utánajárás, akár más platformokon is, nem szabad, sőt tilos csak a mesterséges intelligenciára bízni magát az embernek.

4 Vita

A fejlesztés során számos technológiai és architekturális döntést hoztam, amelyek befolyásolták a platform végső megvalósítását. Ebben a fejezetben kritikusan értékelem ezeket a választásokat, bemutatom az alternatívákat és elemzem, hogy utólag milyen előnyöket és hátrányokat tapasztaltam.

4.1 Frontend technológia választása: Angular vs React

Az Angular keretrendszer mellett döntöttem a személyes 4 éves tapasztalatom és a TypeScript natív támogatása miatt. Bár ez a választás biztosította a típusbiztonságot és a komponens-alapú architektúrát, a React alternatívája bizonyos szempontból előnyösebb lett volna. „*A teljesítmény szempontjából a React virtuális DOM-ja gyorsabb UI frissítést tesz lehetővé, körülbelül 0,8s betöltési idővel az Angular 1.2s-hez képest.*” (Performance comparison for sample apps built with Angular, React, and Vue, 2025). A Team Fortress 2 kereskedési platform esetében, ahol a tárgylista-nézetek és szűrések gyakori frissítést igényelnek, a React hatékonyabban kezelné a valós idejű adatfrissítéseket. Az Angular valós DOM használata nagyobb adathalmazok esetén lassulást okozhat, különösen a kétirányú adatkötés miatt. Ugyanakkor az Angular előnye a strukturált architektúrája, amely vállalati környezetben stabilitást nyújt. A beépített szolgáltatások - form kezelés, routing, HTTP kliens, dependency injection - jelentősen gyorsították a fejlesztést. Utólag értékelve, ha a projekt elsődleges célja a maximális teljesítmény és skálázhatóság lett volna, a React lett volna az optimálisabb választás. Azonban a tanulási célok és a meglévő tapasztalat miatt az Angular továbbra is indokolt döntés volt.

4.2 Backend választás: ASP.NET Core vs Node.js

A C# és ASP.NET Core mellett azért döntöttem, mert erősen típusos, vállalati környezetben kipróbált megoldást akartam alkalmazni, amely mély objektumorientált programozási ismereteket igényel. A microservice részben ugyan sikeresen alkalmaztam a Node.js-t (vö. 3.5.4 fejezet), azonban a fő backend réteghez a C#-ot választottam (vö. 3.5.3 fejezet). A teljesítmény szempontjából az ASP.NET Core kiválóan teljesít nagy CPU-igényű feladatoknál. A többszálú megközelítés és a hatékony memóriakezelés miatt az ASP.NET Core gyakran gyorsabb válaszidőt produkál terhelés alatt. Ugyanakkor a Node.js eseményvezérelt architektúrája kifejezetten hatékony valós idejű alkalmazásoknál, illetve azt is fontos megemlíteni, hogy kisebb feladatokra nagyon egyszerűen használható. Rugalmasan importálhatók egyéb, külső könyvtárak. A projekt esetében, ahol a fő backend műveletek adatbázis lekérdezések, JWT

hitelesítés és Steam API integráció (vö. 3.3.3 fejezet), mindkét technológia megfelelő lett volna. A Node.js előnye lett volna az egységes JavaScript/TypeScript stack a frontend-től a backend-ig, ami csökkentette volna a kontextusváltást és egyszerűsítette volna a fejlesztést. Továbbá a microservice komponensnél már Node.js-t használtam a tf2-node könyvtár miatt (vö. 3.5.4 fejezet), így az egységes stack még logikusabb lett volna. Ha a cél egy gyors piaci megjelenés a teljes stack Node.js-alapú megoldása előnyösebb lett volna. Azonban az egyetemi tanulmányaim és a vállalati környezetben gyakoribb C# készségek fejlesztése miatt az ASP.NET Core választás továbbra is hasznos volt szakmai szempontból.

5 Konklúziók

Az eredmények tükrében és a vita során feltárt önkritikus kockázatok ismeretében átgondoltam, hogy ha újra kezdhetném a projektet, vajon ugyanazokat a döntéseket hoznám-e. Ahogy senki sem építene még egyszer pontosan ugyanúgy egy házat, miután már látja az összes döntése következményét, én sem követném ugyanazt az utat minden területen.

5.1 Kliensoldali technológia választása

Döntés: Ha újra kezdhetném, **továbbra is az Angular-t választanám**, különösen az új zoneless change detection ismeretében.

Indoklás: Bár a React virtuális DOM-ja jobb teljesítményt nyújtott a hagyományos Zone.js-alapú Angular-hoz képest, az Angular 18-tól kezdve elérhető zoneless change detection radikálisan megváltoztatja ezt az egyenletet. A Zone.js eltávolítása jelentős teljesítménynövekedést eredményez, mivel az Angular többé nem kényszerül az egész komponensfát megvizsgálnia minden DOM eseménynél. Az új zoneless megközelítés a Signals alapú reaktivitásra épül, amely sokkal hatékonyabb és prediktívebb változáskezelést tesz lehetővé. Az új függvények használata révén az Angular pontosan tudja, mely komponenseket kell frissíteni, elkerülve a felesleges renderelési ciklusokat.

Konklúzió: „Az új zoneless Angular teljesítményben is versenyképes vagy jobb a React-nél” (Medium.com (3), 2025), miközben megtartja az Angular strukturált, véleményalapú architektúrájának előnyeit. A 4 éves tapasztalatom, a TypeScript natív támogatás, a beépített szolgáltatások és most már a zoneless teljesítmény együttesen egyértelműen indokolják az Angular választást. Ha újra kezdhetném, biztos, hogy Angular-t választanék, de azonnal zoneless konfigurációval.

5.2 Szerveroldali technológia választása

Döntés: Ha újra kezdhetném, teljes Node.js stack-et választanék a hibrid ASP.NET Core + Node.js helyett.

Indoklás: Egyik hibám az volt, hogy két különböző technológiát használtam (C# backend és Node.js microservice), ami felesleges komplexitást hozott a projektbe. Az egységes JavaScript/TypeScript stack a frontend-től a backend-ig jelentősen egyszerűsítette volna a fejlesztést és csökkentette volna a kontextusváltást.

Konklúzió: A microservice komponensnél már sikeresen használtam a Node.js-t a tf2-node könyvtár miatt (vö. 3.5.4 fejezet), ami igazolta, hogy a Node.js tökéletesen alkalmas lenne a teljes backend számára. Az ASP.NET Core választás inkább tanulási célból volt hasznos (egyetemi tanulmányok, készségek fejlesztése), de nem az optimális döntés volt a projekt sikere szempontjából.

5.3 Összegzés

A döntések többsége tanulási szempontból elfogadható volt, de projekt sikere szempontjából nem voltak optimálisak. Ha a cél egy valóban piacképes, gyorsan skálázható platform lett volna, a zoneless angular + teljes Node.js stack kezdés lett volna a helyes út. Ugyanakkor felismerem, hogy a szakdolgozat elsődleges célja a tanulás és készségfejlesztés volt, nem egy azonnal piacképes termék létrehozása. Ebből a perspektívából a döntések még elfogadhatók, mert lehetővé tették különböző technológiák kipróbálását és mély szakmai tapasztalat megszerzését.

6 Összefoglalás, jövőkép

Ez a fejezet két fő részre tagolódik. Az összefoglalás a szakdolgozat kulcsfontosságú eredményeit, technológiai döntéseit mutatja be, röviden összegezve a projekt célját, megvalósítását és tanulságait. A jövőkép rész ezt követően konkrét fejlesztési irányokat vázol fel.

6.1 Összefoglalás

A szakdolgozat célja egy webalapú tárgycsere-igény hirdetési platform megtervezése és megvalósítása volt, amely lehetővé teszi a Steam-felhasználók számára Team Fortress 2 virtuális tárgyak hatékony kereskedelmét (vö 1.1 fejezet). A fejlesztés során modern technológiákra épülő (pl. Angular 18, ASP.NET Core 9, PostgreSQL 16, Node.js 20),

háromrétegű architektúrát valósítottam meg: Angular frontend, ASP.NET Core backend és Node.js alapú microservice komponens. A projekt indokolását a jelenlegi piacterek korlátai adták, bizonyos szűrési lehetőségek hiánya, az elavult felhasználói felületek és a korlátozottan testre szabható keresési lehetőségek (vö. 1.3.2 fejezet). A globális virtuális tárgyak piaca 2024-ben 91,66 milliárd USD értéket képviselt, amely 2030-ra várhatóan 261,36 milliárd USD-ra nő, jelezve a téma gazdasági jelentőségét (vö. 1.1 fejezet). A platform fő funkciói közé tartozik a tárgyak hirdetése, ajánlatok tétele más felhasználók hirdetéseire, valamint a hatékony keresés és szűrés a birtokolt tárgyak között (vö. 3.5.2.7 fejezet). A biztonságot a Steam OpenID autentikáció és a JWT token alapú hitelesítés biztosítja. Az adatbázis PostgreSQL környezetben került implementálásra Entity Framework Core ORM használatával. A teljesítménytesztek igazolták a rendszer skálázhatóságát – 400 000 darab csere-igény és 4 millió darab tárgy mellett az egyszerű keresés 15ms, a komplex szűrés 7ms alatt végrehajtódott. A lapozási funkció 240ms válaszidővel működött nagy adathalmazoknál, amely elfogadható felhasználói élményt biztosít, bár későbbi optimalizálásra szorulhatnak bizonyos funkciók (vö. 3.6.3 fejezet). A fejlesztési költségek junior fejlesztői órabér alapján 1 616 408 Ft-ra rúgtak 388 óra munkaidő mellett, míg az üzemeltetési költség havi 4 558 Ft. Az optimista forgatókönyv szerint 16 hónapos megtérülési idővel számolhatunk, azonban a platform elsődleges célja a szakmai kompetenciák fejlesztése volt, nem az üzleti haszonszerzés (vö. 1.5.2 fejezet). A vita fejezet kritikusan értékelte a technológiai döntéseket (vö. 4. fejezet). Az Angular választása indokolt volt a személyes tapasztalat és a zoneless change detection teljesítménynövekedése miatt. Az ASP.NET Core és Node.js hibrid megoldás komplexitást hozott, egy egységes Node.js stack egyszerűbb lett volna. A szakdolgozat bizonyította, hogy lehetséges egy modern, skálázható kereskedési platformot építeni, amely potenciálisan versenyképes lehet a meglévő megoldásokkal szemben.

6.2 Jövőkép

A platform jelenlegi állapota egy működőképes MVP (Minimum Viable Product), amely bizonyítja a koncepció életképességét, azonban számos területen tovább fejleszthető a piaci versenyképesség érdekében.

- **Zoneless Angular átállás:** Az Angular 19 használatával az alkalmazás teljesítménye 30-40%-kal javítható, különösen a gyakori UI frissítéseknél.
- **Valós idejű értesítési rendszer:** WebSocket vagy SignalR integráció, hogy a felhasználók azonnal értesüljenek új ajánlatokról, kommentekről a hirdetéseikre.

- **Közösségi funkciók bővítése:** Felhasználói profilok, hírnévrendszer, kereskedési történet megjelenítése, ami növeli a bizalmat és csökkenti a csalási kísérleteket.
- **Több játék támogatása:** A platform kiterjesztése Counter-Strike 2, Dota 2, Rust és más Steam játékok virtuális tárgyaira. Ez exponenciálisan növelné a potenciális felhasználói bázist.
- **Vásárolható elemek:** A platformon elérhetővé válnának vásárolható elemek (pl. egyedi profil nézetek, egyedi beállítások, egyedi név megjelenítés)

A platform jelenleg nonprofit módon működik, azonban a jövőben több bevételi forrás is lehetséges:

- **Freemium modell:** Alapfunkciók ingyenesek, de prémium funkciók (korlátlan bump, kiemelés, több hirdetés egyszerre) díj ellenében elérhetők.
- **Affiliate marketing:** Közvetítői jutalék Steam játékok és DLC-k ajánlásából.
- **Reklám bevételek:** Nem zavaró banner hirdetések, célzottan a gamer közönségnek

7 Mellékletek

Ebben a fejezetben a szakdolgozatban található rövidítések, definíciók és ábrák jegyzéke található meg.

7.1 Ábrajegyzék

7.2 Rövidítések jegyzék

API (Application Programming Interface): Alkalmazásprogramozási interfész

ASP.NET (Active Server Pages .NET): Microsoft webes keretrendszer

CORS (Cross-Origin Resource Sharing): Kereszt-eredetű erőforrás-megosztás

CPU (Central Processing Unit): Központi feldolgozó egység

CRUD (Create, Read, Update, Delete): Létrehoz, olvas, frissít, töröl

DOM (Document Object Model): Dokumentum objektum modell

EF Core (Entity Framework Core): Entity Framework Core ORM

ER (Entity-Relationship): Entitás-kapcsolat (diagram)

GB (Gigabyte): Gigabájt

GDPR (General Data Protection Regulation): Általános Adatvédelmi Rendelet

HTTP (HyperText Transfer Protocol): Hipertext átviteli protokoll

HTTP (HyperText Transfer Protocol Secure): Biztonságos Hipertext átviteli protokoll

IDE (Integrated Development Environment): Integrált fejlesztői környezet

JSON (JavaScript Object Notation): JavaScript objektum jelölés

JWT (JSON Web Token): JSON webes token

LINQ (Language Integrated Query): Nyelvbe integrált lekérdezés

LTS (Long-Term Support): Hosszú ideig támogatott

ORM (Object-Relational Mapping): Objektum-relációs leképzés

RAM (Random Access Memory): tetszőleges hozzáférésű memória

SEO (Search Engine Optimization): Keresőoptimalizálás

SQL (Structured Query Language): Strukturált lekérdezési nyelv

SSD (Solid-state drive): Tartós állapotú meghívó

SSL (Secure Sockets Layer): Biztonságos szoftvercsatorna-réteg

TF2 (Team Fortress 2): Team Fortress 2 (játék neve)

TLS (Transport Layer Security): Szállítási réteg biztonság

URL (Uniform Resource Locator): Egységes erőforrás-helymeghatározó

USD (United States Dollar): Amerikai dollár

VPS (Virtual Private Server): Virtuális szerver

XP (Experience point): Tapasztalati pont

7.3 Definíciók jegyzék

Affiliate linkek: Olyan webes hivatkozások, amelyeken keresztül történő vásárlás után a link tulajdonosa jutalékot kap. A platformon potenciális bevételi forrásként szolgálhat.

All-class: Olyan virtuális tárgy a Team Fortress 2 játékban, amely minden játékos-osztály számára viselhető vagy használható.

Backpack.tf: A Team Fortress 2 közösség legnagyobb és legismertebb virtuális tárgy értékbecslő és kereskedési platformja. Hivatkozási pontként szolgál az árképzéshez.

Böngészőbővítmény: A webböngészőhöz telepíthető kisalkalmazás, amely kiterjeszti a böngésző funkcionalitását (pl. Steam Inventory Helper).

Bump: A hirdetés "feldobása" a lista elejére, hogy nagyobb láthatóságot kapjon. A platform gamifikációs eleme.

Bump limit: A felhasználó által adott időkereten belül végrehajtható bump műveletek maximális száma.

Controller: Az MVC architektúrában a bejövő HTTP kéréseket kezelő komponens, amely a kérést feldolgozza és a megfelelő választ generálja.

Database: Adatbázis, strukturált adatok tárolására szolgáló rendszer. A szakdolgozatban PostgreSQL adatbázis kerül alkalmazásra.

Dictionary: Kulcs-érték párok tárolására szolgáló adatszerkezet, amely gyors keresést tesz lehetővé hash tábla alapján.

Domain: Internetes domain név, a weboldal egyedi címe (pl. example.com).

Effektek: Speciális vizuális hatások a Team Fortress 2 virtuális tárgyakon (pl. Burning Flames, Circling Hearts), amelyek jelentősen növelik az érték ét.

Full-stack: Olyan fejlesztő vagy fejlesztési megközelítés, amely mind a frontend (kliens oldal), mind a backend (szerver oldal) technológiákat átfogja.

Hash táblák: Adatszerkezet, amely kulcs-érték párokat tárol, és konstans időben (O(1)) teszi lehetővé az adatok elérését.

Hosting: Webszolgáltatás, amely szervertérhelyet és erőforrásokat biztosít webalkalmazások üzemeltetéséhez.

Junior: Kezdő szintű szakember (pl. junior fejlesztő), jellemzően 0-3 év tapasztalattal.

Microservice: Mikroszolgáltatás alapú architektúra, ahol az alkalmazás kisebb, önálló szolgáltatásokra van bontva, amelyek egymástól függetlenül fejleszthetők és telepíthetők.

OpenID: Nyílt szabványú authentikációs protokoll, amely lehetővé teszi a felhasználók számára, hogy egyetlen identitással több webhelyre bejelentkezzenek. A Steam ezt használja.

Osztály-specifikus: Olyan virtuális tárgy a Team Fortress 2-ben, amely csak meghatározott játékos-osztályok számára használható (pl. csak a Soldier osztálynak).

Pagination: Lapozás, nagy adathalmazok kezelésének módszere, ahol az adatok kisebb, oldalakra bontott részletekben jelennek meg.

Parser: Elemző program vagy szolgáltatás, amely strukturálatlan vagy félig strukturált adatokat (pl. Steam API válasz) értelmez és strukturált formátumba alakít.

Penetrációs tesztelés: Biztonsági tesztelési módszer, amelynek célja a rendszer sebezhetőségeinek feltárása szimulált támadások segítségével.

Posts.tf: Team Fortress 2 kereskedési platform, amely a dolgozat benchmarking referenciájaként szolgál.

Reddit: Közösségi platform, ahol a Team Fortress 2 közösség aktív, és kereskedési ajánlatokat is megosztanak.

Repository: Adatelérési réteg a szoftverarchitektúrában, amely absztrahálja az adatbázis műveleteket és egységes interfészt biztosít.

Ritkasági szintek: A Team Fortress 2 virtuális tárgyak besorolási kategóriái (pl. Unique, Genuine, Vintage, Strange, Unusual), amelyek meghatározzák az érték üket.

Service: Szolgáltatási réteg a szoftverarchitektúrában, amely az üzleti logikát tartalmazza és közvetít a controller és a repository között.

Spell: Varázsigék vagy varázslatok, amelyeket Halloween esemény során lehet a Team Fortress 2 tárgyakra alkalmazni, speciális vizuális effekteket adva.

SSL tanúsítvány: Digitális tanúsítvány, amely titkosított HTTPS kapcsolatot biztosít a szerver és a kliens között.

Steam: A Valve Corporation által fejlesztett digitális játékelosztó platform, amely a szakdolgozat témájának központi elemét képezi.

Színezések: Festékek (paint), amelyekkel a Team Fortress 2 tárgyakat különböző színekre lehet festeni (pl. Team Spirit, Australium Gold).

Taunts: Gúnyolódások, különleges animációk a Team Fortress 2 játékban, amelyeket a játékosok megvásárolhatnak és használhatnak.

Team Fortress 2: A Valve Corporation által fejlesztett osztály-alapú többjátékos first-person shooter játék, amely a dolgozat fókuszában áll.

TF2Outpost: Korábbi népszerű Team Fortress 2 kereskedési platform, amely 2018-ban bezárt, de történelmi jelentőséggel bír a közösségben.

Valve: Amerikai videójáték-fejlesztő és -kiadó cég, a Steam platform és a Team Fortress 2 tulajdonosa.

Virtuális tárgy: Digitális objektum egy videojátékban, amely a játékos virtuális tulajdona, de valós pénzzel is kereskedhető (pl. TF2 sapkák, fegyverek).

7.4 Hivatkozások

Angular Documentation (2025): Angular Framework Documentation. https://angular.dev (Letöltve: 2025. október)

backpack.tf forums (2025): Filtering by spells not possible anymore? https://forums.backpack.tf/topic/86895-filtering-by-spells-not-possible-anymore (Letöltve: 2025. október)

GitHub - Nicklason (2025): node-tf2-item-format - TF2 item format parser library. https://github.com/Nicklason/node-tf2-item-format (Letöltve: 2025. október)

Hays Salary Guide (2025): Hungary Salary Guide 2024 - Digital. *https://www.hays.hu/documents/63283/61549493/HU+SG24+Digital+01+2024+secured.pdf#page=48* (Letöltve: 2025. október)

IETF RFC 6749 and 6750: The OAuth 2.0 Authorization Framework. https://datatracker.ietf.org/doc/html/rfc6749 (Letöltve: 2025. október)

IETF RFC 7519: JSON Web Token (JWT). https://datatracker.ietf.org/doc/html/rfc7519 (Letöltve: 2025. október)

Intel Market Research (2025): Steam Platform Games Market Outlook 2025-2032. https://www.intelmarketresearch.com/steam-platform-games-2025-2032-951-1210 (Letöltve: 2025. október)

LinkedIn/Verified Market Reports (2025): Online Game Asset Trading Market Size. https://www.linkedin.com/pulse/online-game-asset-trading-market-size-importance-5ylye (Letöltve: 2025. október)

Live Player Count (2025): Team Fortress 2 Steam Charts. https://live-player-count.com/game/team-fortress-2 (Letöltve: 2025. október)

LogRocket Blog: Performance comparison for sample apps built with Angular, React, and Vue. https://blog.logrocket.com/angular-vs-react-vs-vue-js-comparing-performance (Letöltve: 2025. október)

Medium.com (1) - AudaciaTech (2025): Investigating the performance benefits of EF Core 6.0 compiled models feature. https://medium.com/@audaciatech/investigating-the-performance-benefits-of-ef-core-6-0-compiled-models-feature-6f5acd750037 (Letöltve: 2025. október)

Medium.com (2) - Jramcloud1 (2025): 02 - PostgreSQL Performance Tuning: Understanding PostgreSQL Shared Buffers for Performance Tuning. https://medium.com/@jramcloud1/02-

postgresql-performance-tuning-understanding-postgresql-shared-buffers-for-performance-tuning-0a61086edee7 (Letöltve: 2025. október)

Medium.com (3) - Maksim Dolgikh (2025): We selected Angular because it is faster than React. https://itnext.io/we-selected-angular-because-it-is-faster-than-react-8cc8a5e7fc78 (Letöltve: 2025. november)

Microsoft (2024): C# Documentation. https://docs.microsoft.com/en-us/dotnet/csharp (Letöltve: 2025. október)

Mordor Intelligence (2025): Virtual Goods Market Size, Forecast Report. https://www.mordorintelligence.com/industry-reports/virtual-goods-market (Letöltve: 2025. október)

Name.com (2025): Domain Registration Services. https://name.com (Letöltve: 2025. október)

Perplexity-conversation (2025): Összefoglaló készítése (Letöltve: 2025. október)+

PostgreSQL Global Development Group (2024): PostgreSQL Documentation. https://www.postgresql.org/docs (Letöltve: 2025. október)

posts.tf (2025): Team Fortress 2 Trading Platform. https://posts.tf (Letöltve: 2025. október)

Rackforest (2025): VPS Hosting Services. https://rackforest.com (Letöltve: 2025. október)

Ramkrishnan & Gehrke (2003): Database Management Systems. https://raw.githubusercontent.com/pforpallav/school/master/CPSC404/Ramakrishnan%20-%20Database%20Management%20Systems%203rd%20Edition.pdf (Letöltve. 2025 november)

Reddit r/tf2 (1) (2024): Let's do a poll on TF2 player demographics. https://www.reddit.com/r/tf2/comments/1ejfb56/lets_do_a_poll_on_tf2_player_demographics_how_old (Letöltve: 2025. október)

Reddit r/tf2 (2) (2024): Tf2 User Interface is Horrible. https://www.reddit.com/r/tf2/comments/1h61nqj/tf2_user_interface_is_horrible (Letöltve: 2025. október)

Reddit r/tf2 (2025): How to filter for Spells on bp.tf. https://www.reddit.com/r/tf2/comments/1kziyho/how_to_filter_for_spells_on_bptf (Letöltve: 2025. október)

Silberschatz (2010): Database System Concepts. https://www.slideshare.net/slideshow/database-system-concepts-6th-edition-ebook-pdf/277440767 (Letöltve: 2025. november)

SQ Magazine (2025): Steam Statistics 2025 - Users, Revenue, Top Games, Trends. https://sqmagazine.co.uk/steam-statistics (Letöltve: 2025. október)

Steam Community (2025): Steamworks Web API Documentation. https://steamcommunity.com/dev (Letöltve: 2025. október)

Varoufakis, Y. (2012): Arbitrage and Equilibrium in Team Fortress 2. https://gwern.net/doc/economics/2012-varoufakis-teamfortress2arbitrage.html (Letöltve: 2025. október)

7.5 Forráskódok

Jelen fejezetben található néhány olyan kódrészlet az alkalmazás bizonyos részeiből (pl. kliens oldali felületek, megoldások, illetve szerveroldali részek), amelyek fontosabb szerepet töltenek be az alkalmazás során.

7.5.1 Csere-igény keresés kódrészlet

```
172     public async Task<object> SearchTradesAsync(ICollection<TradeItem> items, int page = 1, int pageSize = 50)
173     {
174         IQueryable<Trade> query = _db.Trades
175             .Where(t => t.Deleted == false && t.Status != "closed");
176 
177         if (items != null && items.Any())
178         {
179             var hasCustomSpells = items.Any(i => i.Name == "CUSTOM_SPELLS");
180 
181             if (hasCustomSpells)
182             {
183                 query = query.Where(t => t.Items.Any(i => i.Spells != null && i.Spells.Count > 0));
184             }
185             else
186             {
187                 foreach (var searchItem in items)
188                 {
189                     var localSearchItem = searchItem;
190 
191                     query = query.Where(t => t.Items.Any(dbItem =>
192                         dbItem.Defindex == localSearchItem.Defindex &&
193                         (localSearchItem.Effect == null || dbItem.Effect == localSearchItem.Effect) &&
194                         (localSearchItem.paintDefindex == null || dbItem.paintDefindex == localSearchItem.paintDefindex) &&
195                         (localSearchItem.Killstreak == null || dbItem.Killstreak == localSearchItem.Killstreak) &&
196                         (string.IsNullOrEmpty(localSearchItem.Killstreaker) || dbItem.Killstreaker == localSearchItem.Killstreaker) &&
197                         (string.IsNullOrEmpty(localSearchItem.Sheen) || dbItem.Sheen == localSearchItem.Sheen) &&
198                         dbItem.IsSelling == localSearchItem.IsSelling
199                     ));
200                 }
201             }
202         }
203 
204         var totalCount = await query.CountAsync();
205 
206         var pagedTrades = await query
207             .Include(t => t.Items)
208             .Skip((page - 1) * pageSize)
209             .Take(pageSize)
210             .AsNoTracking()
211             .ToListAsync();
212 
213         return new
214         {
215             Trades = pagedTrades,
216             Page = page,
217             PageSize = pageSize,
218             TotalCount = totalCount
219         };
220     }
```

47. ábra - SearchTradesAsync csere-igény keresés – kódrészlet – Forrás: Saját implementáció

7.5.2 Item-selector-facade kódrészlet

src > shared > item-selector > item-selector.facade.ts > ...

```
You, 3 weeks ago | 1 author (You)
import { Injectable } from '@angular/core';
import { BehaviorSubject } from 'rxjs';
import { ItemSelectorService } from '../item-selector.service';
import { Item } from '../models/item.model';
import { MessageService } from 'primeng/api';
import { ErrorMessage } from '../models/enums/error-message.enum';
      You, 4 months ago • refact(tf2-node-fromat usage) ...
You, 3 weeks ago | 1 author (You)
@Injectable({ providedIn: 'root' })
export class ItemSelectorFacade {
  private _items = new BehaviorSubject<Item[]>([]);
  private _itemsToTrade = new BehaviorSubject<Item[]>([]);
  private _itemsForTrade = new BehaviorSubject<Item[]>([]);
  private _itemsEditToTrade = new BehaviorSubject<Item[]>([]);
  private _itemsEditForTrade = new BehaviorSubject<Item[]>([]);
  private _itemsSearchToTrade = new BehaviorSubject<Item[]>([]);
  private _itemsSearchForTrade = new BehaviorSubject<Item[]>([]);
  private _itemsOffer = new BehaviorSubject<Item[]>([]);
  private loadedPages = new Set<string>();
  private selectedItemIds = new Set<string>();
  private selectedEditItemIds = new Set<string>();
  private selectedSearchItemIds = new Set<string>();
  private selectedOfferItemIds = new Set<string>();
  loading = false;
  items$ = this._items.asObservable();
  itemsLength = 0;
  itemsToTrade$ = this._itemsToTrade.asObservable();
  itemsForTrade$ = this._itemsForTrade.asObservable();
  itemsEditToTrade$ = this._itemsEditToTrade.asObservable();
  itemsEditForTrade$ = this._itemsEditForTrade.asObservable();
  itemsSearchToTrade$ = this._itemsSearchToTrade.asObservable();
  itemsSearchForTrade$ = this._itemsSearchForTrade.asObservable();
  itemsOfferTrade$ = this._itemsOffer.asObservable();
  private _customIdCounter = 0;
  private _customEditIdCounter = 0;
  private _customSearchIdCounter = 0;
  constructor(
    private itemService: ItemSelectorService,
    private messageService: MessageService
  ) {}
  private deepEqual(obj1: unknown, obj2: unknown): boolean {
    return JSON.stringify(obj1) === JSON.stringify(obj2);
  }
  loadItemsLazy(first: number, rows: number, steamid: string): void {
    this.loading = true;
    const pageKey = `${first}-${rows}`;
    if (this.loadedPages.has(pageKey)) {
      this.loading = false;
      return;
    }
    this.loadedPages.add(pageKey);
    this.itemService.fetchItems(first, rows, steamid).subscribe({
      next: fetchedItems => {
        const currentItems = this._items.getValue();
        this._items.next([...currentItems, ...fetchedItems]);
        this.itemsLength = this._items.getValue().length;
      },
      error: err => console.error('Lazy loading failed', err),
```

48. ábra - ItemSelectorFacade csere-igény kezelő osztály – kódrészlet – Forrás: Saját implementáció

src > shared > item-selector > item-selector.facade.ts > ...

```
  9  export class ItemSelectorFacade {
 63    onAddItem(item: Item) {
 81    }
 82    onAddEditItem(item: Item) {
 83      const currentItems = this._itemsEditToTrade.getValue();
 84      if (currentItems.length >= 10) {
 85        this.showMaxLimitMessage();
 86        return;
 87      } else {
 88        const exists = currentItems.some(existingItem =>
 89          this.deepEqual(existingItem, item)
 90        );
 91
 92        if (!exists) {
 93          const itemWithCustomId = {
 94            ...item,
 95            customId: (++this._customEditIdCounter).toString(),
 96          };
 97          this._itemsEditToTrade.next([...currentItems, itemWithCustomId]);
 98          this.selectedEditItemIds.add(item.id);
 99        }
100      }
101    }
102    onAddSearchItem(item: Item) {
103      const currentItems = this._itemsSearchToTrade.getValue();
104      if (currentItems.length >= 10) {
105        this.showMaxLimitMessage();
106        return;
107      } else {
108        const exists = currentItems.some(existingItem =>
109          this.deepEqual(existingItem, item)
110        );
111        if (!exists) {
112          const itemWithCustomId = {
113            ...item,
114            customId: ++this._customSearchIdCounter,
115          };
116          this._itemsSearchToTrade.next([...currentItems, itemWithCustomId]);
117          this.selectedSearchItemIds.add(item.defindex.toString());
118        }
119      }
120    }
121    onOfferItem(item: Item) {
122      const currentItems = this._itemsOffer.getValue();
123      if (currentItems.length >= 10) {
124        this.showMaxLimitMessage();
125        return;
126      } else {
127        const exists = currentItems.some(existingItem =>
128          this.deepEqual(existingItem, item)
129        );
130
131        if (!exists) {
132          const itemWithCustomId = { ...item, customId: ++this._customIdCounter };
133          this._itemsOffer.next([...currentItems, itemWithCustomId]);
134          this.selectedOfferItemIds.add(item.id);
135        }
136      }
```

49. ábra - Kliensoldali csere-igény kezelő osztály – kódrészlet – Forrás: Saját implementáció

7.5.3 Item-container kódrészlet

src > shared > item > item-container.component.ts > ...

You, 3 weeks ago | 1 author (You)

```typescript
import {
  Component,
  EventEmitter,
  inject,
  Input,
  OnChanges,
  Output,
  SimpleChanges,
} from '@angular/core';
import { ItemComponent } from './item.component';
import { ItemFacade } from './item.facade';
import { TradeServiceFacade } from '../../app/add-trade/trade-service.facade';
import { ItemSelectorFacade } from '../item-selector/item-selector.facade';
import { Item } from '../models/item.model';
```

You, 3 weeks ago | 1 author (You)

```typescript
@Component({
  selector: 'app-item-container',
  imports: [ItemComponent],
  templateUrl: './item-container.component.html',
})
export class ItemContainerComponent implements OnChanges {
  @Input() item!: Item;

  @Input() disabled = false;
  @Input() canDelete = false;
  @Input() canModify = false;
  @Input() showQuantity = false;
  @Output() selectEmitter: EventEmitter<Item> = new EventEmitter<Item>();
  @Output() removeEmitter: EventEmitter<Item> = new EventEmitter<Item>();
  @Output() customizeEmitter: EventEmitter<Item> = new EventEmitter<Item>();
  facade = inject(ItemFacade);
  itemSelectorFacade = inject(ItemSelectorFacade);
  tradeFacade = inject(TradeServiceFacade);

  ngOnChanges(changes: SimpleChanges): void {
    if (changes['item'] && this.item) {
      this.facade.checkItemExtras(this.item);
    }
  }
}
```

50. ábra – ItemContainerComponent tárgy megjelenítő elem – kódrészlet – Forrás: Saját implementáció

51. ábra - ItemContainerComponent tárgy megjelenítő elem – kódrészlet – Forrás: Saját implementáció