

Debreceni Egyetem
Informatikai Kar

TÖBBFELHASZNÁLÓS WEBES SZEREPJÁTÉK FEJLESZTÉSE DJANGO KERETRENDSZERREL

Témavezető:

Dr. Szathmáry László
egyetemi docens

Készítette:

Tamás Ádám
mérnökinformatikus

Debrecen
2022

TARTALOMJEGYZÉK

1.	Bevezetés.....	4
2.	Felhasznált technológiák.....	6
2.1.	A Django webes keretrendszer.....	6
2.1.1.	<i>A Django bemutatása.....</i>	<i>6</i>
2.1.2.	<i>A Django és a Python kapcsolata.....</i>	<i>8</i>
2.1.3.	<i>Az MVT architektúra és komponensei.....</i>	<i>10</i>
2.1.4.	<i>A Django felépítése.....</i>	<i>12</i>
2.1.5.	<i>A Django működése.....</i>	<i>16</i>
2.1.6.	<i>Adatbáziskezelés.....</i>	<i>18</i>
2.2.	A Django Channels és a WebSocketek.....	19
2.2.1.	<i>A Django Channels kialakulása.....</i>	<i>19</i>
2.2.2.	<i>WSGI és ASGI interfészek.....</i>	<i>19</i>
2.2.3.	<i>Consumer-ek bemutatása.....</i>	<i>20</i>
2.2.4.	<i>A Channel Layer-ek szerepe.....</i>	<i>23</i>
2.2.5.	<i>WebSocket kezelés JavaScriptben.....</i>	<i>24</i>
2.2.6.	<i>A valós idejű kommunikáció folyamata.....</i>	<i>25</i>
2.3.	A front-end technológiái.....	26
2.3.1.	<i>HTML, CSS és JavaScript.....</i>	<i>26</i>
2.3.2.	<i>A jQuery keretrendszer.....</i>	<i>27</i>
2.3.3.	<i>Az AJAX technika.....</i>	<i>28</i>
3.	A feladat bemutatása és a szoftver ismertetése.....	29
3.1.	A feladat bemutatása.....	29
3.2.	A szoftver ismertetése.....	30
3.2.1.	<i>A szoftver futtatásának követelményei.....</i>	<i>30</i>
3.2.2.	<i>A szoftver használata.....</i>	<i>31</i>
3.2.3.	<i>A szoftver felépítése.....</i>	<i>40</i>
3.2.4.	<i>Továbbfejlesztési lehetőségek.....</i>	<i>67</i>
4.	Összefoglalás.....	69

5.	Irodalomjegyzék.....	71
5.1.	Irodalmi források	71
5.2.	Internetes források	71
6.	Köszönetnyilvánítás	73

1. BEVEZETÉS

Napjainkban egyre jobban érezhető a gyorsuló világ hatása az életünkre. A nap folyamán folyamatosan különböző interakciók érnek minket. Minden percet próbálunk kitölteni valamivel, és ez gyakran eredményezi azt, hogy felesleges dolgokat csinálunk. Saját magamon is tapasztaltam, hogy hajlamos vagyok a telefon vagy a gép előtt ragadni hosszú percekre, akár órákra is, és valójában csak görgetni az egyes közösségi oldalakat, bármiféle cél nélkül.

Talán ennek is köszönhetően, de egyre nagyobb népszerűségnek örvendenek az online szerepjátékok. Ezeknek a játékoknak a lényege a legtöbb esetben az, hogy a játékos által létrehozott karaktert fejlessze, és minél nagyobb szintet érjen el, ezzel megelőzve a többi játékost. Ezekkel a játékokkal nem szükséges hosszú órákon keresztül játszani. Gyakran néhány perc is elegendő, ami alatt a játékos fejleszti a karakterét, esetleg néhány küldetést lejátszik a szintlépés eléréseért. Jellegéből adódóan, az ilyen típusú játékok alkalmasak a napjainkban keletkezett üresjáratok kitöltésére, például a munkahelyen vagy az iskolában mikor épp' van egy kis pihenőidőnk, és valamivel el szeretnénk foglalni magunkat. Szakdolgozatom célja egy ilyen típusú játék létrehozása volt. Próbáltam egy lehetséges megoldást nyújtani ezeknek az üresjáratoknak a produktív betöltésére. Az általam elkészített szerepjáték kicsit eltér a többitől. Itt a karakter fejlesztését kisebb minijátékokkal lehet elérni. Ezek a játékok direkt úgy lettek kiválasztva és megvalósítva, hogy egyrészt segítse a felhasználót a kikapcsolódásban, azonban ezzel egy időben fejlessze a logikus gondolkodást és a memóriát is. Így elmondhatjuk, hogy a játékkal eltöltött idővel ténylegesen profitálhatunk is. Az előbb említett készségek fejlesztése a mindennapi teendőink végzésekor is hasznunkra válhat. A játékot barátainkkal, kollégáinkkal együtt is játszhatjuk, hiszen a játékon belül lehetőségünk van beszélgetni is egymással chaten keresztül.

A webalkalmazások fejlesztése iránti érdeklődés egészen általános iskolás koromig nyúlik vissza. Akkoriban találkoztam először a HTML és CSS nyelvekkel, amelyek a webalkalmazás fejlesztésének alapjai. Az egyetemi tanulmányaim során találkoztam a Python programozási nyelvvel. Igyekeztem minél jobban elmélyülni, és minél több tudást magamba szívni ezzel kapcsolatban. Ennek a folyamatnak a során találkoztam a Django keretrendszerrel is. Akkoriban nem tudtam, hogy mik azok a keretrendszerek, és hogyan lehet egy weboldalt felbontani szerver és kliens oldalra. Mivel érdekelt a technológia, elkezdtem beleásni magam, és sikerült egy alap tudást összeszedni a webalkalmazások fejlesztésének és működésének a

folyamatáról. A kutatás után a Django keretrendszer alapjait is elsajátítottam. Az alapok után láttam mennyi mindent lehet ennek a keretrendszernek a használatával megvalósítani, ezért úgy döntöttem, hogy a szakdolgozatomat erről a technológiáról fogom írni. Ez egy remek lehetőség volt arra is, hogy egyetemi kereteken belül létrehozzam az első nagyobb projektem. Egy olyan webalkalmazást, amelyet aztán a későbbiekben referenciaként akár fel is használhatok, hogyha ehhez kapcsolódó területen helyezkedek el. Korábban csupán kisebb weboldalakat készítettem, amelyek mögött nem volt komolyabb funkcionalitás. Ennek a projektnek az elkészítésével nem csak piacképes tudást, hanem gyakorlatot is szereztem egy webalkalmazás elkészítésével kapcsolatban. A dolgozatban használt technológiák között van olyan is, ami csupán néhány éve jelent meg, viszont már most rengeteg lehetőség rejlik benne. Erre külön büszke vagyok, hogy sikerült a webalkalmazásba beépíteni és működésre bírni.

A dolgozat célja, hogy bemutassa a szerepjáték elkészítéséhez használt technológiákat, valamint a játéknak a működését felhasználói és fejlesztői szemszögből. Épp ezért a dolgozatot két fő részre bontottam fel. Az első részben bemutatom az alkalmazás létrehozásához használt technológiákat. Részletezem azoknak az alapvető működését és felépítését. Ha semmit nem tudunk az adott technológiáról, ez a dolgozat segíthet abban, hogy egy alap tudást megszerezzünk a technológiával kapcsolatban. A második részben az elkészült alkalmazást mutatom be. Ebben a részben először a felhasználói oldal kerül ismertetésre, majd azt követően a fejlesztői rész. A felhasználói rész az alkalmazás kinézetéről tartalmaz információkat, illetve egy általános leírást, hogy az olvasó tisztában legyen az egyes részek működésével. A fejlesztői részben részletezem az alkalmazás konkrét felépítését, az alkalmazás mögött rejlő modelleknek a kapcsolatát. Továbbá bemutatom, hogy a felhasználói részben ismertetett funkciók hogyan lettek megvalósítva kód szinten. Egy különálló fejezetben szó esik az alkalmazás futtatásához szükséges követelményekről is. Ezáltal, ha valaki saját maga akarja futtatni az elkészült alkalmazást, ezt a részt elolvasva lehetősége lesz rá.

A dolgozatot elolvasva az olvasó kapni fog egy képet arról, hogy egy Django keretrendszer alapú webalkalmazást hogyan kell létrehozni, és hogyan működnek az alapvető funkciói. Továbbá megtudja, hogy az általam létrehozott alkalmazás hogyan működik, és hogyan lettek az egyes részei felépítve.

2. FELHASZNÁLT TECHNOLOGIÁK

Ebben a fejezetben a szoftver elkészítéséhez használt technológiákat és módszereket mutatom be részletesebben. Részletezem azt is, hogy miért pont ezekre a technológiákra esett a választás a projekt tervezésekor.

2.1. A Django webes keretrendszer

A szoftver magját a Django webes keretrendszer alkotja. Ez a keretrendszer kezeli a legalapvetőbb funkciókat a programban. A következőkben ez a technológia kerül bemutatásra részletesebben.

2.1.1. A Django bemutatása

A projekt tervezésekor fontos szempont volt, hogy egy olyan keretrendszer legyen a szoftver alapja, amely megfelel a jelenkori elvárásoknak, és egy modern, jól karbantartható weboldalt lehessen létrehozni a felhasználásával. A Django egy ingyenesen elérhető és nyílt forráskódú webes keretrendszer (1. ábra). Alapját a Python programozási nyelv képezi. Hivatalosan 2005 júliusában jelent meg, azonban a fejlesztése már 2003 őszén elkezdődött. Eredeti alkotói Adrian Holovaty és Simon Willison Python programozók voltak.



1. ábra
A keretrendszer logója

Napjainkban az egyik legelterjedtebb és legnépszerűbb keretrendszer webalkalmazások fejlesztéséhez. A következőkben felsorolom a legfontosabb alapelveit és tulajdonságait a keretrendszernek, amelyek közrejátszottak abban, hogy ezt választottam a szoftver alapjának:

- Gyors alkalmazás fejlesztés: a keretrendszer használatával rendkívül gyorsan lehet komplett alkalmazásokat elkészíteni. Ez többek között annak is köszönhető, hogy alapvető elve a keretrendszernek a DRY (Don't Repeat Yourself), melynek célja, hogy

a programozó elkerülje a kódismétlést. Ezáltal egy tisztább, egyszerűbb és redundancia mentes kód készíthető el. Ez a tulajdonsága különösen hasznos volt a keretrendszer kiválasztásakor, hiszen a szoftver elkészítéséhez csupán néhány hónap állt rendelkezésre.

- Újrafelhasználhatóság: az alkalmazások, amelyeknek az alapját ez a keretrendszer alkotja, úgynevezett app-okból épülnek fel. Ezek tulajdonképpen kisebb modulok, amely a komplett alkalmazást alkotják. Az egyes modulok újabb funkcionalitást tartalmaznak, amellyel bővíthető a már meglévő alkalmazás. A modulok könnyen hozzáadhatóak, valamint eltávolíthatóak az alkalmazásból. Ezáltal a modulon belül megírt kód akár több alkalmazásban, változtatás nélkül felhasználható, abban az esetben, hogyha ugyanazt a funkcionalitást kívánjuk elérni. A szoftver fejlesztése során rendkívül hasznos volt ez az úgynevezett moduláris felépítés. Könnyen fel lehetett osztani az elvégzendő munkát, valamint a tesztelés és hibaelhárítás is sokkal egyszerűbben és gyorsabban zajlott.
- Skálázhatóság: rendkívül jól skálázható. Az alkalmazás egyes rétegei függetlenek a többitől, ezáltal bármelyik rétegben módosítás végezhető el anélkül, hogy befolyással lenne a másikra.
- Beépített funkciók: a keretrendszerben előre létrehozott, és már implementált rendszerek vannak, ezzel nagyban segítve a fejlesztést. Ilyen például az autentikációs rendszer, valamint az admin panel is. Mivel a szakdolgozat, illetve a szoftver célja nem ezeknek a rendszereknek a létrehozására irányult, ezért különösen hasznos volt ezeknek a megléte.
- Biztonság: nagy hangsúly van fektetve a biztonságra is. Védelmet biztosít többek között a CSRF sebezhetőség, SQL befecskendezés és clickjacking ellen is. Továbbá a jelszavakat is titkosítva tárolja, melynek értékét még a programozó sem tudja elérni.
- Közösség támogatása: mivel ennyire elterjedt keretrendszerről van szó, rendkívül sokan használják, és mélyítik el tudásukat benne napról napra egyre jobban. Rengeteg oktató videó és könyv érhető el a témával kapcsolatban, valamint az online fórumokon szintén hasznos információkhoz lehet jutni. Az egyik legfontosabb tényezők közé tartozik ez a tulajdonsága. Mivel az egyetemi tanulmányaim alatt nem találkoztam a Django-val, ezért saját magamnak kellett elsajátítanom a keretrendszer sajátosságait, és megszerezni azt a tudást, amelynek a felhasználásával el tudtam készíteni a megtervezett szoftvert.

A Django egy szerver oldali keretrendszer, melynek a fő feladata, hogy kezelje a klienstől érkező HTTP kérélmeket, és választ adjon rájuk, a megírt funkcionalitás alapján. Alapvetően kezelhető full-stack keretrendszerként is, mivel a template rendszernek köszönhetően elő tudja állítani a megjelenítéshez szükséges HTML fájlokat, és azokat el tudja küldeni a kliens oldalra. Azonban napjainkban a bevett szokás az, hogy a Django csak a szerver oldali folyamatokat kezeli, és a kliens oldalon a megjelenítésért egy kliens oldali keretrendszer felel. Gyakori párosítás például a Django-React vagy a Django-Angular keretrendszerek használata. Ezzel méginkább elősegítve az újrafelhasználhatóságot, és nem utolsó sorban sokkal egyszerűbb az egyes részek karbantartása is. A fejlesztés a szerver oldal megtervezésével és létrehozásával kezdődött. A kliens oldalon nem tudtam még, hogy szükségem lesz-e valamilyen keretrendszer használatára, ezért mikor a Django került kiválasztásra, nagy hasznát vettem annak, hogy nem szükséges külön a kliens oldalon is kiépíteni valamilyen rendszert az egyes oldalak megjelenítéséhez.

2.1.2. A Django és a Python kapcsolata

A Django keretrendszer teljes egészében a Python programozási nyelv felhasználásával készült el. A Python egy magas szintű, objektum-orientált programozási nyelv (2. ábra). Tervezője egy holland kutató és programozó, Guido van Rossum. A nyelv első változata 1991-ben jelent meg.



2. ábra
A Python logója

Napjainkban az első helyet foglalja el a legnépszerűbb programozási nyelvek listáján, a TIOBE index szerint.¹ Szintaxisának köszönhetően jól olvasható és könnyen érthető kódot lehet írni ennek a nyelvnek a felhasználásával. Többek között ennek is köszönhető, hogy a fejlesztés

¹ A TIOBE index a programozási nyelvek népszerűségét méri a keresési számok alapján:
<https://www.tiobe.com/tiobe-index/>

gyors és hatékony. Az ebből származó költségek alacsonyabbak, és a karbantartás, illetve a hibaelhárítás is könnyebben végezhető el. Hasonlóan a Django-hoz, a Python is moduláris felépítésű. Az egyes modulokat egyszerűen be lehet importálni az adott programba, hogyha szükség van rájuk. Ezáltal az egyszer megírt kód több projektben is felhasználható. Magas szintű programozási nyelv révén, a Pythonban megírt kód könnyen hordozható különböző számítógépek és operációs rendszerek között is. A programnyelv dinamikus, ami azt jelenti, hogy nem szükséges a változóknak a típusát meghatároznunk, valamint automatikus memóriakezelést is végrehajt a programozó helyett. Összehasonlítva például a C programozási nyelvvel, ott a programozónak kell gondoskodnia a már nem szükséges memóriaterület felszabadításáról. Az alkalmazás fejlesztése során a legtöbb esetben megkönnyítette a munkát az előbbi két tulajdonsága a nyelvnek. Azonban néhány esetben bonyodalmat okozott, hogy nem volt konkrétan megadva egy változónak a típusa. Ezekben az esetekben manuálisan ellenőriztem le a pontos típusát a változónak. Szerencsére erre is van beépített függvény, a **type()** kulcsszóval könnyen ellenőrizhető egy objektumnak a típusa. A Python alkalmas többek között szoftverfejlesztésre, különböző scriptek megírására, az egyik leghatékonyabb nyelv a gépi tanuláshoz és mesterséges intelligenciához, valamint nem utolsósorban webalkalmazás fejlesztéséhez is. A Django-ban létrehozott webalkalmazás funkcionalitása is Python kódban írható meg. A programozási nyelv számos adattípust támogat, ebből jelen esetben kiemelendő a szótár adattípus (3. ábra). Az általam megírt alkalmazás ezt az adattípust használja a legtöbbször, például ebben az adattípusban adódnak át az adatok a template-nek, illetve a szerver és kliens közötti kommunikációnak is ez az adattípus az alapja.

```
context = {
    'form': form,
    'username': current_username,
    'email': current_email,
    'profile_image': current_profile_image,
    'error_message': error_message,
}
```

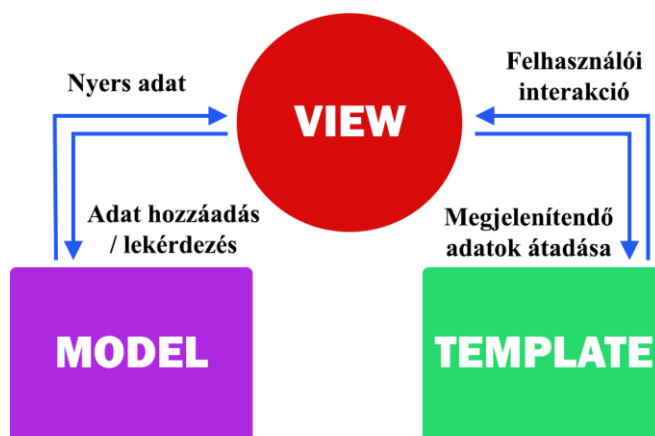
3. ábra
Szótár adatszerkezet a szoftverből

A szótár kulcs-érték pároknak a kollekciónja, ahol a kulcsnak egyedinek kell lennie a szótár egészére nézve. Nem lehet ugyanazon kulcs-érték pár kétszer ugyanabban a szótárban. Az egyes kulcs-érték párokat vesszővel kell elválasztani. Az utolsó elem után nem szükséges vesszőt tenni, de mivel problémát se okoz, ezért én onnan se hagytam el. Ennek az oka, hogyha esetleg egy új kulcs-érték pár kerülne a szótárba, emiatt ne kapjak hibaüzenetet, ezáltal

gyorsítva a fejlesztést. Összességében a szótárak segítségével hatékonyan lehet adatokat tárolni, valamint felhasználni a webalkalmazáson belül.

2.1.3. Az MVT architektúra és komponensei

A Django az MVC (Model-View-Controller) tervezési mintán alapszik, azonban az implementálás során egy kisebb változtatást eszközöltek rajta. A fő különbség, hogy a vezérlő (controller) komponenst maga a keretrendszer kezeli. A beérkező kérések egyből a View-hoz mennek, amely az alkalmazás logikájának számít. Emiatt a változtatás miatt az MVC helyett Django-ban az MVT (Model-View-Template) tervezési minta van használatban (4. ábra).



4. ábra

Az MVT tervezési minta szerkezete

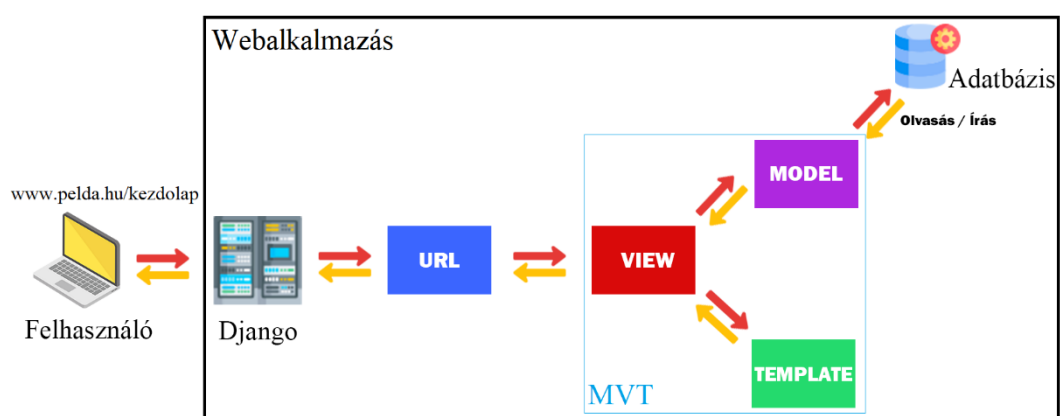
A tervezési minta első rétege a **Model**. Ez a réteg írja le az adatbázistábláknak a felépítését. Az egyes táblákat egyszerű Python osztályokkal definiálhatjuk, melyeknek az egyes attribútumai a táblák oszlopai. További függvények is hozzáadhatóak az egyes osztályokhoz, amelyeknek a segítségével különböző műveleteket végezhetünk az adatokon, például bizonyos attribútumokra való szűrés is ezzel oldható meg a legkönnyebben. Ez a réteg teremti meg a kapcsolatot az adatbázissal, az adatok eléréséért felel. A szoftvernek a tervezésekor a modellek helyes és hatékony megtervezése fontos szempont volt, hiszen az egész szoftver ezekre a modellekre épül fel.

A tervezési minta második rétege a **View**. Ez a réteg fogadja az egyes kéréseket és küldi a válaszokat a kliensnek. A view egyszerű Python kód, ami lehet függvény (*function based view*) és osztály is (*class based view*). A szoftverben a függvény alapú változatot használtam, és a

továbbiakban magyarul a nézetfüggvény elnevezést fogom használni rá. Ez a réteg éri el az egyes modelleken keresztül az adatokat, és adja át azokat a megfelelő template-ekhez. Az átadás a korábban említett kulcs-érték párok használatával történik, Python szótár segítségével. A kulcs az az érték, amelyet a template-eknél használhatunk az értékek megjelenítésére.

A tervezési minta legutolsó rétege a **Template**. Ez a réteg felel az adatok helyes megjelenítéséért. Magába foglalja az egyes HTML, CSS és JavaScript fájlokat. A felhasználónak küldött HTTP válaszban ez a fájl kerül elküldésre és megjelenítésre a kliens oldalon. A Django ennek a segítségével tud könnyen HTML oldalakat dinamikusan előállítani. Beépített nyelvvel rendelkezik (DTL – Django Template Language). Ennek a segítségével lehetséges az adatok megjelenítése az egyes oldalakon. Például a nézetfüggvényből átadott kulcs-érték párokra `{{ kulcs-neve }}` formátumban tudunk hivatkozni. Ilyenkor a kulcshoz tartozó érték fog megjelenni az oldalon. Lehetőség van *for* ciklus és *if-else* szerkezetek létrehozására is. Ezáltal még dinamikusabbá lehet tenni a weboldalt. *For* ciklust például a következő módon lehet létrehozni: `{% for item in items %} <p>{{item}}</p> {% endfor %}`. Látható, hogy `{% %}` tagok közé kell helyezni és le is kell zárni a ciklust, melynek a szintaxisa a Pythonban használttal egyezik meg. Az *items* az egy nézetfüggvényből kapott kulcs érték, amit ilyenkor nem szükséges `{ { }` közé tenni.

A 5. ábra egy egyszerűsített folyamatábrán keresztül mutatja be, mi történik, mikor egy felhasználó megnyitja a webalkalmazást. Ennek a kibővítése a *Django működése* fejezetben lesz olvasható.



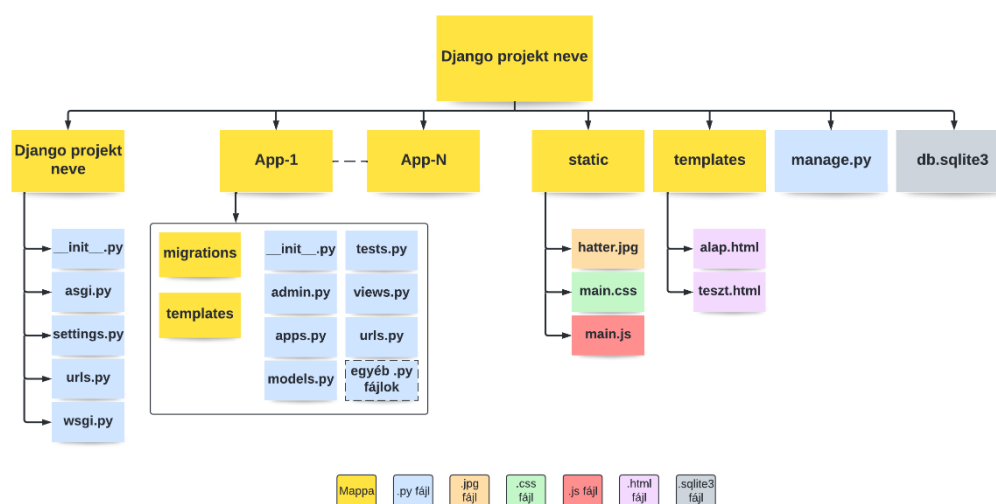
5. ábra
Az MVT tervezési minta működése

A megnyitás során elküldött HTTP kérelem először a Django szerverhez fut be. Ezután a *urls.py* fájl felhasználásával végigmegy az összes definiált URL-en a szoftver, amelyekhez az egyes

nézetfüggvények vannak társítva. Amint egyezést talál, meghívja és lefuttatja az adott nézetfüggvényben megírt kódot. Ezután a függvény által visszaadott template elküldésre kerül és megjelenik a kliens oldalon. Amennyiben nem található megegyező URL, vagy pedig a nézetfüggvényben valamilyen hiba következik be, a Django megjeleníti az adott hibát a megfelelő hibakezelő függvény segítségével.

2.1.4. A Django felépítése

A Django projektnek előre meghatározott mappaszerkezete van, melyben a forrásfájlok helyezkednek el. Ezáltal a webalkalmazás egyes alkotóelemei szétválasztva tárolhatóak, mellyel átláthatóbb képet kapunk az egész webalkalmazásról. Egy alap Django projekt mappaszerkezetét mutatja be a 6. ábra.



6. ábra
Alap Django projekt mappaszerkezete

Jól látható, hogy a projekt létrehozásakor a fő tároló mappának, és a benne található egyik almappának a neve – mely egyben a projektnek a neve is – megegyezik. Ezért a bevett szokás a Django programozók körében, hogy a fő tároló mappát átnevezik, hogy ez a későbbiekben ne okozzon félreértéseket. Az *src* (source) nevet szokás adni ilyenkor a mappának, utalva arra, hogy a webalkalmazás forrásfájljai találhatóak a mappában. A következőkben bemutatásra kerülnek az egyes alkotóelemek részletesebben.

Django projekt neve: ez a mappa a projekt létrehozásának a pillanatában jön létre. A projektnek az alapvető konfigurációs fájljait tartalmazza.

Az `__init__.py` egy üres Python fájl, amelynek célja, hogy jelezze a Python interpreternek, hogy ez a mappa egy Python csomag (package).

A `settings.py` az egyik legfontosabb része a struktúrának. Ebben van benne az összes beállítása a projektnek. Többek között az adatbázisbeállítások, az általunk létrehozott app-ok nevei, a statikus fájljainknak az elérési útja, valamint a middleware-eknek és a projektnek további alapvető beállításai, amelyek az alkalmazás működtetéséhez szükségesek.

A `urls.py` tartalmazza az összes elérési utat az egyes erőforrásokhoz. Mikor a felhasználó megnyitja a webalkalmazást, ez a fájl kerül meghívásra. Ez alapján dől el melyik nézetfüggvény fut le, és ezáltal melyik template jelenik meg az oldalon a felhasználónak. Ebbe a fájlba lehet már működő, valós URL-eket írni, valamint hozzá lehet adni egy másik app `urls.py` állományát. Általában a második opció a gyakoribb. Hatékonyabb, hogyha egy app saját maga kezeli az egyes elérési útvonalait, és a fő fájlba csupán csak beilleszteni szükséges ezeket. Ez a már korábban említett újrafelhasználhatóságot könnyíti, és a fejlesztést gyorsítja.

A `wsgi.py` és az `asgi.py` konfigurációs fájlok, interfészként szolgálnak a webalkalmazás és a webszerver között. Ezeknek a pontos működése egy külön fejezetben lesz bemutatva.

Appok: minden egyes app külön mappában helyezkedik el. Ahogy korábban írtam, az appok kisebb részei a teljes projektnek, mellyel újabb funkcionalitást lehet hozzáadni az alkalmazásunkhoz. Ahogy a 6. ábra mutatja, egy projekten belül több appot is létre lehet hozni, ezeknek a száma nincsen korlátozva. Az egyes app-ok felépítése azonos.

Az `__init__.py` funkciója ugyanaz, mint a fő mappában lévő `__init__.py`. Jelzi, hogy az adott mappa egy csomag.

Az `admin.py` segítségével lehet a Django admin felületéhez hozzáadni az adott app-ot és a benne használt modelleket. Mi szabhatjuk meg, mely attribútumai jelenjenek meg az admin felületen az egyes modelleknek. Megadható az is, hogy ezek az attribútumok módosíthatóak-e vagy sem, illetve számos egyéb tulajdonság, amellyel az admin felületen történő munkát segíthetjük elő. Ezen a felületen hozható létre manuálisan, kódolás nélkül az egyes modellek példányai, valamint tekinthetjük meg az adatbázisban tárolt elemeket.

Az `apps.py` az egyes appok konfigurációs fájlja, mely az app létrehozásakor jön létre. A szoftver fejlesztése alatt nem volt szükség ennek a fájlnak a módosítására.

A `models.py` tartalmazza az alkalmazás egyes modelljeit. Ez a korábban már részletezett MVT architektúra M része. Ebben a fájlban létrehozott modell osztályok alkotják az adatbázis tábláit.

A *tests.py* nevéből adódóan az app-hoz tartozó tesztek megírására alkalmas. Alapvetően ez egy üres fájl, csupán a *TestCase* modul van importálva a fájlba. Ez a modul az egységtesztek (*Unit Test*) írására alkalmas.

A *views.py* az MVT architektúra V része. Ebben a fájlban találhatóak az app-hoz tartozó nézetfüggvények, amelyeknek a segítségével történik az interakció a felhasználó és a webalkalmazás között.

A *urls.py* hasonló felépítésű, mint a főmappában található társa. Itt találhatóak az egyes elérési utak az app-on belüli nézetfüggvényekhez. Itt már nem jellemző egy másik app *urls.py* állományának a beillesztése, ahogy azt a főmappában lévő verziónál említettem, azonban programozói szempontból megoldható lenne. A 7. ábra bemutatja a URL-ek megadásának két módját. Az első esetben meg kell adni első paraméternek az elérési utat, majd az elérési úthoz tartozó nézetfüggvény nevét. Ezután a harmadik paraméterként megadható egy név is, amivel a későbbiekben hivatkozni lehet az adott URL-re a kódon belül. A második esetben egy URL fájlt illesztünk be egy másik appból. Az első paraméternek, hasonlóan az előbbihez, az elérési útvonalat kell megadni, majd az *include* kulcsszóval beilleszthető az adott *urls.py* állomány, *app-név.urls-fájl-név* formátumban. Ebben az esetben a *namespace* kulcsszót használjuk, hogy az adott elérési útnak nevet adjunk.

```
urlpatterns = [  
    path('admin/', admin.site.urls),  
  
    # hagyományos url megadás  
    path('regisztracio/', register_page_view, name='register'),  
    path('bejelentkezés/', login_page_view, name='login'),  
  
    # urls.py fájl beillesztése egy másik app-ból  
    path('jatek/', include('game.urls', namespace='game')),  
]
```

7. ábra
A fő *urls.py* egy részlete

Ezekon a fájlokon kívül további *.py* fájlok is hozzáadhatóak a mappaszerkezethez. Attól függően, hogy mennyi és milyen funkcionalitást adunk hozzá az egyes app-okhoz, az adott kódrészleteket érdemes lehet egy külön fájlba írni. A szoftver fejlesztése során például külön fájlba szedtem az egyes konstanst értékeket, ezáltal sokkal átláthatóbban tudtam tárolni ezeket. A *views.py* mérete is csökkenthető ezzel, hogyha egy terjedelmesebb programkód adott

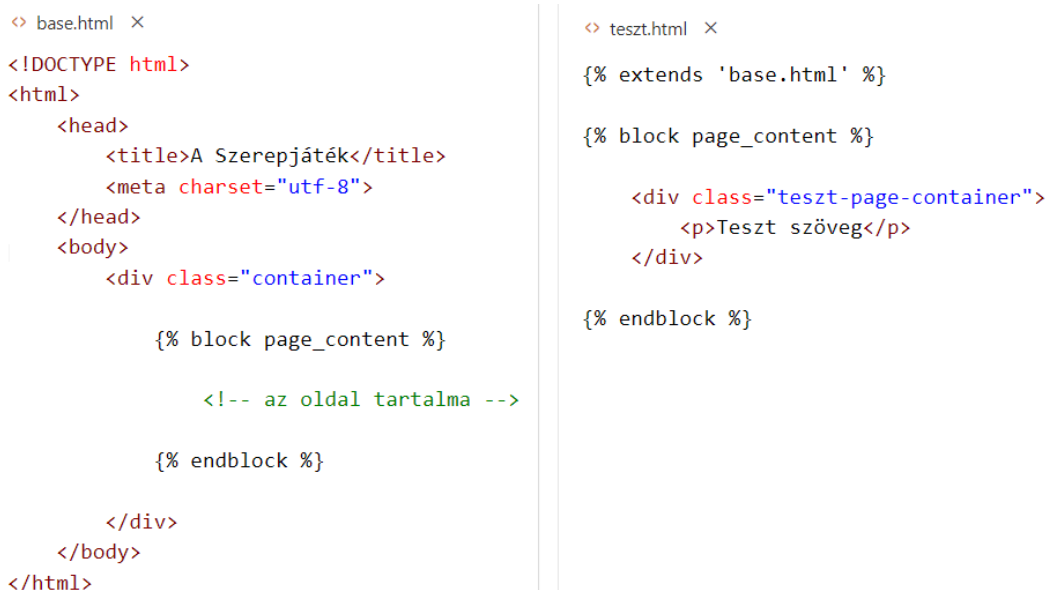
függvényeit külön fájlba tároljuk, és csak a fő függvény van a *views.py*-ban. A Python fájlokon kívül található még két mappa is az app mappán belül.

A *migrations* mappa tárolja az egyes módosításokat, amiket fejlesztés közben végzünk a modellen, és ezáltal az adatbázis sémán is. Például hogyha a modellhez egy új attribútumot adunk hozzá, vagy veszünk el, és ezt a módosítást a *migrate* paranccsal életbe léptetjük, az ebben a mappában egy *.py* állományban tárolva lesz.

A *templates* mappa tárolja az egyes *.html* állományokat, azaz template-eket. Ez az MVT architektúra T része. Alapesetben ez a mappa az app-on belül nem létezik. Azonban ez a bevett szokás, hogy az appokhoz tartozó template-eket az appon belül tároljuk. Itt tárolhatóak még továbbá a HTML fájlokhoz tartozó JavaScript és CSS fájlok is.

static: ez a mappa tárolja a programhoz tartozó statikus fájlokat. Ilyen fájlok például a képek, a videók, az egyes hangfájlok, például az MP3 kiterjesztésű fájlok, de ide lehet sorolni a CSS és JavaScript fájlokat is. Azonban az előző kettő nem feltétlenül csak itt tárolható, ezt a programozó dönti el. Gyakran a *templates* mappában helyezkednek el ezek a fájlok, ahogy azt az előbb említettem, vagy pedig a HTML fájlokon belül.

templates: ez a projektnek a fő tároló mappája a template-k tekintetében. Azonban itt általában csak egy alap váz kerül eltárolásra. Tulajdonképpen egy sablon, ami megadja a szerkezetét a többi HTML fájlnek. A Django-ban ez a *template inheritance*, azaz template öröklődés segítségével érhető el. Ez egy nagyon hasznos funkciója a keretrendszernek.



```
<> base.html ×
<!DOCTYPE html>
<html>
  <head>
    <title>A Szerepjáték</title>
    <meta charset="utf-8">
  </head>
  <body>
    <div class="container">

      {% block page_content %}

        <!-- az oldal tartalma -->

      {% endblock %}

    </div>
  </body>
</html>

<> teszt.html ×
{% extends 'base.html' %}

{% block page_content %}

  <div class="teszt-page-container">
    <p>Teszt szöveg</p>
  </div>

{% endblock %}
```

8. ábra
A template öröklődés bemutatása

A 8. ábra bemutatja, hogyan működik ez a gyakorlatban. A bal oldali fájl a *base.html*, a jobb oldali pedig egy adott appon belül használt *teszt.html*. Látható, hogy nem szükséges minden HTML állományt a 0-ról felépíteni. A közös egységeket, például az oldal címét, illetve egyéb attribútumokat elég csak a *base.html*-ben egyszer megírni. Ebben a fájlban létrehoztunk egy *page_content* nevű blokkot. Ez a blokknak a neve, ami kezelhető egy változóként is, neve tetszőlegesen megadható. Ez lesz az a rész, ahova a többi HTML oldal tartalma fog beillesztésre kerülni. A HTML oldalak az *extends* kulcsszó segítségével tudják kiterjeszteni a *base.html*-t. Ilyenkor meg kell adni hasonlóan kapcsos zárójelek között a *block* kulcsszót és a nevét, illetve le is kell zárni azt. Ami ezek között a tag-ek között van, az fog megjelenni az oldalon. Fontos, hogy ami ezeken kívülre esik, az nem fog megjelenni az alap fájlban. Ehhez hasonló blokkok több helyen és több névvel is megadhatóak egyszerre. Például létre lehetne hozni egy hasonló blokk párost a *base.html* fájl *<head>* részében, és az egyes kiterjesztő HTML fájlok testre tudnák szabni az adott *<head>* részeket, hogyha szükség lenne rá.

manage.py: ennek a fájlnak segítségével lehet az alapvető parancsokat kiadni a parancssorban, például a webservert futtatásához is ez a fájl szükséges.

db.sqlite3: ez egy adatbázis fájl. A Django alapvetően beépített SQLite adatbázist bocsájt a rendelkezésünkre. Ilyenkor nem kell törődnünk az adatbázis létrehozásával, egyből használhatjuk a programot. Ez az a fájl, amelyben tárolva vannak az általunk létrehozott modellek és hozzájuk kapcsolódó adatok.

2.1.5. A Django működése

Az előző fejezetben bemutatásra kerültek a Django projekt alap alkotóelemei. A most következőkben részletezem mi történik, mikor a felhasználó megnyit egy Django-ban írt weboldalt, és bemutatásra kerül az ilyenkor végrehajtott kérelem-válasz folyamata. Ennek egy része már ismertette volt az MVT architektúrájánál. Ez a rész az ott leírtak kibővítésére szolgál. Mikor a felhasználó megnyit egy oldalt, egy HTTP kérelmet fog küldeni az erőforrások lekérdezésére. A szoftver esetében ezek a template fájlokat jelentik legtöbbször. Ezt a kérelmet a webservert kapja meg. A Django beépített fejlesztői webserverral van ellátva, ezt azonban élesben nem érdemes használni, mivel nem biztonságos és nem hatékony. A webservert kezeli a HTTP kérelmeket, és ez az, ami a választ is visszaküldi a kliensnek. A webalkalmazás pedig az üzleti logikát, a funkcionalitásokat jelenti, hogy az egyes kérelmek esetén mi jelenjen meg az elküldendő HTTP válaszban. A webalkalmazás része a URL-ek kezelése és az MVT

architektúra bemutatott részei is. A HTTP kérelem és válasz nem egy kétlépcsős folyamat, a Django keretrendszer számos egyéb kódot és ellenőrzést futtat a háttérben. A kommunikáció alapja a WSGI (Web Server Gateway Interface), amelynek a feladata, hogy kapcsolatot teremtsen a webservert és a Django alkalmazás, azaz a webalkalmazás között. A felhasználótól beérkező kérelem ezen az interfészen keresztül fog eljutni az alkalmazáshoz. Ennek az interfésznek a feladata, hogy a HTTP kérélmeket átalakítsa Python objektumokká, amelyeket utána az alkalmazásban fel tudunk dolgozni. Ezek az úgynevezett *HTTP Request* objektumok. A beérkezett kérelem ezután a *middleware*-eken keresztül fog eljutni a fő URL állományhoz. A *middleware*-ek valamilyen feladat végrehajtására alkalmas bővítmények. A HTTP kérelem és válasz során kerülnek lefuttatásra. Például az *AuthenticationMiddleware* minden beérkezett kérelmet kibővíti az éppen bejelentkezett felhasználó objektummal, amire a nézetfüggvényen belül tudunk hivatkozni. Miután a kérelem beérkezett a webalkalmazáshoz, a megfelelő URL alapján a *HTTP Request* objektum átadódik a megfelelő nézetfüggvénynek. Ez az objektum szerepel a nézetfüggvény első paraméterében, amit szabvány szerint *request*-nek nevezünk. Ezután végrehajtásra kerül az MVT architektúránál részletezett folyamat, amely során a függvény elő fog állítani egy *HTTP Response* objektumot, amit már vissza lehet küldeni a kliensnek. Ez az objektum előállítható például a *return HttpResponse('HTML kód')* parancs használatával, amelynek paraméterül egy HTML kód adható meg szöveges formátumban. Ennek a kódnak az eredménye fog megjelenni ebben az esetben a felhasználónak az oldalon. Továbbá megadhatóak a már előállított template fájlok, a *return render(request, 'template_fájl_neve.html', context)* paranccsal. Ilyenkor a megadott HTML fájl fog megjelenni. Első paraméternek a beérkezett *request* objektumot kell megadni, majd a template fájlhoz az elérési útját. A harmadik paraméter egy Python szótár, amely tartalmazza azokat a kulcs-érték párokat, amiket át akarunk adni a template-nek, hogy ott kerüljön feldolgozásra a DTL nyelv segítségével. A Django *template tag*-ek és a JavaScript is lehetőséget ad arra, hogy a kliens oldalon hajtsunk végre funkcionálisokat. Azonban az ajánlás az, hogy inkább a nézetfüggvény tartalmazza ezeket a kódrészleteket, és a kliens oldalon csak olyan funkciók legyenek, amiket csak ott lehet elvégezni. Az előállított objektum ezután a WSGI interfész felé lesz elküldve. Az objektum keresztül megy mindazon *middleware*-n, melyen a kérelem beérkezése során is, csak fordított sorrendben. A WSGI ezután átalakítja a megkapott objektumot küldésre alkalmas formátumra, majd a webservert átadva elküldésre kerül a HTTP válasz a kliensnek.

2.1.6. Adatbáziskezelés

A Django többféle adatbázist is támogat. Többek között PostgreSQL, MariaDB, MySQL, Oracle és SQLite adatbázisokat.² Alap esetben az SQLite adatbázist használja, ez az alap konfiguráció módosítása nélkül teljesen működőképes tud lenni. Azonban előfordulhat, hogy ez az adatbázis nem felel meg a programozónak. Ezt a keretrendszert használva nagyon könnyen lehet az egyes adatbázisokat lecserélni. Csupán elég a *settings.py* beállításain módosítani és máris egy másik adatbázis lesz használatban.

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': BASE_DIR / 'db.sqlite3',
    }
}

DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql_psycopg2',
        'NAME': 'my_database_name',
        'USER': 'my_user',
        'PASSWORD': 'my_password',
        'HOST': 'localhost',
        'PORT': '',
    }
}
```

9. ábra
Django adatbázis konfiguráció

A 9. ábra bemutatja, hogy mennyire hasonló tud lenni két különböző típusú adatbázis konfigurációja. A felső kódrészlet az SQLite alap beállításait mutatja be, az alatta lévő pedig az általam létrehozott PostgreSQL adatbázis konfigurációját. Az adatbázis típusokat ténylegesen néhány sor segítségével lehet cserélni, és az alkalmazás többi részén nem szükséges változtatásokat végezni. Ez a Django ORM-nek (Object-Relational Mapper) köszönhető. Ennek a használatával nem szükséges az egyes adatbázisok nyelvezetét megtanulnia a fejlesztőnek. Egyszerű Python kód segítségével tudunk adatokat menteni az adatbázisba, annak típusától függetlenül.

² A Django által támogatott adatbázisok teljes listája a dokumentációban tekinthető meg: <https://docs.djangoproject.com/en/3.2/ref/databases/>

2.2. A Django Channels és a WebSocketek

Ebben a fejezetrészben a Django Channels kerül bemutatásra. Ennek a technológiának a segítségével valósítottam meg a chat működését a szerver oldalon, a kliens oldalon pedig WebSocket kezelést alkalmaztam, amelyről szintén szó esik ebben a fejezetben.

2.2.1. A Django Channels kialakulása

A Django Channels nyújt megoldást arra, hogy kezeljük a valós idejű, kétirányú kommunikációt a Django alkalmazás és a felhasználó között, WebSocketek segítségével. Mikor a WebSocketek megjelentek, nem igazán volt kiforrott megoldás ennek a megvalósítására. Azelőtt statikus weboldalak voltak, a hagyományos HTTP kérelem-válasz segítségével történt a weboldalak megjelenítése. A WebSocketek megjelenésével azonban lehetőség nyílt valós idejű funkciók létrehozására. Ezeknek a funkcióknak az eléréshez a sima HTTP protokoll nem volt megfelelő. Ennek az oka, hogy a protokollt arra tervezték, hogy választ adjon a klientsztől érkező kérelemre, egyszeri alkalommal, ami után a kapcsolat le is zárul. Ezzel szemben, a WebSocketek folyamatos, akár hosszabb ideig tartó TCP/IP kapcsolatot tartanak fenn a szerver és a kliens között. Ez a kétirányú és full-duplex csatorna lehetővé teszi, hogy bármelyik fél üzenetet küldjön a másiknak, késleltetés nélkül, bármikor. Amint a kapcsolat kiépült, fenn is fog maradni egészen addig, amíg azt az egyik fél le nem zárja. A HTTP protokoll esetében mindig ki kell építeni a kapcsolatot minden egyes kérelem-válasz során. Ezáltal a WebSocketek használatával sokkal gyorsabb kommunikációt érhetünk el. A WebSocket kapcsolat kiépítése HTTP-n keresztül zajlik. Ilyenkor a szerver és a kliens megegyezik, hogy inntől kezdve WebSoceten keresztül kommunikálnak tovább. Ezzel a TCP kapcsolat továbbra is fenn fog maradni a HTTP válasz után is. A Django Channels egy új technológia, egy projekt, ami kiegészíti az alap Django keretrendszer funkcionalitását. Az első változata 2017-ben jelent meg. Készítője Andrew Godwin, a Django egyik fő fejlesztője. Ennek a projektnek a használatával az alkalmazás képes kezelni hosszan tartó kapcsolatokat igénylő protollokat is. Ilyen például az előbb részletezett WebSocket kapcsolat is.

2.2.2. WSGI és ASGI interfészek

Ahogy azt már korábban említettem, a WSGI (Web Server Gateway Interface) és az ASGI (Asynchronous Server Gateway Interface) interfészként szolgálnak a webalkalmazás és a

webszerver között. Feladatuk, hogy kezeljék a kientől érkező kérelmeket. Egyszerre csak az egyik fajta interfész működhet. Fő különbség a kettő között abban rejlik, ahogy a beérkező kérelmeket kezelik. A WSGI egyszerre csak egy beérkező kérelmet tud kezelni. Miután választ küldött az adott kérelemre, akkor tudja a következő kérelmet feldolgozni. Ezáltal a kérelmek szekvenciálisan vannak feldolgozva, szinkron módon. Ezzel szemben az ASGI egyszerre több kérelmet is tud kezelni aszinkron módon, ezáltal a legtöbb esetben sokkal gyorsabb működés érhető el. Az ASGI tekinthető a WSGI egy újabb verziójának, hiszen ugyanúgy tudja kezelni a beérkező kérelmeket, mint a WSGI, csak sok esetben sokkal hatékonyabban. Az általam készített szoftver szempontjából elengedhetetlen az ASGI használata, hiszen a WebSocket kapcsolat is aszinkron, tehát a másik interfész alkalmatlan lenne erre a feladatra.

2.2.3. Consumer-ek bemutatása

A consumer tartalmazza azt a kódot, amellyel a webalkalmazás kezeli a WebSocketen keresztül érkező, kientől származó kérelmeket. Hasonló a Django MVT architektúra View részéhez. A szoftver fejlesztése során generikus consumer-eket használtam. Ezeknek a lényege, hogy adott funkcionalitásokat, amely kezeli a WebSocket kérelmeket nem kell újraírni, hanem a már előre megírt funkcionalitást csak ki kell bővíteni. Ennek köszönhetően gyorsabban haladhat a fejlesztés. Ezeknek a consumer-eknek több fajtája is van:

- `WebsocketConsumer`
- `AsyncWebsocketConsumer`
- `JsonWebsocketConsumer`
- `AsyncJsonWebsocketConsumer`
- `AsyncHttpConsumer`

A programozó ezek közül a consumer-ek közül választhat, attól függően melyik illik a legjobban az elkészítendő programhoz. Az általam készített szoftver az *AsyncJsonWebsocketConsumer*-t használja. Aszinkront választottam, mivel úgy gondoltam, hogy a fejlesztés során hasznát tudom majd venni az aszinkron végrehajtásnak, valamint korábban már dolgoztam JSON formátumú objektumokkal Python-ban, ezért evidens volt, hogy itt is egy ilyen fajta consumer-t fogok választani. A 10. ábra egy, a szoftverből kivágott és leegyszerűsített consumer-t ábrázol, ami alapján bemutatom ennek a fájlnak a működését. A Django projektben a consumer egy sima *.py* kiterjesztésű állomány. Az elterjedt elnevezése ennek a fájlnak *consumers.py*. Ez a fájl az adott app-hoz tartozik, tehát az egyes app-oknak saját

`consumers.py` fájlja lehet. Látható, hogy egy sima Python osztályként definiálható egy consumer, amely az adott típusú consumer-t terjeszti ki. Az első metódusa a **`connect()`**, amely a WebSocket kapcsolat létrehozásakor fut le. A példában a csatlakozási kérelem elfogadásra kerül, azonban lehetne bizonyos ellenőrzéseket végezni, és egy *if-else* szerkezettel akár meg is lehetne tagadni a csatlakozást. Ebben az esetben a WebSocket kapcsolat nem jönne létre. A consumer meghívásakor elérhető az úgynevezett *scope* objektum is. Ebben az objektumban rengeteg információ van a beérkezett kérelemről, amelyet a `self.scope` paranccsal érhetünk el. Ahogy a példában is látható, a `user` kulcsszóval elérhetjük azt a felhasználót, aki csatlakozni akar WebSocketen keresztül. A **`disconnect()`** metódus a WebSocket lezárulásakor fut le. Ebben az esetben is megadhatjuk, hogy milyen funkcionalitást hajtson végre a szoftver.

```
class PrivateChatRoomConsumer(AsyncJsonWebsocketConsumer):
    async def connect(self):
        print('PrivateChatConsumer: ' + str(self.scope['user']) + ' connected')

        # csatlakozás elfogadása
        await self.accept()

    async def disconnect(self, code):
        print('PrivateChatConsumer: ' + str(self.scope['user']) + ' disconnected')

    async def receive_json(self, content, **kwargs):
        command = content.get('command', None)
        room_id = content.get('room_id')
        user = self.scope['user']

        if command == 'send':
            # üzenet küldése a kliensnek
        elif command == 'join':
            # felhasználó hozzáadása az online felhasználókhoz
        elif command == 'leave':
            # felhasználó eltávolítása az online felhasználóktól

@database_sync_to_async
def save_private_chat_room_message(user, room, message):
    return PrivateChatRoomMessage.objects.create(user=user, room=room, content=message)
```

10. ábra
A `consumers.py` alap függvényei

A **`receive_json()`** metódus a kientől érkező üzeneteket kezeli, és kapja meg az első paraméterként Python szótár formátumban. Az üzenet JSON string formátumban érkezik a kientől, azonban ez átalakításra kerül automatikusan. Ezáltal sima Python kóddal egyszerűen

kezelni tudjuk. A példában a kientől különböző parancsok érkezhettek, és az alapján a consumer végrehajtja a feladathoz tartozó funkcionalitást.

Az alap függvényeken kívül megadhatunk számos egyéb függvényt, a consumer-en belül és kívül is. Fontos megemlíteni, hogy a consumer-en belül, mivel egy aszinkron consumer-ről van szó, csakis aszinkron függvényeket lehet megadni. Az osztályon kívül megadható nem aszinkron függvény is, azonban hogyha azt használni akarjuk a consumer-en belül, át kell alakítani aszinkronná. Ezt a példában látható `@database_sync_to_async` dekorátorral érhetjük el. Erre azért is szükség van, mivel ezek a függvények legtöbbször adatbázis műveleteket hajtanak végre (a példában is egy privát üzenet objektumot hozok létre az adatbázisban), és az adatbázis műveletek szinkron műveletek.

Ahhoz, hogy a Django projekt tudja, hogy az adott WebSocket kapcsolathoz melyik consumer tartozik, létre kell hozni egy konfigurációs fájlt (11. ábra). Ez a `routing.py` nevű állomány az általam készített szoftverben. Látható az összes consumer és elérési útjaik. Ezeket az elérési utakat a kliens oldalon használjuk majd fel JavaScriptben, ezt az ehhez kapcsolódó fejezetben fogom ismertetni.

```
from channels.auth import AuthMiddlewareStack
from channels.routing import ProtocolTypeRouter, URLRouter
from channels.security.websocket import AllowedHostsOriginValidator
from django.urls import path
from public_chat.consumers import PublicChatRoomConsumer
from private_chat.consumers import PrivateChatRoomConsumer
from team.consumers import TeamConsumer
from notification.consumers import NotificationConsumer

application = ProtocolTypeRouter({
    'websocket': AllowedHostsOriginValidator(
        AuthMiddlewareStack(
            URLRouter([
                path('kozossege/<room_id>/', PublicChatRoomConsumer.as_asgi()),
                path('uzenetek/<room_id>/', PrivateChatRoomConsumer.as_asgi()),
                path('csapat/<room_id>/', TeamConsumer.as_asgi()),
                path('', NotificationConsumer.as_asgi()),
            ])
        )
    ),
})
```

11. ábra
A `routing.py` konfigurációs fájl

Ez a fájl több szintből épül fel. A `websocket` kulcsszóval adjuk meg, hogy az ilyen típusú kapcsolatokat kezelje az ASGI. A `ProtocolTypeRouter` segítségével megadhatjuk, hogy milyen

típusú ASGI app-ot készítünk. Az *AllowedHostsOriginValidator* az engedélyezett domain-eknek a halmazát adja meg, ennek akkor van jelentősége, hogyha a szoftvert egy éles webserverre helyezzük át. Az *AuthMiddlewareStack* biztosítja azt, hogy a *scope* objektumhoz hozzáférjünk a consumer-en belül. A *URLRouter* pedig azoknak az elérési utaknak a listája, ahova a WebSocketeket csatlakoztatjuk.

2.2.4. A Channel Layer-ek szerepe

A Channel Layer egy mechanizmus, ami lehetővé teszi, hogy az alkalmazás különböző példányai között kommunikáció jöhessen létre. Ennek a segítségével lehet megvalósítani az általam készített szoftverben, hogy az egyes chatszobákba beküldött üzeneteket ne csak az adott felhasználó lássa, hanem mindenki más is abban a szobában, valós időben. Tehát ezzel a mechanizmussal valósítható meg a broadcast a felhasználók között. A folyamat során az egyes példányoknak a consumer-ei között zajlik a kommunikáció. A Channel Layer használata során rendelkezésünkre állnak csoportok és csatornák (channel-ek). Az egyes csatornákat lehet tekinteni az egyes felhasználókhoz tartozó postaládának is. Ezen keresztül lehet az adott felhasználónak, akik tulajdonképpen az alkalmazásnak egy példányai, üzenetet küldeni. Az adott felhasználóhoz tartozó consumer pedig kezelni fogja a beérkező üzenetet. Minden csatornának saját neve van, ennek az ismerete szükséges ahhoz, hogy az üzenetet el lehessen küldeni az adott csatornába. Csatornáknak egy kollekciónak nevezzük csoportnak. Egy csatornát hozzá lehet adni, illetve el lehet távolítani egy csoportból. Az általam készített szoftverben ezek a csoportok a chatszobák, amelyekben az egyes felhasználók benne vannak. Feltételhez lehet kötni, hogy az egyes chatszobához ki csatlakozhat, hiszen hogyha nem adódik hozzá az adott csatorna az adott csoporthoz, a csatornához tartozó felhasználó nem fogja megkapni az üzeneteket. Ezzel a módszerrel ezen a szinten is korlátozni lehet, hogy ki csatlakozhat a szobához, nem csak a WebSocket kapcsolat kiépítése során, amelyet a consumer-ek bemutatásánál említettem. A csoportok segítségével lehet megvalósítani a broadcast üzenet küldést a felhasználók között. Mivel a csoportoknak is van nevük, így a csoport nevére hivatkozva el lehet küldeni az összes benne lévő csatornán keresztül az adott üzenetet a felhasználóknak.

2.2.5. WebSocket kezelés JavaScriptben

Ahhoz, hogy a felhasználó, azaz a kliens kommunikálni tudjon a szerverrel és a webalkalmazás consumer-eivel WebSocketen keresztül, WebSocket kapcsolatot kell létrehozni. A szoftverben ehhez a WebSocket API-t használtam. Ennek a technológiának a segítségével lehet a böngészőből WebSocket kapcsolatot kezdeményezni a szerver felé. A 12. ábra bemutatja, hogyan lehet JavaScriptben létrehozni ilyen típusú kapcsolatot.

```
var wsScheme = window.location.protocol == "https:" ? "wss" : "ws";
var wsPath = wsScheme + "://" + window.location.host + "/uzenetek/" + roomId + "/";

// websocket beállítás
privateChatRoomSocket = new WebSocket(wsPath)
```

12. ábra
WebSocket beállítás JavaScriptben

Először meg kell határozni a kapcsolat típusát. A *wss* kulcsszó jelzi a biztonságos, titkosított kapcsolatot, azaz mikor a HTTPS protokoll van használatban. A *ws* kulcsszó a HTTP protokoll használatát jelzi, azaz a nem biztonságos, titkosítatlan kapcsolatot. Az elérési útnak, amely az ábrán a *wsPath*, meg kell adni az előbb leírt kapcsolat típust, majd a *routing.py* fájlban definiált elérési utat. Ezzel lesz a megfelelő consumer-hez hozzákötve a socket. Ezután létre kell hozni magát a WebSocketet, paraméterül pedig megadni azt a URL-t, amelyet az imént állítottunk elő. Erre a címre fogja a szerver a választ küldeni. Ilyenkor a kliens egy HTTP kérelmet küld a szervernek, amelyben a kapcsolatot WebSocket kapcsolatra szeretné váltani. Ekkor lefut a consumer-ben található *connect* függvény is. Miután létrejött a kapcsolat, a kliens és a szerver a továbbiakban WebSocketen keresztül kommunikál egymással. A kommunikáció JSON szöveges formátumú keretekkel történik, a consumer típusa miatt. A szerver oldalon a consumer küldi ezeket a kereteket, amelyek Python szótárak kezdetben, azonban a Django alkalmazás ezt automatikusan átalakítja az átküldéshez megfelelő formátumra. A WebSocket objektumhoz különböző beépített eseményfigyelő függvények tartoznak, amellyel érzékelni tudja ezeknek a kereteknek a beérkezését is, de sok egyéb más eseményt is fel tud dolgozni. A 13. ábra ezek közül az események közül mutat be néhányat, melyeket az alkalmazásban is használok.


```

// websocket megnyitása
notificationSocket.onopen = function(event) {
| console.log("A kapcsolat sikeresen létrejött");
};

// bejövő üzenet kezelése a consumertől
notificationSocket.onmessage = function(event) {
| console.log("Beérkezett adatcsomag a consumertől: ", event.data);
| var data = JSON.parse(event.data);
};

// hibaüzenetek kiírása
notificationSocket.onerror = function(event) {
| console.log("Hiba");
};

// websocket bezárása
notificationSocket.onclose = function(event) {
| console.log("A kapcsolat bezárult");
};

```

13. ábra
WebSocket eseménykezelő függvények

Látható, hogy például az *onmessage* függvény a beérkezett adatcsomagokat kezeli. Ezek az adatcsomagok JSON stringként érkeznek a klienshez. Először átalakításra kerül a **JSON.parse** paranccsal JavaScript objektummá, amely után a benne lévő adatokat fel tudja használni a program.

2.2.6. A valós idejű kommunikáció folyamata

A következőkben röviden bemutatom, hogyan zajlik a kommunikáció a kliens és a Django alkalmazás között, a Django Channels-t és a WebSocketeket használva. Ez a folyamat részletesebben is be lesz mutatva, mikor az alkalmazáson belül ismertetem ennek lépéseit. Ezt a fejezetet az előbb leírtak összefoglalásának szánom.

A már ismertetett WebSocket kapcsolat kiépítésével indul a folyamat. A kliens az adott oldal megnyitásakor kezdeményezi ezt a kapcsolatot. Ilyenkor a szerver oldalon a megnyitott oldalhoz kapcsolódó consumer-ben lévő *connect* függvény fut le, valamint a kliens oldalon szereplő *onopen* eseményfigyelő függvény is. Ezt követően, ahogy létrejött a kapcsolat, lefut az az eseményfigyelő függvény a kliens oldalon, ami a WebSocket kapcsolat létrejöttét figyeli. Ez a függvény a szoftverben egy csatlakozási parancsot küld a szervernek, de hasonló módon történik az egyéb parancsok küldése is a program különböző részein. A küldés a *send()* parancs segítségével történik. Az adat JSON string formátumra lesz átalakítva még a kliens oldalon,

ami a megfelelő formátum az adatsomag küldéséhez. Ezt az elküldött adatsomagot a consumer *receive_json* függvénye kezeli le. A webalkalmazás automatikusan átalakítja a JSON string formátumú adatsomagot Python szótárrá, így a programozónak csak egyszerűen ki kell szednie a megfelelő kulcs-érték párokat a consumer-ben. A függvény elején a szótárból kinyerésre kerül, hogy milyen parancs érkezett a kientől, és egy *if-else* szerkezetet használva lefutnak a megfelelő függvények a parancs végrehajtásához. Ezután a szerver valamilyen válasz adatsomagot küld a kliensnek. A *group_send()* parancs segítségével történik az adatsomag csoportba küldése, amiben a csatornák vannak. Az egyes klienseknek, azaz csatornáknak, a *send_json()* parancs használatával történik az adatsomag elküldése. Ez a parancs JSON formátumra alakítja a Python szótárat. A kliens *onmessage* függvénye érzékeli azt, hogyha valamilyen adatsomag jön a szerver felől. Ez a függvény az előző részben részletezett módon megfelelő formátumra alakítja az adatot és megjeleníti a kliensnek. Ezáltal a weboldalon valós időben jeleníthető meg adat anélkül, hogy az oldalt frissíteni kellene. A kommunikáció vagy a kapcsolat során, ha bármilyen hiba lép fel, akkor a kliens oldalon lévő *onerror* függvény fut le, a szerver oldalon pedig a megfelelő hibakezelő függvény.

2.3. A front-end technológiái

A most következő fejezet részben a kliens oldalon használt technológiák kerülnek bemutatásra. Ezeknek a technológiáknak a használatával alakítottam ki a szoftver felületét, és tettem még interaktívabbá a weboldalt.

2.3.1. HTML, CSS és JavaScript

A weboldal kinézetének a magját természetesen a HTML (HyperText Markup Language) alkotja. Az egyes HTML fájlok, amelyek sima szöveges fájlok *.html* kiterjesztéssel, írják le a weboldal szerkezetét, és tartalmazznak egyéb alap információkat, például, hogy milyen karakterkódolást használjon a böngésző.

A CSS (Cascading Style Sheets) felel a weboldal stílusáért, megjelenéséért. Ennek a nyelvnek a segítségével lehet megadni, hogy az egyes HTML elemek hogyan helyezkedjenek el a weboldalon és hogyan nézzenek ki. Bármilyen formázást, amit a weboldalon végrehajtottunk ez a fájl tartalmazza alapesetben.

A JavaScript egy objektum-orientált programozási nyelv. Használatával bővíthetők a kliens oldalon a weboldal funkcionalitásai. Animációk létrehozáshoz, felhasználói interakció kezeléséhez és a szerverrel való kommunikációhoz egyaránt alkalmas. Továbbá, dinamikusan módosítható a CSS és HTML fájlok kódja is a segítségével. A szoftverben a szerver oldalon használt Python programozási nyelv mellett, ebben a nyelvben íródott a legtöbb funkcionalitás. Összességében ez a három nyelv együttműködése szükséges ahhoz, hogy a kliens oldalon egy jó felhasználói élményt nyújtó weboldalt hozzunk létre.

2.3.2. A jQuery keretrendszer

A jQuery egy JavaScript könyvtár. Rendkívül megkönnyíti és egyszerűsíti a JavaScript kódot. A használatával a HTML elemek elérése, azoknak a stílusának és tartalmának módosítása gyors és egyszerű. Továbbá, animációk létrehozása és azoknak különböző attribútumainak a beállítása néhány sorral megvalósítható. Ezeken kívül alkalmas egyéb, sima JavaScripttel is megoldható feladat megvalósítására, például az eseményfigyelő függvények beállítását is képes elvégezni. A 14. ábra bemutatja a jQuery használatát a gyakorlatban. Látható, hogy \$ jelet kell tenni az utasítás elejére, majd az adott HTML elemre hivatkozni zárójelek között. Ilyenkor a CSS-ben használatos jelek használhatóak, tehát a . az osztályt, míg a # jel az ID-t jelöli. Meg lehet adni az elemnek csak a típusát is, például az *img* vagy *div* kulcsszavakat. Ilyenkor az összes ilyen típusú elemre érvényes lesz a kód. Ezeknek a használatát lehet kombinálni is, tehát meg lehet adni egy ID-t majd egy elem típusának a nevét. Így lehet szűkíteni a kiválasztott elemeket.

```
$(".loader-container").fadeOut("slow")           // osztály elrejtése animációval
$("#arena-title-center h1").html("Aréna")       // adott div-ben a h1 elem tartalmának módosítása
$("#main-page-button").css("visibility", "hidden") // adott elem css attribútumának módosítása

// adott div háttér színének a láthatóságának a módosítása animációval
$("#timer").animate({
  backgroundColor: "rgba(255, 0, 0, 0.2)"
}, 600, function(){
  $(this).animate({
    backgroundColor: "rgba(255, 0, 0, 1.0)"
  }, 600)
})
```

14. ábra
A jQuery használata

További hasznos függvényei közé tartozik az *animate()*. Ennek a használatával lehet módosítani egy elemnek a megjelenésén animációval. A példában látható, hogy a *#timer* div-

nek az áttetszőségét állítom át egy kisebb értékre 600 milliszekundum alatt, amivel az elem halványabb lesz. Majd az animáció végrehajtása után lefut egy *callback* függvény, ami visszaállítja az eredeti értékre, szintén 600 milliszekundum alatt. Ezzel egy villogó effektust lehet elérni.

2.3.3. Az AJAX technika

Az AJAX (Asynchronous JavaScript And XML) egy technika, amely a weboldal interaktivitásaért nagy mértékben felelős. A használatával a kliens oldalról adatokat tudunk küldeni és fogadni a szervertől, az oldal frissítése nélkül. A folyamat során egy egyszerű HTTP kérelem kerül elküldésre a szervernek. A szerver visszaküld egy választ, a megfelelő adattal, amelyet JavaScriptben fel lehet dolgozni és megjeleníteni a weboldalon. A szerver oldalon ezt a megfelelő nézetfüggvénnyel kezeljük. Itt a különbség, hogy a függvény nem egy HTML oldalt hoz létre és küld el, hanem az általunk meghatározott típusú objektumot. A szoftverben ez egy JSON objektum. A nézetfüggvényben ezt a *return JsonResponse(data)* paranccsal tudjuk megtenni, ahol a *data* egy Python szótár, amely az elküldésre szánt adatokat tartalmazza. Ezeket a nézetfüggvényeket ugyanúgy kell kezelni, mint a hagyományos társait. Tehát ugyanúgy hozzá kell őket adni a *urls.py* fájlhoz. Ezekben az esetekben különösen fontos, hogy az adott URL-nek beállítsuk a *name* attribútumát, hiszen a kliens oldalon azt használjuk, hogy meghívjuk az adott függvényt. A szoftver bemutatásánál még jobban el lesz magyarázva ennek a technológiának a gyakorlati működése.

3. A FELADAT BEMUTATÁSA ÉS A SZOFTVER ISMERTETÉSE

Ebben a fejezetben az elkészült szoftvert mutatom be részletesen. Először a feladat leírása, majd a futtatáshoz szükséges követelmények lesznek olvashatóak. Ezután a szoftver egyes funkcióinak a használatát mutatom be. Végül a szoftver felépítéséről lesz szó, ahol részletezem mi történik a programban kód szinten, az egyes funkciók végrehajtásakor.

3.1. A feladat bemutatása

Az általam készített szoftver egy online szerepjáték, amit másokkal együtt játszhatunk. Minden játékosnak van egy karaktere, amit fejleszteni tud. A játék célja, hogy a karakterünk minél fejlettebb és erősebb legyen, és ezáltal a legelső helyet foglalja el a ranglistán. A regisztráció során három különböző karaktertípusból választhatnak a felhasználók. Ezeknek a karakter típusoknak eltérő tulajdonság értékeik vannak. Mindegyik típusnak más a fő tulajdonsága, amely azt az adott karaktert erősebbé teszi. Ennek a megvalósításához létre kellett hozni egy felhasználói modellt, amelyben benne van egyrészt minden szükséges információ az adott fiókkal kapcsolatban, valamint a felhasználó által választott karakter tulajdonságai is. A játékban arannyal lehet fejleszteni a karaktert. Aranyat különböző nehézségű minijátékokkal lehet szerezni. Ezeket a játékokat el kellett készíteni, és megoldani, hogy mikor egy felhasználó befejez egy játékot, és azt meg is nyerte, a megfelelő mennyiségű arany jóváírásra kerüljön a felhasználónak az adatbázisban. A játékosok tudnak egymással harcolni, amelyet a szoftver szimulál le. Ehhez létre kellett hozni egy logikát, amely eldönti, hogy melyik játékos nyeri a harcot, figyelembe véve a megfelelő attribútum értékeit az egyes karaktereknek. A felhasználók létre tudnak hozni csapatokat, és csapatba rendeződve megküzdeni egymással. Ennek a megvalósításához is szükség volt egy jól felépített modellre, illetve az előbb említett logikára, ami eldönti, hogy az épp egymással harcoló karakter közül melyik az erősebb. A játékosok tudnak egymással írásban beszélgetni is chaten keresztül. Három különböző típusú chat áll rendelkezésre a szoftveren belül. Van a publikus chat, ahova az összes felhasználó megkötés nélkül írhat mindenkinek. A második a privát chat, ahol egy felhasználó egy másikkal tud beszélgetni privátban. A harmadik pedig a csapat chat, amelyet csak az egyes csapatok

csapattagjai láthatnak és használhatnak. Létre kellett hozni a megfelelő logikát, amely létrehozza dinamikusan a csapat és privát chatszobákat (a publikus chatből mivel egy darab van, statikusan lett létrehozva), és a modellt, amely leírja az egyes chatszobáknak a felépítését. A szobák után meg kellett oldani, hogy valós időben, és megfelelően működjön a chat a felhasználók között. A szoftverben létrehoztam egy értesítési rendszert is, amely jelzi, hogyha új üzenete érkezett a felhasználónak az egyes chateken. Hasonlóan a chathez, itt is meg kellett oldani, hogy ez valós időben történhessen és jelezze is a felhasználónak. Továbbá rengeteg kisebb funkció van, amelyet a későbbiekben részletezni fogok. Összességében meg kellett valósítani, hogy a kliens és a szerver hatékonyan és hiba nélkül tudjon kommunikálni egymással, és adatot küldeni egymás között valós időben. A felület kinézetére is nagy hangsúlyt fektettem. Egy jól felépített HTML struktúrát kellett létrehozni, amely stabil alapja lehetett a weboldalnak, majd ezt követően megírni a hozzá tartozó CSS és JavaScript fájlokat is.

3.2. A szoftver ismertetése

Ebben a fejezetrészen részletesen ismertetem a szoftver felépítését és szerkezetét. Bemutatom, hogyan működik a szoftver használat közben, valamint ismertetem a szoftver futtatásához szükséges alapkövetelményeket.

3.2.1. A szoftver futtatásának követelményei

A szoftver fejlesztését és tesztelését Windows 10 operációs rendszeren végeztem, ezért ez az ajánlott a szoftver futtatásához. A szoftver Linux alatt is működőképes, azonban a felhasznált modulok és beállítások okozhatnak nem várt futtatási hibákat az eltérő operációs rendszer miatt, ezeket alaposan tesztelni kellene. A chatnek a működéséhez fel kell telepíteni a Redis-t. Ez a program szükséges ahhoz, hogy a Django Channels-ben használt csoportokban lévő felhasználók üzeneteit mindenki megkapja az adott chatszobában. Ez egy kulcs-érték adatbázis lényegében, amely tartalmazza a programban létrehozott csatornák és csoportok neveit, ezzel működtetve az üzenetek szétküldését az egyes csatornákon keresztül. Ez azonban nem működik Windows 10-en, ezért egy alternatívát kerestem ennek a megoldására. A Memurai nevű alkalmazást telepítettem fel, amely a Redis funkcionalitását váltja ki, és működik Windowson

is.³ A szoftver fejlesztése kezdetekor a 3.1.2-es verzió volt a legújabb, ezért annak a használatát javasolom, annak is a *Developer* változatát. Ez a verzió ingyenesen letölthető. Az egyetlen hátránya, hogy 10 naponta újra kell indítani a szolgáltatást, de fejlesztői környezetben ez nem volt probléma. Valószínűleg az újabb változatokkal is gond nélkül futna, de ez nem lett letesztelve. Ezen kívül fel kell telepíteni még a Python-t is.⁴ A szoftver fejlesztéséhez a 3.9.13-as verziót használtam. Ezután a szoftver futtatásához célszerű létrehozni egy virtuális környezetet. Ilyenkor a további csomagok, például maga a Django is, ebbe a virtuális környezetbe lesz feltelepítve, így szeparáltan tudjuk tárolni a különböző verziójú projektjeinket. Virtuális környezet létrehozására Windows alatt a `python -m venv [virtuális környezet neve]` parancsot használhatjuk a parancssorban. Ezzel legenerálódik egy mappa, a virtuális környezet nevével, amit meg kell nyitni a `cd [virtuális környezet neve]` parancs kiadásával. A környezetet ezután aktiválni kell, a `Scripts\Activate` parancssal. Ezt követően fel lehet telepíteni a Django-t. A szoftver fejlesztése elején a 3.2.13-as verziót választottam. Habár elérhető lett volna már a Django 4.0 is, nem azt választottam, mivel nem LTS kiadásról volt szó. Az általam választott verzió 2024 április végéig támogatott, ezért egy sokkal jobb választásnak bizonyult a számomra. A Django feltelepítéséhez a `python -m pip install Django==3.2.13` parancsot lehet használni. A projekt struktúrában azonban van egy `requirements.txt` nevű fájl, amely tartalmazza az összes olyan modult, amit a fejlesztés során használtam, köztük a Django-t is. Ezeket a modulokat egyetlen parancs, a `pip install -r requirements.txt` kiadásával fel lehet telepíteni, amely után az alkalmazás már használatra kész állapotban lesz. Az alkalmazás futtatáshoz a `python manage.py runserver` parancsot kell használni.

3.2.2. A szoftver használata

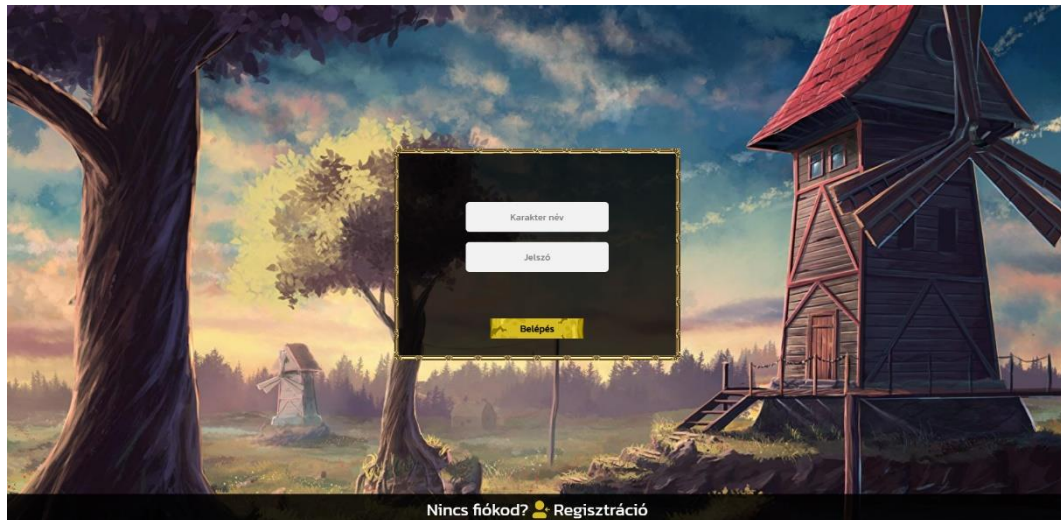
A szoftver használatának bemutatása előtt érdemes még egyszer letisztázni a játék lényegét: a minél magasabb szint, és minél erősebb karakter elérése, és ezáltal minél magasabb helyezés elfoglalása a ranglistán. A karakter attribútumait, azaz az erősségét arannyal tudjuk fejleszteni. A szintlépés pedig tapasztalati pontok szerzésével érhető el. Aranyat és tapasztalati pontot a minijátékok játszásával lehet szerezni. Abban az esetben, hogyha sikeresen lejátszunk egy játékot, aranyat és tapasztalati pontot szerzünk. A ranglistán pedig az lesz előrébb, aki minél

³ A Memurai a Redis egy alternatívája, amelyet használhatunk Windowson is. Innen tölthető le: <https://www.memurai.com/get-memurai>

⁴ A Python-t innen szerezhetjük be: <https://www.python.org/downloads/>

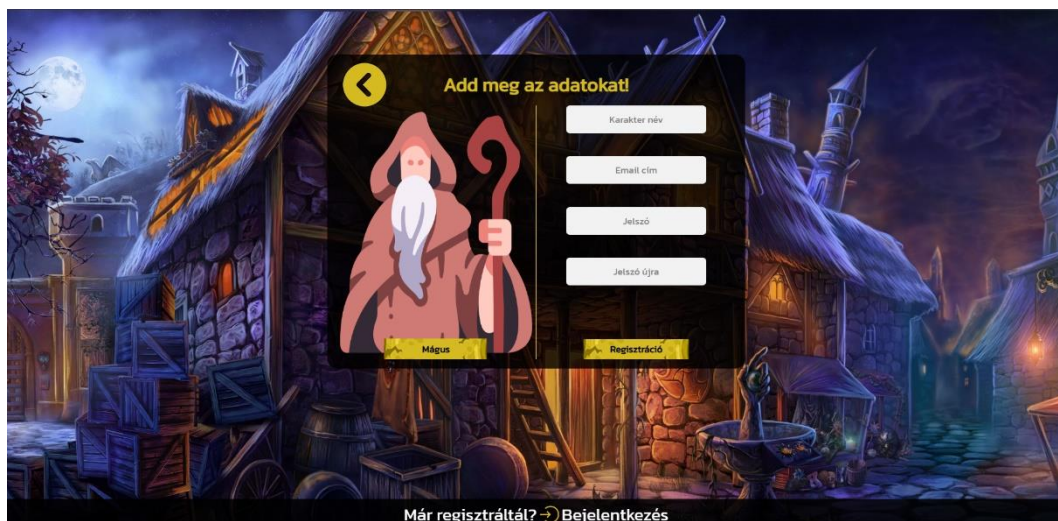
több becsületpontot gyűjt össze. Becsületpontot egy másik játékos legyőzésével lehet szerezni, az arénában vívott harcok után. Ha kikapunk, azzal becsületpontot veszünk. Tehát ha minél erősebb és fejlettebb a karakterünk, annál több játékost tudunk legyőzni, és annál kevesebb eséllyel kapunk ki másoktól. Ezáltal előrébb lehetünk a ranglistán.

A következőkben a szoftver használatát fogom bemutatni felhasználói szemszögből. Mikor megnyitjuk a weboldalt, a bejelentkező oldal töltődik be legelőször (15. ábra).



15. ábra
A bejelentkező oldal

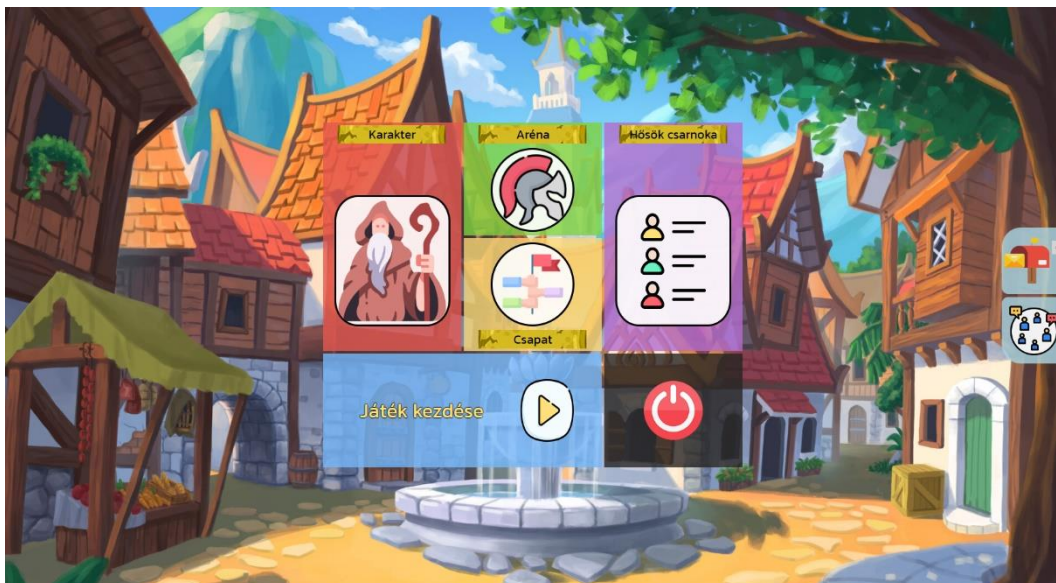
Ha már van profilunk, a felhasználónév és jelszó megadásával be tudunk jelentkezni. Abban az esetben, hogyha hibásan adjuk meg, vagy nem is létezik a megadott felhasználónév-jelszó páros hibaüzenetet kapunk. Abban az esetben, ha nincs profilunk, az oldal alján található *Regisztráció* fülre kattintva navigálhatunk el a regisztrációs oldalra. Ezen az oldalon először egy lapozósáv jelenik meg, amely a választható karakter típusokat mutatja be nekünk. Rendelkezésünkre áll egy *Kiválaszt* gomb, amelyre kattintva betöltődik egy regisztrációs űrlap, ahol az adatainkat tudjuk megadni (16. ábra). Ilyenkor a bal oldalon megjelenik az általunk választott karakter típusának a képe és neve is.



16. ábra
A regisztrációs űrlap

A weboldal az adatok megadásánál valós időben jelzi, hogyha az általunk megadott adat nem felel meg az elvárásoknak. Például, ha túl gyenge jelszót adunk meg, vagy az általunk választott felhasználónév foglalt, az adott bemenet alatt egy piros hibaüzenet jelenik meg. Abban az esetben, ha ennek ellenére is rányomunk a *Regisztráció* gombra, hibaüzenetet kapunk, ami kiírja, hogy mely adatok nem megfelelőek. Az űrlap bal felső sarkában található egy gomb is, amellyel vissza tudunk térni a karaktertípus választó oldalra. Az oldal alján található egy *Bejelentkezés* fül, amelyre kattintva vissza tudunk navigálni a bejelentkező oldalra. Sikeres regisztráció esetén az oldal automatikusan visszadob az előbb említett oldalra, ahol bejelentkezés után már használhatjuk is a játékot.

A bejelentkezés után a weboldal kezdőlapjára kerülünk (17. ábra). Az itt található menüpontokkal lehet elnavigálni az egyes oldalakra az alkalmazáson belül. A *Karakter* menüpontnál, a karakterünk típusának a képe fog megjelenni. Tehát az ábrán mutatott példában a karakterünk típusa mágus. A képernyő jobb szélén látható ikonok a privát (felső) és publikus (alsó) chatek ikonjai. A navigáció mellett ezen az oldalon működik az értesítési rendszer is. Hogyha egy felhasználó új üzenetet kap valamelyik chaten, az adott chathez tartozó menüpont elkezd villogni. A privát chat esetében az új üzenetek száma is kiírásra kerül. Az ábrán látható szürke háttérű 0 helyett az új üzenetek száma fog megjelenni piros háttérrel. Ha 9-nél több új üzenete van egy felhasználónak, akkor a 9+ felirat fog megjelenni. A csapatchat esetén a csapat feliratú menü fog villogni.



17. ábra
Az alkalmazás kezdőlapja

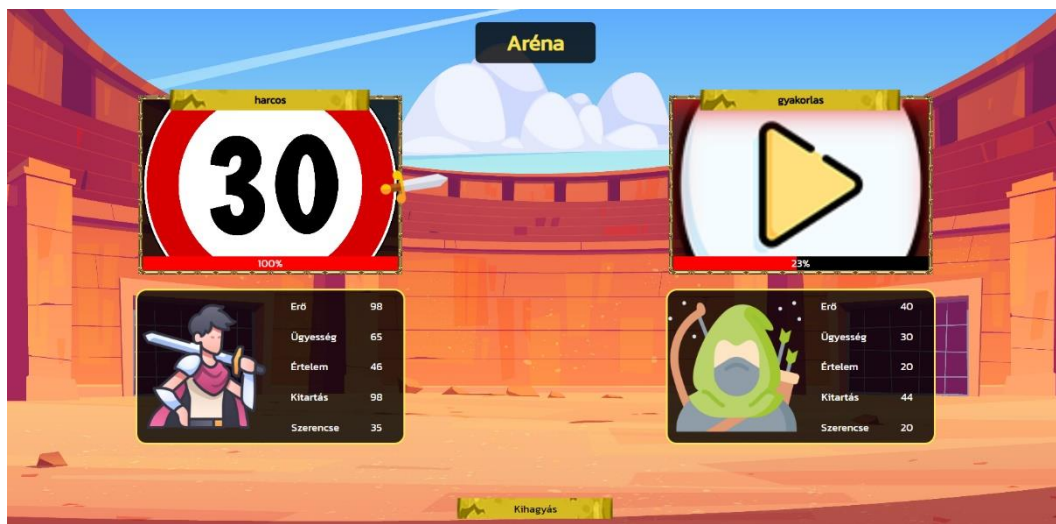
A *Karakter* menüpontra kattintva tekinthető meg a felhasználói profilunk (18. ábra). Az oldal tetején a felhasználónév látható. Az első belépés során egy alapértelmezett, üres kép jelenik meg a profilkép helyén. A profil szerkesztése földre kattintva tölthetünk fel új profilképet, törölhetjük a régit, valamint módosíthatjuk a felhasználónevet, a jelszót, és az email címet is. Jelszó megadásánál mindkétszer meg kell adni a jelszót. Abban az esetben, hogyha csak az egyik helyre írjuk be az új jelszót, érvénytelen lesz a módosítás. Ez abban az esetben is igaz, hogyha a két jelszó eltérő vagy nem elég erős. Erről hibaüzenetben tájékoztat minket a program. Sikeres jelszó módosítás során az alkalmazás kiléptet minket, ezért újra be kell lépni, már az új jelszóval. A szoftver a felhasználónév és az email cím módosításánál is ellenőrzést végez, és ha foglalt vagy nem megfelelő az új érték, szintén hibaüzenetet kapunk. A profilkép alatt látható a karakternek a szintje írásban és egy folyamatjelző sávval ábrázolva is. Ez dinamikusan változik, mikor tapasztalati pontokat szerzünk. Szintlépéskor a megszerzett tapasztalati pont nullázódik. Minél magasabb szintet érünk el, annál több tapasztalati pontra van szükség a szintlépéshez. Ezek alatt találhatóak a karakternek az egyes attribútumai. Ezeket az attribútumokat a felhasználó maga növelheti. Egy attribútum növelése egységenként 4 aranyba kerül. A felhasználó aranyainak a száma a másik oldalon látható. Ez a szám dinamikusan, valós időben változik ahogy a felhasználó növeli az attribútumainak az értékét. Hogyha már nincs elég arany az attribútumok növelésére, akkor a sárga háttérű plusz ikonok szürkévé válnak, valamint nem lehet a továbbiakban kattintani őket.



18. ábra
A felhasználói profil

A jobb felső sarokban látható a felhasználónak a leírása, amellyel be tudjuk mutatni magunkat más felhasználóknak, esetleg néhány érdekességet vagy vicces szöveget lehet ide írni. Leírást egyszerűen adhatunk meg: csak bele kell kattintani a téglalapba, majd begépelni a szöveget. A begépelés után, mikor kikattintunk a téglalapról, a szoftver érzékeli ezt az eseményt, és elmenti az adatbázisba az új leírást. A következő sorban látható, hogy melyik csapat tagjai vagyunk. Hogyha nem vagyunk egyik csapat tagja sem, akkor a *Jelenleg nincs csapata* felirat jelenik meg. A karakter rész alján látható még az, hogy mikor regisztráltuk a profilt az oldalra. Az oldal legalján jelenik meg a lejátszott meccseknek a száma, és hogy abból mennyi a győztes és a nyertes párbaj. Ezeknek a számát akkor láthatjuk, hogyha a megfelelő mező felé mozdítjuk a kurzort. Más felhasználóknak is meg lehet tekinteni a profilját. A különbség, hogy ilyenkor nem jelennek meg az attribútumokat növelő plusz ikonok, valamint a profil szerkesztése gomb sem. Helyette egy kard ikon és egy üzenet küldés felirat jelenik meg. A kard ikonra kattintva lehet az adott karaktert megtámadni, az üzenet küldésre kattintva pedig privát üzenet küldésre van lehetőség. Továbbá azt is láthatják más felhasználók, hogy mikor jelentkezünk be legutóbb az oldalra.

Az *Aréna* menüpontra kattintva lehetőségünk van a szoftver által ajánlott játékosok megtámadására. Ilyenkor a ranglistán körülöttünk lévő játékosokból szed ki kettő játékost ajánlott ellenfélnek. Megjeleníti a profilképüket, valamint az attribútumaikat egymás mellett. A kiválasztott játékosra kattintva kezdődik a harc szimulálása (19. ábra).



19. ábra
A párbaj szimulációja

A győztes 20 becsületpontot szerez, a vesztes ugyanennyit veszít el. Becsületpont nem lehet 0-nál kevesebb, tehát ha valaki kevesebb mint 20 becsületponttal elveszít egy párbajt, a becsületpontjainak a száma nem fog átesni mínuszba.

A ranglistát a *Hősök csarnoka* menüpontra kattintva érhetjük el (20. ábra). Itt becsületpont szerint vannak rendezve a játékosok és a csapatok is.

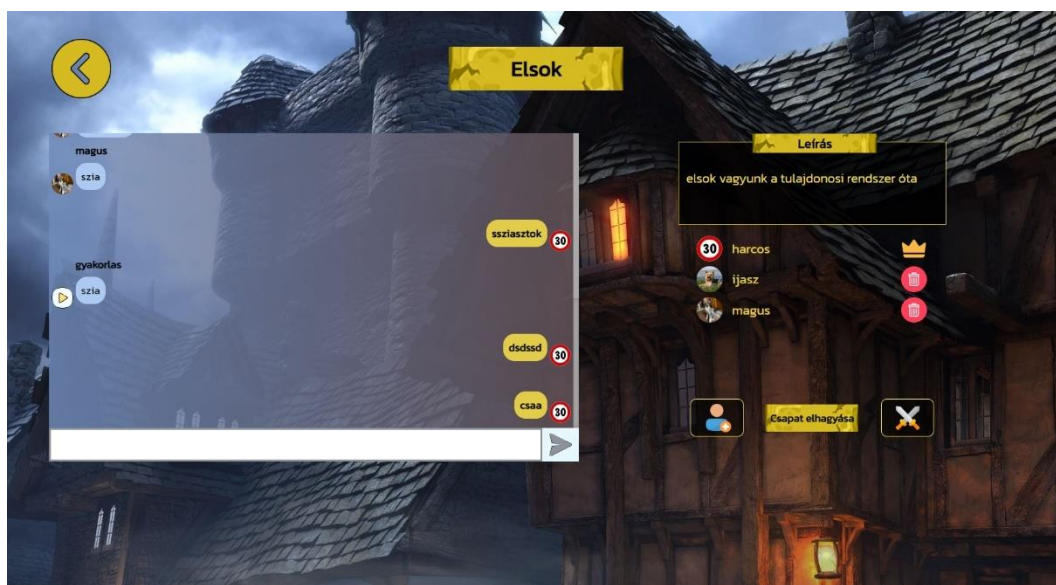


20. ábra
A játék ranglistája

Lehetőség van keresésre is, amely valós időben történik meg. Ahogy elkezdjük gépelni egy játékos vagy csapat nevét, a találatok egyből megjelennek az eredeti lista helyén. Az adott

felhasználóra vagy csapatra kattintva meg lehet tekinteni azoknak a profilját is. A játékosok és csapatok megjelenítése külön listában történik meg. Ezek között a listák között az oldal alján található kapcsolóval tudunk váltogatni. A saját csapatunknak és karakterünknek a neve fehérrel jelenik meg.

A *Csapat* menüpont annak a csapatnak az oldalát tölti be, amelynek a tagjai vagyunk (21. ábra). Abban az esetben, hogyha még egyiknek sem vagyunk a tagja, akkor erre a gombra kattintva tudunk csatlakozási kérelmet küldeni egy csapatnak, vagy csinálni egy sajátot magunknak. Saját csapat létrehozásakor egy csapatnevet kell megadni, valamint opcionálisan egy csapatleírást. A csapat létrehozásakor létrejön az ahhoz tartozó chat is, ahova a csapattagok tudnak majd üzeneteket küldeni egymásnak. Csatlakozási kérelmet egy csapat profiljára kattintva tudunk küldeni a *Csatlakozás* gombra kattintva. Több csapatnak is küldhető csatlakozási kérelem, azonban ahogy az egyik elfogadja közülük, az összes többi kérelem törlésre kerül. Egy játékos csak egy csapat tagja lehet. A csatlakozás elfogadása vagy elutasítása a csapatvezető feladata, csakis ő láthatja ezeket a kérelmeket. A csapatvezető a csapatot létrehozó felhasználó, vagy ha már nincs a csapatban az eredeti alkotó, akkor a legrégebb óta csatlakozott felhasználó a tulajdonos.



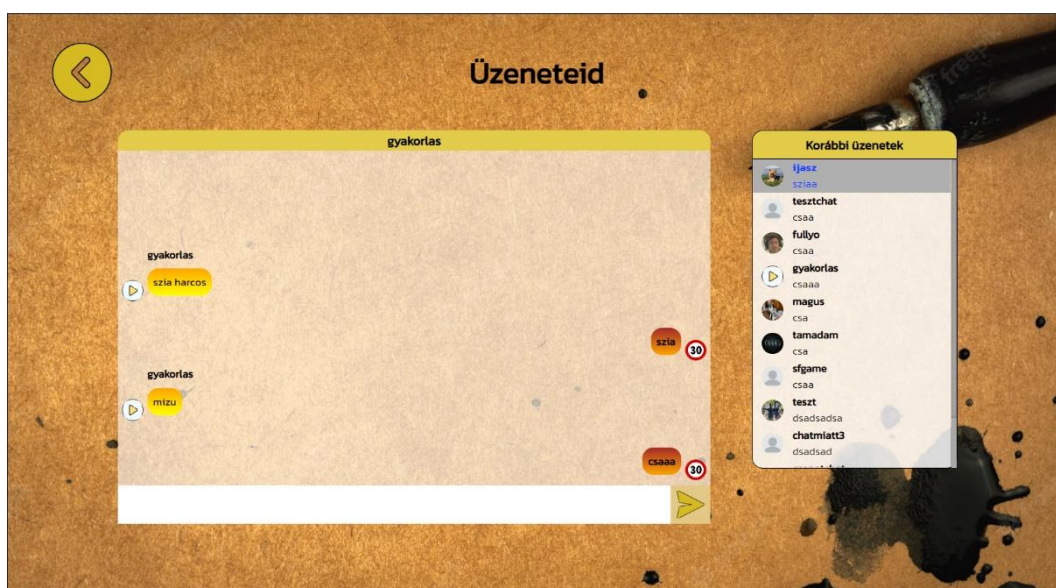
21. ábra
A csapat profilja

A vezető tud továbbá támadást indítani egy másik csapat ellen. Az előbbiekhöz szükséges ikonok csak a vezetőnél láthatóak. Támadás során ugyanaz zajlik le, mint egy aréna harcban, csak egymás után több játékosal. Játékosok eltávolítására is van lehetőség, ez szintén csak a

vezető jogosultsága. A piros háttérű szemetes ikonra kattintva távolítható el egy játékos a csapatból. A csapatnak a leírása a felhasználói profilban szereplő leíráshoz hasonlóan módosítható, szintén csak a vezető által.

A *publikus chat* a képernyő jobb oldalán elhelyezkedő, alsó ikonra kattintva érhető el. Ez a chat a szerver összes játékosának elérhető, mindenki tud beszélni mindenkivel. Az oldal alján látható, hogy hányan vannak jelenleg a publikus szobában, és ki is írja azoknak a felhasználóknak a nevét. A chat felépítése ugyanolyan, mint a csapat profilban látott chat.

A *privát chat* menüpont a publikus chat ikon felett helyezkedik el (22. ábra). Ennek a használatával bárkivel személyesen beszélgethetünk az alkalmazáson belül. Mikor először küldünk valakinek üzenetet, a felhasználó profiljánál található üzenet küldés ikonra kell kattintani. Ilyenkor létrejön egy chatszoba. A jobb oldalon az összes korábbi üzenetváltás látható. Abban az esetben, hogyha létrehozunk egy chatszobát, de nem írunk bele semmit, értesítés nem fog érkezni. Azonban a chatszoba létrejötte látható lesz mindkettő felhasználó számára.



22. ábra
Privát üzenetküldés

Az olvasatlan üzenetek sötétszürke háttérrel, és kék betűszínnel fognak megjelenni a jobb oldalon. Az egyes chatszobák tetején látható a felhasználó neve, akivel épp beszélgetünk. Erre a névre kattintva el lehet navigálni az adott felhasználó profiljára.

A *Játék kezdése* menüpont a minijátékok elindítására szolgál. Először egy játék választó oldal töltődik be, ahol három különböző típusú játékból választhatunk (23. ábra). Minden játéknak

van egy bemutató szintje, amiért nem jár arany és tapasztalati pont sem. Ezen a szinten csak a játék bemutatása a cél, hogy a felhasználó megértse, hogy mi is lesz a feladat. Amiért már jár nyeremény, az a könnyű, közepes és nehéz szintek. Minél nehezebb egy játék, annál több arany és tapasztalati pont is jár a teljesítéséért, valamint valamennyivel több idő áll rendelkezésünkre a játék teljesítéséhez. A játék kiválasztásához először rá kell kattintani a játék nevére. Ilyenkor meg fog jelenni a szintválasztó felület. Az adott szint kiválasztása után megjelenik az *Indítás* gomb, amivel el lehet indítani a játékot. A visszaszámláló egyből elindul, amint a játék betöltésre került. A játékosnak ennyi idő áll rendelkezésre a játék megoldására. Az utolsó 10 másodpercben vörösen elkezd villogni a visszaszámláló, így jelezve a játékosnak, hogy sietnie kell.



23. ábra
A játékválasztó felület



24. ábra
Az Aknakereső minijáték

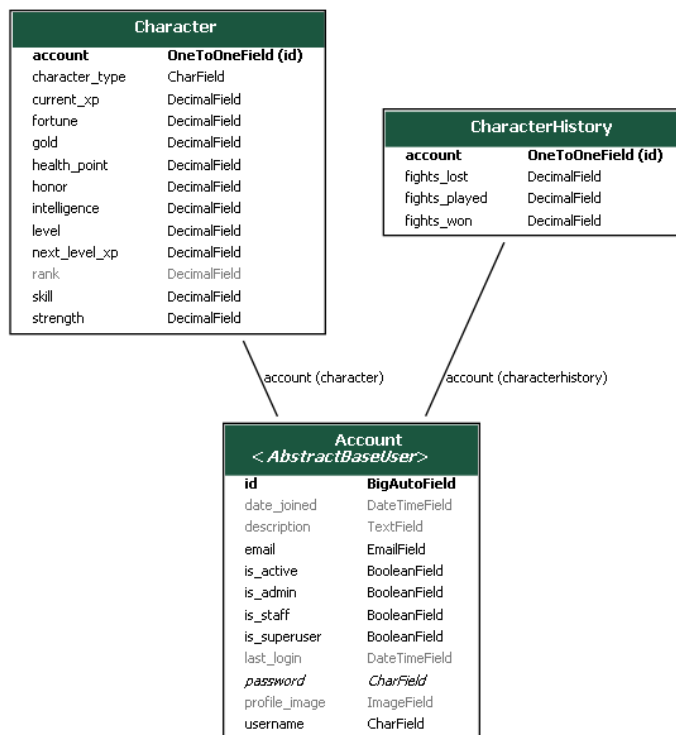
Abban az esetben, hogyha lejár az idő, semmilyen szankció nem sújtja a játékost, korlátlanul újra próbálkozhat a játék megoldásával. A szoftver ilyenkor megmutatja, mi lett volna a helyes megoldása a játéknak. Mivel minden játék véletlenszerűen generálja le a játéktérképet, sosem lesz ugyanaz a megoldás. A 24. ábrán az Aknakereső játék látható.

3.2.3. A szoftver felépítése

A következőkben a szoftver fejlesztői szempontból történő bemutatása lesz olvasható.

Az alkalmazás egy Django alapú webalkalmazás. A szerver oldalon a Django kezeli minden folyamatot. A kliens oldalon nem használtam semmilyen front-end keretrendszert, a kinézetet sima HTML, CSS és JavaScript használatával oldottam meg, illetve a Django template rendszer segítségével. A projektstruktúra kialakítása során igyekeztem követni a szabványos Django alkalmazások felépítését. Létrehoztam a statikus fájloknak egy *static* mappát a főkönyvtárban, valamint az alap HTML fájloknak egy *templates* mappát. Továbbá az összes app-ban megtalálható egy *urls.py* fájl, az app-hoz tartozó URL-ek tárolására, továbbá egy *templates* mappa, az app nézetfüggvényeihez tartozó HTML fájloknak. Azonban néhány helyen eltértem ettől a felépítéstől. Azokból az app-okból kiszedtem a *urls.py* fájlt, ahol nem volt kettőnél több URL használatban. Ezeket a fő *urls.py* állományba helyeztem el. Továbbá azoknál az appoknál, ahol nincs semmilyen template kiépítve, értelemszerűen a *templates* mappa is elhagyásra került.

Az alkalmazás hét, általam létrehozott appból áll. Ezek együttese alkotja a teljes szoftvernek a teljes funkcionalitását. Az első, az *account* app, amely a felhasználóknak a kezeléséért felel, és a magja a többi modulnak. A *core* app tartalmazza az összes olyan funkciót, amely nem illeszthető be az egyes appokba. Ez tartalmazza például a kezdőlapot, a ranglistát, valamint az appok által használt konstansoknak a kollekciónak is. Ezt követi a *public_chat* és *private_chat* app, amelyek a nevükből származó chatek működését teszik lehetővé. Ezek után következik a *team* app, amely a csapat funkcionalitást adja hozzá a programhoz, valamint tartalmazza a csapat chatet is. Miután az összes chat modul létrejött, ezeket követi a *notification* app, amely az értesítési rendszer működéséért felel. Ezeknek az appoknak a használatával már egy teljes rendszer kiépítésre került a háromfajta chatnek a működtetéséhez, az értesítési rendszerrel. A legutolsó modul a *game*, amely tartalmazza a játéknak a logikáját. Kezeli és szimulálja a játékosok közötti harcokat, számontartja az egyes játékosok statisztikáit, az arany, tapasztalat és becsületpontok jóváírásáért is felel. Továbbá ez a modul tartalmazza az egyes minijátékokat is. Tehát a szoftvernek a nagyobb részét a chat teszi ki kód szinten, maga a játék az egyetlen modulba lett elhelyezve. A következőkben ezeknek az appoknak a felépítését és működését fogom részletezni.



25. ábra
A felhasználói modell

Az *account* app került létrehozásra legelőször. Ez felel a felhasználók teljeskörű kezeléséért. Egy felhasználó objektum 3 tábla együtteséből tevődik össze (25. ábra). Fontosnak tartottam, hogy a felhasználó alapadatai, például a neve vagy az email címe egy külön táblában legyen kezelve (*Account*). Ez azért volt fontos, mivel a szoftver tervezésekor kettő nagy részre bontottam a projektet: az egyik volt a chat rész, a másik pedig maga a játék. A chat bele lett illesztve a játék struktúrájába, azonban kód szinten a játék és a chat két teljesen elkülönülő rendszer. A chathez nincs szükség a felhasználó által létrehozott karakternek az adataira. Ezzel a megvalósítással, az egyes részeit a szoftvernek könnyen újra lehet használni egy másik projekten belül. A chatmodulok könnyen kiemelhetők az alkalmazásból, és beilleszthetők egy másikba. Az *Account* tábla a fő tábla. Ez felel a felhasználók beléptetéséért és azonosításáért. A karakterhez tartozó alapinformációk a *Character* táblába lettek eltárolva. Ezen kívül létrehoztam egy *CharacterHistory* táblát is. Ebben a karakterről lehet tárolni különböző statisztikákat, amely csak a játékhoz köthető. Ezt a későbbiekben könnyen bővíteni is lehet, a másik kettő tábla módosítása nélkül. Ezek között a táblák között *OneToOne* kapcsolat áll fenn, hiszen minden *account* objektumhoz egy darab *Character* és *CharacterHistory* objektum kapcsolódik. Ezek a táblák kiterjesztései a fő táblának. Az *Account* tábla létrehozásakor az *AbstractBaseUser* osztályból örököljük annak funkcióit. Ezt a modellt használva lehetőségünk van a teljes *User* modell testreszabására. Az alaposztály tartalmazza a belépéshez szükséges azonosító funkciókat, tehát azt már nem kellett megírni, azonban semmilyen más attribútumot nem. Azokat mind én adtam meg. Ebben az esetben azt is be kell állítani, hogy a program az osztály mely attribútumát használja felhasználónévnek a beléptetés során. Ezt a `USERNAME_FIELD = attribútum neve` sorral adhatjuk meg. Az osztály elkészítése után be kellett állítani a *settings.py* fájlban, hogy ezt a modellt használja a program a beléptetéshez. Ezt a `AUTH_USER_MODEL = 'account.Account'` sorral lehetett beállítani. Az *account* utal az appnak a nevére, míg az *Account* a Python osztály neve, amellyel definiáltuk a modellt.

```

class Character(models.Model):
    account = models.OneToOneField(Account, on_delete=models.CASCADE, primary_key=True)

    ROLES = (
        ('warrior', 'warrior'),
        ('mage', 'mage'),
        ('scout', 'scout'),
    )

    character_type = models.CharField(verbose_name='character type', max_length=20, choices=ROLES, default='')

    level = models.DecimalField(verbose_name='level', max_digits=19, decimal_places=0, default=1) # szint

    # tulajdonságok
    strength = models.DecimalField(verbose_name='strength', max_digits=19, decimal_places=0, default=8) # erő
    skill = models.DecimalField(verbose_name='skill', max_digits=19, decimal_places=0, default=6) # ügyesség
    intelligence = models.DecimalField(verbose_name='intelligence', max_digits=19, decimal_places=0, default=7) # értelem
    health_point = models.DecimalField(verbose_name='health point', max_digits=19, decimal_places=0, default=10) # életerő
    fortune = models.DecimalField(verbose_name='fortune', max_digits=19, decimal_places=0, default=8) # szerencse

    rank = models.DecimalField(verbose_name='rank', max_digits=19, decimal_places=0, blank=True, null=True) # helyezés
    honor = models.DecimalField(verbose_name='honor', max_digits=19, decimal_places=0, default=1) # becsületpont
    gold = models.DecimalField(verbose_name='gold', max_digits=20, decimal_places=0, default=100) # arany
    current_xp = models.DecimalField(verbose_name='xp', max_digits=20, decimal_places=0, default=0) # jelenlegi xp pont
    next_level_xp = models.DecimalField(verbose_name='next level xp', max_digits=20, decimal_places=0, default=150) # következő szint xp pont

    objects = CharacterManager()

    def __str__(self):
        return self.account.username

```

26. ábra
A *Character* tábla felépítése

A 26. ábra mutatja be a *Character* táblának a felépítését. A *Model* osztályból örököljük a beépített funkciókat, ezáltal az osztály egy SQL táblaként fog működni, melynek az attribútumai lesznek a táblák attribútumai. Látható a *OneToOne* kapcsolat az *Account* táblához a legelső sorban. Ilyenkor meg lehet adni mi történjen abban az esetben, hogyha a *Character* objektumhoz tartozó *Account* objektum törlésre kerül. Az *on_delete* kulcsszóval adható meg ez a törlési rutin, amely az SQL szabványból ismerős lehet. Megadható az ábrán is látható *CASCADE*, ebben az esetben, ha törlődik az *Account* objektum, akkor a *Character* objektum is törlésre kerül. Ezen kívül lehet még használni a *PROTECT*, *RESTRICT*, *SET_NULL*, *SET_DEFAULT* és a *DO_NOTHING* kulcsszavakat is. A *primary_key = True* megadásával pedig megakadályozzuk, hogy duplikációk jöjjenek létre, hiszen egy felhasználónak egy karaktere lehet. Továbbá használtam még *CharField* (string típus) illetve *DecimalField* (fix pontosságú szám) típusú mezőket. Látható, hogy mindkét típusnak saját attribútumai vannak. A *CharField* esetében meg kell adni a maximális hosszát a megadható stringnek, míg a *DecimalField*-nél a számjegyeknek a maximális számát, illetve a tizedesjegyek maximális számát is. Ezek megadása nélkül hibaüzenetet kapunk. A *verbose_name* kulcsszóval adhatunk az egyes attribútumoknak nevet, amely olvashatóbbá teszi az adatbázisban szereplő attribútumokat. Ha ezt nem adjuk meg külön, a Django alapból létrehozza ezt a változónév alapján. Látható, hogy a *character_type* attribútumnál szerepel a *choices* kulcsszó. Ezzel egy

iterálható adatsorozat adható meg. Az ábrán egy *tuple* típusú sorozat látható. Mindegyik *tuple* első eleme az az érték, amely a modellben beállítható, a második pedig a programozó számára olvasható érték. Jelen esetben ez a kettő ugyanaz. Ezzel a logikával lehet meghatározni, hogy melyik felhasználónak milyen típusú karaktere van.

A regisztráció során létrejön a korábban említett 3 tábla objektum, melyek között *OneToOne* kapcsolat áll fenn. A következő néhány sorban bemutatom a regisztráció folyamatát. Mikor egy felhasználó megnyitja a regisztrációs oldalt, a 27. ábra által megjelenített nézetfüggvény fut le.

```
def register_page_view(request):
    form = AccountRegistrationForm()
    character_type = None
    if request.method == 'POST':
        form = AccountRegistrationForm(request.POST)

        if form.is_valid():
            user = form.save()
            character_type = form.cleaned_data.get('character_type')
            if character_type == Character.ROLES[0][0]: # 'warrior'
                set_warrior(user)
            elif character_type == Character.ROLES[1][0]: # 'mage'
                set_mage(user)
            elif character_type == Character.ROLES[2][0]: # 'scout'
                set_scout(user)
            else:
                pass

            update_rank()

            return redirect('login')

        else:
            character_type = form.cleaned_data.get('character_type')

    context = {
        'form': form,
        'character_type': character_type,
    }

    return render(request, 'account/register.html', context)
```

27. ábra
A regisztráció nézetfüggvénye

A függvénynek kettő visszatérési pontja van, azonban három eshetőséget kezel le. Az első, mikor a felhasználó megnyitja a weboldalt. Ebben az esetben létrejön az általunk készített regisztrációs űrlap, az *AccountRegistrationForm* és megjelenítésre kerül az oldalon (28. ábra). Ilyenkor a felhasználó meg tudja adni az adatait, és regisztrálhat az oldalra. Látható, hogy az egyes *input* és *submit* mezőkön kívül megadásra került egy *csrf_token* nevű *template tag*, amely a CSRF (Cross-Site Request Forgery) támadások ellen nyújt védelmet. Ennek a támadásnak a

során a támadó által végrehajtani kívánt parancsok hajtódnak végre a felhasználó nevében. Ezt a kódot az összes űrlap esetében használni kell, különben a Django beépített funkcionalitása le fogja tiltani a beérkező kérelmet.

```
<form method="POST" action="" id="registration-form">
  <div id="form-part1">
    {%csrf_token%}
    <div class="input-field-with-error-text">
      <input type="text" name="username" maxlength="20" autofocus required id="id_username">
      <span id="username-not-available">Foglalt vagy hibás felhasználónév</span>
    </div>
    <div class="input-field-with-error-text">
      <input type="email" name="email" maxlength="128" required id="id_email">
      <span id="email-not-available">Foglalt vagy hibás email cím</span>
    </div>
    <div class="input-field-with-error-text">
      <input type="password" name="password1" autocomplete="new-password" required id="id_password1">
      <span id="password-weak">A jelszó nem megfelelő erősségű</span>
    </div>
    <div class="input-field-with-error-text">
      <input type="password" name="password2" autocomplete="new-password" required id="id_password2">
      <span id="password-not-match">A két jelszó nem egyezik meg</span>
    </div>
    <div>
      <select name="character_type" id="id_character_type">
        <option value="warrior">warrior</option>
        <option value="mage">mage</option>
        <option value="scout">scout</option>
      </select>
    </div>
  </div>
  <div id="form-part2">
    <input type="submit" value="Regisztráció">
  </div>
</form>
```

28. ábra
A regisztrációs űrlap HTML kódja

A regisztráció esetén egy POST kérelem fog beérkezni a szerverhez, tehát le fog futni az *if* szerkezetben meghatározott funkcionalitás. Abban az esetben, hogyha az űrlap megfelelően kitöltésre került, akkor a *form.save()* segítségével a felhasználó objektum létrehozásra kerül az adatbázisban. Ezt követően, kinyerjük az űrlapból a megadott karakter típust, a *form.cleaned_data.get()* segítségével, melynek paraméterül adjuk a HTML fájlban szereplő *select*-nek a *name* attribútumának az értékét. Így tudunk hivatkozni a beállított értékére a szerver oldalon. Ezután a karaktertípusnak megfelelően beállítjuk a karakter tulajdonságainak az értékét. Végül lefut egy *update_rank()* nevű függvény, amely frissíti a játékban a ranglistát, és elhelyezi benne az új felhasználót. Ezután átirányít minket a program a bejelentkező oldalra, ahol már be tudunk majd jelentkezni. Abban az esetben, hogyha az űrlapban helytelen adat szerepel és így került elküldésre a szervernek, akkor hibaüzenetet jelenítünk meg az oldalon. Az egyes hibaüzeneteket az általunk készített *AccountRegistrationForm* generálja az adatok ellenőrzése során. Ez a *form* változóba lesz eltárolva. Ezeket a hibaüzeneteket a *form.errors*

használatával tudjuk kinyerni, ezért az egyik változó a *form*, amit átadunk a kliens oldalra. Továbbá, szintén átadjuk a karakter típusát is hiba esetén is, mivel a regisztrációs oldalon már nem szeretnénk, hogy a felhasználónak megint ki kelljen választani a karakternek a típusát. Ilyenkor az oldal már előre betölti a választását a felhasználónak, és csak az adatait kell újra megadnia.

```
class AccountRegistrationForm(UserCreationForm):
    character_type = forms.ChoiceField(choices=Character.ROLES)
    email = forms.EmailField(max_length=128, error_messages = {'invalid': 'Helytelen email cím'})

    class Meta:
        model = Account
        fields = ['username', 'email', 'password1', 'password2']
```

29. ábra
A regisztrációs űrlap definiálása

Ahogy azt láthatjuk, a regisztrációs űrlapok segítségével tudunk új felhasználókat hozzáadni az oldalunkhoz. Ezt a *forms.py* fájlban hoztam létre (29. ábra). Az általam létrehozott űrlap a *UserCreationForm* osztályból örököli meg az egyes funkciókat. Ez az osztály elvégzi az ellenőrzéseket, miszerint létezik-e a megadott felhasználónév, megfelelő-e a jelszó és email cím is. Három alap mezője van, a felhasználónév és a jelszó1, valamint jelszó2. Ezeken kívül hozzáadtam még az email címet és a karakter típust is, mint kitöltendő mező regisztráció során. Egy *Meta* osztályt létrehozva lehet megadni, hogy az űrlap melyik modellhez kapcsolódik. Az előbb említett ellenőrzéseket testre is szabhatjuk. Ezek a beépített ellenőrző függvények a *clean_attribútum_neve* formátumban írhatóak át. A program ezeket automatikusan le fogja futtatni a regisztráció pillanatában.

Regisztrációkor ahogy azt korábban említettem, a program jelzi valós időben, hogyha nem megfelelően adtunk meg egy adatot. Azt az ellenőrzést is ebben az appban végezzük el. jQuery-t és AJAX-ot használtam a kliens oldalon, és egy egyszerű nézetfüggvényt a szerver oldalon ennek a megvalósításához (30. ábra). Az ábrán a felső függvény a kliens oldalon, a JavaScript részben található, míg az alsó egy szerver oldali Python nézetfüggvény. A jQuery segítségével egy eseményfigyelő függvénnyel detektáljuk azt, hogy a felhasználó valamit begépett a bemeneti mezőre. Ilyenkor lefut az AJAX technika, és elküldi a szervernek a begépett szöveget, ami az ábrán mutatott példában a felhasználó neve (*username*).

```

register.html X
$("#id_username").on("input", function(){
    var username = $(this).val()
    console.log(username)
    $.ajax({
        url: "{% url 'account:validate_username_realtime' %}",
        data: {
            "username": username
        },
        dataType: "json",
        success: function(data){
            if(data.is_available && data.is_long_enough){
                console.log("elérhető")
                resetInputFieldError("usernameField")
            }
            else if(data.is_available && !data.is_long_enough){
                console.log("nincs megadva username")
                setInputFieldError("usernameField")
            }
            else{
                console.log("nem elérhető")
                setInputFieldError("usernameField")
            }
        },
        error: function(data) {
            console.log("hiba")
        },
    })
})

views.py X
def validate_username_realtime(request):
    username = request.GET.get('username')

    is_long_enough = False
    is_available = not Account.objects.filter(username=username).exists()
    if len(username) > 0:
        is_long_enough = True

    data = {
        'is_available': is_available,
        'is_long_enough': is_long_enough,
    }

    return JsonResponse(data)

```

30. ábra
A valós idejű adatellenőrzés folyamata

Látható, hogy a dolgozat elején részletezett *template tagek* segítségével megadtuk a megfelelő nézetfüggvényhez a URL-t. Az *account* az a *namespace*-nek adott érték a fő *urls.py* fájlban, a kettőspont után pedig magának a URL-nek a *name* attribútuma lett megadva. Ebből látszik, hogy ez a URL nem közvetlenül a fő *urls.py*-ban szerepel. Ezután beállítottuk az adatsomag típusát JSON formátumra. Ezt követően lefut az ábrán látható Python függvény. Kinyerjük a

megkapott kérelemből az adatot a *request.GET.get()* segítségével. Itt paraméterként azt kell megadni, amit a kliens oldalon kulcs értéként adtunk meg. Ezután ellenőrzést végez a program, hogy az adatbázisban létezik-e már ilyen nevű felhasználó, valamint, hogy a karakter neve megfelelő hosszúságú-e. Ezeket az adatokat visszaküldi *JsonResponse* segítségével. Ezt az adatsomagot fogja megkapni a *success* része az AJAX hívásnak. Ezután a megkapott adatnak megfelelően jelenik meg, vagy tűnik el a hibaüzenet. Ha valamilyen hiba lép fel, akkor az *error* függvényrész fog lefutni.

Az *account* modul ezeken kívül kezeli még a bejelentkezést, a kijelentkezést és a profil adatainak módosítását. Továbbá felel még karakter profilnak a megjelenítéséért és az ott található funkciók működtetéséért is. Ezek külön nem kerülnek bemutatásra, mivel nagyon hasonlóan működnek az előbb részletezett programrészekhez, űrlapok és nézetfüggvények segítségével.

A következő modul a *core*. Mint azt említettem, ebben az appban nincs túl sok funkcionalitás, inkább csak összefogja a többi appnak a közös konstansait, illetve ami egyik modulba se illett igazán, az található meg itt. Ilyen a ranglista is, ahol a csapatokat és felhasználókat lehet valós időben keresni. Ebben a programrészben létrehoztam egy úgynevezett *Pagination* rendszert. Ennek az a lényege, hogy az adatbázisból lekért adat nem egyszerre fog megjelenni, hanem szétarabolva, különböző oldalakra. Ezzel a szerver erőforrásait kíméljük. Elég csak arra gondolni, hogy ha lenne több ezer felhasználónk, enélkül egyszerre kellene betölteni mindet, ezzel lassítva a weboldalt. Így azonban csak az általunk megadott számú felhasználó töltődik be egyszerre. A 31. ábra mutatja ennek a rendszernek a működését. Egy AJAX hívással futtatjuk le az ábrán látható függvényt. A hívás során átadunk egy paramétert, amellyel jelezzük, hogy hányadik oldalt szeretnénk betölteni. A függvény lekérdezi az összes *Character* objektumot az adatbázisból, majd az eredményét átadja a *Paginator* osztálynak. Ez a *QuerySet* a *Character* objektumokat tartalmazó lista lesz. Első paraméternek meg kell adni az előbbi lekérdezés eredményét. Második paraméternek egy számot vár, hogy egy oldalra hány ilyen *Character* objektum tehető. A gyakorlatban ennél nagyobb számot is lehet írni, azonban mivel én egy kis adatbázissal dolgoztam, ezért a demonstráció kedvéért állítottam ilyen alacsonyra ezt a számot. Ezt követően a *p.num_pages* használatával megtudhatjuk, hogy mennyi az összes oldalnak a száma. Ezt a számot hasonlítjuk össze az általunk lekért oldalszámmal, hogy létezik-e, tehát kisebb vagy egyenlő-e az összes oldalszámnál. Hogyha nagyobb, akkor visszaküldünk egy *None* értéket, így a kliens oldal tudni fogja, hogy lekértük az összes felhasználót. Hogyha

létezik ez az oldalszám, először megnöveljük a kapott oldalszám értékét. Ezután az *s* változónak értékül adunk egy *Serializer*-t, aminek a feladata, hogy JSON formátumra hozza, és ezáltal elküldhetővé tegye a *Character* objektumokat a kliensnek. Az *EncodeCharacterObject* serializer-t én hoztam létre. Ez egy egyszerű Python osztály, amely a beépített *Serializer* osztályból örökli meg a funkcióit, és annak egy függvényét írja át. A lényege, hogy Python szótár formátumra alakítsuk a *Character* objektumot, és ezáltal elküldhető legyen a kliensnek JSON formátumban.

```
def load_users_pagination(request):
    try:
        accounts = Character.objects.get_all_accounts_in_ordered_list_without_admins()
        characters = Character.objects.get_all_characters_in_ordered_list_without_admins()

        data = {}
        page_number = request.GET.get('page_number')
        load_page_number = int(page_number)

        p = Paginator(characters, 8)

        print('numpages', p.num_pages)

        if load_page_number <= p.num_pages:
            load_page_number = load_page_number + 1

            s = EncodeCharacterObject()

            print(s.serialize(p.page(page_number).object_list))
            print('next')

            data['users'] = s.serialize(p.page(page_number).object_list)
        else:
            data['users'] = 'None'

        data['load_page_number'] = load_page_number

        return JsonResponse(data)

    except Exception as exception:
        print('Error when getting users' + str(exception))

    return None
```

31. ábra
A *Pagination* rendszer működése

Az *s.serialize(p.page(page_number).object_list)* sor egy összetett funkcionalitást hajt végre. Először a *p.page()* függvény fut le, amelynek paraméterül a lekérni kívánt oldalszámot adjuk. Ennek az eredménye az adott oldal lesz, amely az objektumokat tartalmazza. A *.object_list* segítségével megkapjuk ezeknek az objektumoknak a listáját. Ezután ezeket az objektumokat

átalakítjuk Python szótár formátumra az *s.serialize()* segítségével, amelyet később a küldésre alkalmas JSON formátumra lehet alakítani. Ezt a listát elmentjük egy kulcs-érték párként, majd az új oldalszámmal együtt elküldjük a kliensnek.

A keresés ezen az oldalon szintén valós időben zajlik, tehát ahogy gépeljük a keresett felhasználókat és csapatokat, egyből látjuk annak az eredményét. Ez szintén egy AJAX hívással indul, mely során átadjuk a keresett nevet, valamint azt, hogy játékost vagy csapatot keresünk-e. Ennek az oka, hogy egy függvény kezeli mind a játékosok, mind a csapatok keresését. A függvény elején egy *if-else* szerkezet, valamint az AJAX által küldött információ segítségével meghatározzuk, hogy felhasználót vagy csapatot kell keresni. Felhasználó keresése esetén az *Account.objects.filter(is_admin=False).filter(username__icontains=searched_text)* parancs végrehajtásával kilistázzuk a találatokat (32. ábra).

```
if len(searched_text) > 0:
    accounts = Account.objects.filter(is_admin=False).filter(username__icontains=searched_text)
    character_results = [Character.objects.get(account=account.id) for account in accounts]
    character_results = sorted(character_results, key=lambda character: character.rank)
    results = s.serialize(character_results)
```

32. ábra
A felhasználók keresésének kódrészlete

Az *is_admin* segít kiszűrni, hogy az admin felhasználók ne látszódnak. A *username__icontains* pedig rászűr a fiókok felhasználó nevére. Az *icontains* miatt nem számít, hogy kis vagy nagybetűt írunk a keresés során. Ezután a felhasználónevek alapján kilistázzuk a hozzájuk tartozó karaktereket a *[Character.objects.get(account=account.id) for account in accounts]* parancs segítségével, majd ezt követően a kapott listát rendezzük helyezés szerint. Így a találatok sorrendjében is érvényes lesz az, hogy az van előrébb, akinek a legtöbb becsületpontja van. Ezután ezeket a karakter objektumokat is át kell alakítani szótár formátumra, a *Pagination* rendszerrel ismertetett *s.serialize()* paranccsal. Ezt követően visszaküldésre kerülnek a kliensnek a talált objektumok. A kliens oldalon a megkapott objektumokon egy *for* ciklus segítségével végigmegy a program. JavaScriptet használva új div-ek kerülnek létrehozásra, és a kapott adatok ezekben kerülnek megjelenítésre. A létrehozott elemeket egy tároló elemhez adjuk hozzá, így a felhasználó is láthatja valós időben a találatokat. Az app ezen kívül a kezdőlap megjelenítéséért felel egy egyszerű nézetfüggvény segítségével. Az ehhez tartozó template-ben fog működni a *notification* modul WebSocket része, ez annak a modulnak a bemutatásánál részletezve lesz.

A *public_chat* az első chat modul, ez az app készült el a legelsőnek a három fajta chat közül. Az ehhez tartozó modellt mutatja be a 33. ábra. Látható, hogy két attribútuma van, az első egy egyszerű *CharField*, ami a szoba nevét fogja tartalmazni, és kötelezően meg kell adni.

```
class PublicChatRoom(models.Model):
    name = models.CharField(max_length=60, unique=True, blank=False)
    users = models.ManyToManyField(settings.AUTH_USER_MODEL, blank = True)

    def __str__(self):
        return self.name

    def add_user_to_current_users(self, user):
        if not user in self.users.all():
            self.users.add(user)
            self.save()
            print(f'{user.username} ONLINE')

    def remove_user_from_current_users(self, user):
        if user in self.users.all():
            self.users.remove(user)
            self.save()
            print(f'{user.username} OFFLINE')

    @property
    def group_name(self):
        return f'MainRoom_ID_{self.id}'
```

33. ábra
A publikus chatszoba modellje

A második attribútum egy *ManyToMany* kapcsolat, az *Account* modellhez. Ez a mező a jelenleg a chatszobában tartózkodó felhasználó objektumokat tartalmazza. Látható a modell alatt két segédfüggvény, amelyek kezelik ezt a mezőt. Az *add_user_to_current_users* a consumer-ből lesz lefuttatva, mikor egy felhasználó csatlakozik a chathez. Ebben az esetben a függvény hozzáadja a felhasználót a *users* mezőhöz. A *remove_user_from_current_users* függvény pedig ennek az ellenkezőjét csinálja, mikor egy felhasználó kilép a chatszobából. A modul nézetfüggvénye egy nagyon rövid kód, csupán a publikus chatszobának az ID-ját adja át a template-nek. A WebSocket létrehozásával indul a folyamat, ahogy azt ismertettem a WebSockets bemutatásánál. A */kozosseg/{{room_id}}/* elérési utat használom URL-ként. Itt a *{{room_id}}* a nézetfüggvénytől kapott ID, amellyel a kiválasztott szobát tudjuk megjeleníteni. A *routing.py* fájlban ennek a helye elő van készítve relációs jelek segítségével, oda fog ez az

ID átadódni. A WebSocket kapcsolat kiépítése után a kliens egy *join* csatlakozási parancsot küld a consumer-nek. A consumer-ben az ehhez a parancshoz tartozó függvény fogja kezelni az előbb részletezett *users* mezőt. Továbbá, hozzáadja a felhasználót a szobához tartozó csoporthoz. Mint azt korábban részleteztem a Django Channels bemutatásánál, a csoportok csatornák egy kollekciója, amelyekre a nevükkel tudunk hivatkozni. A modellben látható egy *@property* dekorátorral ellátott függvény. Ez a függvény adja vissza a csoportnak a nevét. Látható, hogy az adott chatszoba példány ID-ja is hozzá van fűzve a névhez, így ha több chatszoba lenne, ezzel a módszerrel meg lennének különböztetve. A *@property* dekorátor lényege, hogy a függvényre úgy tudunk hivatkozni, mintha csak egy változó lenne. A csatlakozás során a felhasználó csatornáját adjuk hozzá az adott csoporthoz. Ezt a *group_add* függvénnyel tudjuk megtenni a consumer-en belül. Meg kell adni a felhasználó csatornáját, amelyet a *self.channel_layer* paranccsal elérhetünk a consumer-ben, továbbá a csoportnak a nevét. A csoport neve a *room.group_name* paranccsal adható meg, ahol a *room* egy publikus chatszoba objektum, míg a *group_name* az előbb említett függvény, ami visszaadja az objektumhoz tartozó csoport nevét. Látható, hogy nem kellett a nyitó és csukó zárójeleket megadni a *@property* dekorátor miatt. Mikor egy felhasználó kilép egy szobából, akkor a *leave* parancs fog elküldésre kerülni, és az ehhez tartozó függvény fog lefutni a consumer-ben. Ilyenkor a felhasználót eltávolítjuk a *users* mezőből, továbbá a csoportból is. A csoportból való eltávolításhoz ugyanazokat a paramétereket szükséges megadni, mint csatlakozáskor, tehát a felhasználó csatornáját és a csoport nevét. A különbség, hogy ilyenkor a *group_discard* parancsot kell használni. Üzenetküldés során a consumer-ben található *receive_json* fogadja a kientől érkező csomagokat. Ebben az esetben a *send* parancs fog befutni a consumer-hez, amely lefuttatja az üzenet küldéséhez kapcsolódó kódot (34. ábra). Először ellenőrizzük, hogy a felhasználó valóban csatlakozott-e a publikus chatszobához, valamint az alkalmazásba is be van-e lépve. Ezeket az ellenőrzéseket ki is hagyhatnánk, mivel az alkalmazás egyéb részein is végre van hajtva ezeknek az ellenőrzése, de így még egy fokkal biztonságosabbá tehető az app. Ezután a *get_public_chat_room()* segítségével lekérjük az adatbázisból a chatszoba objektumot. A következő 3 sorban az értesítést küldjük ki azoknak a felhasználóknak, akik nincsenek a chatszobában, ez a *notification* modulnál lesz részletesebben bemutatva. Ezután a felhasználó által küldött üzenetet elmentjük az adatbázisba. Ezt követően az adott csoportnak elküldjük a beküldött üzenetet a *group_send* paranccsal. Ebben az esetben elegendő csak a csoport nevét megadni.

```

room = await get_public_chat_room(room_id)

current_users = room.users.all()
all_registered_users = Account.objects.all()
await add_or_update_unread_message(user, all_registered_users, room, current_users, message)

# üzenet mentése az adatbázisba
await save_public_chat_room_message(user, room, message)

# ha valakinek nincs profilképe, az alapértelmezett profilképet kapja
try:
    profile_image = user.profile_image.url
except Exception as e:
    profile_image = STATIC_IMAGE_PATH_IF_DEFAULT_PIC_SET

await self.channel_layer.group_send(
    room.group_name,
    {
        'type': 'chat.message', # chat_message
        'user_id': user.id,
        'username': user.username,
        'profile_image' : profile_image,
        'message': message
    }
)

```

34. ábra
Az üzenetküldésért felelős kódrészlet

Ezután a *type* kulcs-érték pár kiemelt fontosságú. Ilyenkor annak a függvénynek a nevét kell megadni, amely ezután az egyes csatornába el fogja küldeni mindenkinek az üzenetet. A *chat_message* függvény végzi ezt a feladatot, ilyenkor ponttal elválasztva, string formátumba kell megadni a *type* értékeként. Továbbá az egyes megjelenítendő információkat is meg kell adni, például a felhasználónevet és a profilkép elérési útját is. A *chat_message* függvény végzi az egyes csatornába történő üzenet küldést, tehát a felhasználókhöz ez juttatja el a beküldött üzeneteket (35. ábra). Először legeneráljuk a küldési időt, amely azt határozza meg, hogy mikor lett az adott üzenet beküldve a szobába. Majd az előző függvénytől kapott információkat felhasználva a *send_json* paranccsal visszaküldjük a kliensnek az egyes üzeneteket. Látható egy üzenettípus konstans is az elküldött adatok között. Ez a kliens oldalon használatos, ezekkel különböztetjük meg a különböző típusú üzeneteket. Amikor valaki csatlakozik vagy kilép a szobából, vagy üzenetet küld, mindegyiknek megvan a saját azonosítója. Ezek egyszerű egész számok. A kliens oldalon a WebSocket *onmessage* eseményfigyelő függvénye kapja meg a *send_json* által küldött adatsomagot.

```

async def chat_message(self, event):
    """
    Akkor fut le, mikor valaki üzenetet küld a chatbe
    Itt történik a tényleges üzenet elküldés a templatehez(a kliensnek)
    """

    print('PublicChatConsumer: chat_message from user: ' + str(event['username']))

    sending_time = create_sending_time(timezone.now())

    await self.send_json({
        'message_type': MESSAGE_TYPE_MESSAGE,
        'user_id': event['user_id'],
        'username': event['username'],
        'profile_image': event['profile_image'],
        'message': event['message'],
        'sending_time': sending_time,
    })

```

35. ábra
Az adatcsomag küldése a kliensnek

Ez a függvény először kinyeri az adatcsomagból, hogy milyen típusú üzenet érkezett a consumer-től a *data.message_type* kóddal. Látható, hogy az átadott kulcs értékre tudunk hivatkozni a hozzá tartozó érték megszerzéséhez. A sima üzenet a 0-val jelölt adatcsomag. Ebben az esetben meghívja az *appendChatMessage*, majd *createChatMessageElement* (36. ábra) nevű függvényeket.

```

// egy adott chat üzenet, amelyben minden benne van: profilkép, üzenet stb.
var newChatMessage = document.createElement("div")
newChatMessage.style.display = "flex"
newChatMessage.style.flexDirection = "column"
newChatMessage.style.width = "100%"
newChatMessage.classList.add("new-chat-message-container")
newChatMessage.setAttribute("id", "messageID-" + messageID)

// a felhasználó neve
var newChatMessageTop = document.createElement("div")
newChatMessageTop.style.flex = "1"
newChatMessageTop.style.display = "flex"
newChatMessageTop.style.flexDirection = "row"
newChatMessageTop.classList.add("new-chat-message-top")

// ... a kód többi része ...

newChatMessage.appendChild(newChatMessageTop)

```

36. ábra
A *createChatMessageElement* függvény egy része

Ezek minden egyes üzenethez legenerálnak egy HTML div típusú objektumot, amelyben elhelyezzük a felhasználó által küldött üzenetet, a profilképét és a nevét, továbbá a küldési időt is egy előre kialakított stílus szerkezetben. Az üzenetek a küldési idő óta eltelt idő szerint, növekvő sorrendben fognak megjelenni az üzeneteket tartalmazó tároló aljától kezdve. Tehát a legújabb üzenet lesz legalul. Újabb üzenet küldése esetén a régebbi üzenetek feljebb tolódnak. Itt is működik a ranglistáknál bemutatott *Pagination* rendszer. Oldalanként 10 üzenetet tölt be a program. Az új oldal betöltése felfele görgetéssel érhető el. Ahogy a legtetejére érünk, érzékelni fogja a JavaScriptben megírt függvény ezt az eseményt, és lekéri az újabb oldalnyi üzenetet a szervertől. Egy adott publikus üzenetnek is saját modellje van (37. ábra). Egy üzenetről eltároljuk, hogy ki küldte, és melyik szobába. Továbbá azt is, hogy az üzenet küldése mikor történt. Ezt a *DateTimeField* segítségével másodpercre pontosan megállapíthatjuk, valamint az üzenet tartalma is elmentésre kerül.

```
class PublicChatRoomMessage(models.Model):
    """
    Üzenet a PublicChatRoomban
    """
    user = models.ForeignKey(settings.AUTH_USER_MODEL, on_delete=models.CASCADE)
    room = models.ForeignKey(PublicChatRoom, on_delete=models.CASCADE)
    sending_time = models.DateTimeField(auto_now_add=True)
    content = models.TextField(unique=False, blank=False)

    objects = PublicChatRoomMessageManager()

    def __str__(self):
        return self.content
```

37. ábra
Egy publikus üzenet felépítése

Látható, hogy itt is beállításra került a CASCADE törlési módszer, tehát ha egy felhasználó törlésre kerül, az általa küldött üzenetek is törlődni fognak. Továbbá, ha a chatszoba törlődik, értelemszerűen a benne lévő üzenetek is törlésre kerülnek. Amiről még nem ejtettem korábban szót, az az egyes modellekhez tartozó Manager osztályok. Látható, hogy a publikus üzenet osztályhoz is tartozik egy ilyen, amelyet az *objects* attribútum megadásával tudunk az adott modellhez rendelni. Ezekben a manager osztályokban megadhatunk különböző lekérdezéseket az adatbázisból, amely az adott modellhez kapcsolódik.

A *private_chat* app hasonlóan működik a publikus chathez. A különbség, hogy a fő modellben itt nem felhasználóknak egy listája van megadva attribútumként, hanem a *user1* és *user2* attribútum, amely a két felhasználót jelzi, akik üzenetet küldenek egymásnak a szobában.

Továbbá, lényeges különbség, hogy itt a chatszobák dinamikusan, a program által kerülnek létrehozásra, valamint lekérdezésre, ha már létezik (38. ábra).

```
def create_or_get_private_chat(user1, user2):
    try:
        print(user1.username)
        print(user2.username)
        private_chat = PrivateChatRoom.objects.get(user1=user1, user2=user2)
    except PrivateChatRoom.DoesNotExist:
        try:
            private_chat = PrivateChatRoom.objects.get(user1=user2, user2=user1)
        except PrivateChatRoom.DoesNotExist:
            private_chat = PrivateChatRoom(user1=user1, user2=user2)
            private_chat.save()

    return private_chat
```

38. ábra

A privát chatszoba létrehozása vagy lekérése

Ezzel a publikus chatszobával ellentétben, nem szükséges statikusan beállítani a szoba ID-ját és azt átadni a consumer-nek a nézetfüggvényben. Az egyéb funkciók, például az üzenet küldése és megjelenítése megegyezik a publikus chatszobánál leírtakkal. Mivel itt több chatszoba érhető el, ezért az oldalon láthatóak a korábbi üzenetváltások más felhasználókkal. Ezeknek a chatszobáknak a megjelenítése az ehhez az oldalhoz tartozó nézetfüggvényben valósul meg. Először lekérjük az összes olyan szobát, amelyben az adott felhasználó benne van, majd annak a szobának a legutóbbi üzenetét és küldési idejét is. Ezzel tudjuk majd sorrendbe helyezni a beérkezett üzeneteket. Ezután ezeket az adatokat egy Python szótárba mentjük el. Mivel több szoba lehet, ezért Python szótárak egy listájába kerül ezeknek az adatcsomagoknak a behelyezése. Ezután ezeket elküldjük a kliensnek, ahol megjelenítésre kerülnek. Az adatcsomagban szerepelni fog az is, hogy az adott szobában van-e olvasatlan üzenete a felhasználónak. Ezzel fogjuk tudni azokat a csomagokat máshogyan megjeleníteni a felületen. Az új üzenetek beérkezése valós időben jelzésre kerül a felhasználónak. Ennek a megvalósításához JavaScriptet használtam. Az teljes oldal újratöltésre kerül bizonyos időközönként, azonban csak egy részét jelenítjük meg az új változatnak. Ezzel a módszerrel lehetőség van egy oldalnak csak egy részét frissíteni. Mivel a korábbi üzenetek egy teljesen különálló modulban helyezkednek el az oldalon, ezért azt a részt megjelenítve lehetőség van az ott található információk frissítésére bizonyos időközönként, az oldal tényleges újratöltése nélkül. Mikor egy felhasználó rákattint a korábbi chatszobák közül az egyikre, egy AJAX hívás történik, amely lekérdezi az adott szoba ID-ját, a másik felhasználó ID értékének a felhasználásával. Ezután a jelenlegi WebSocket kapcsolat bezárásra kerül, majd egy új nyílik

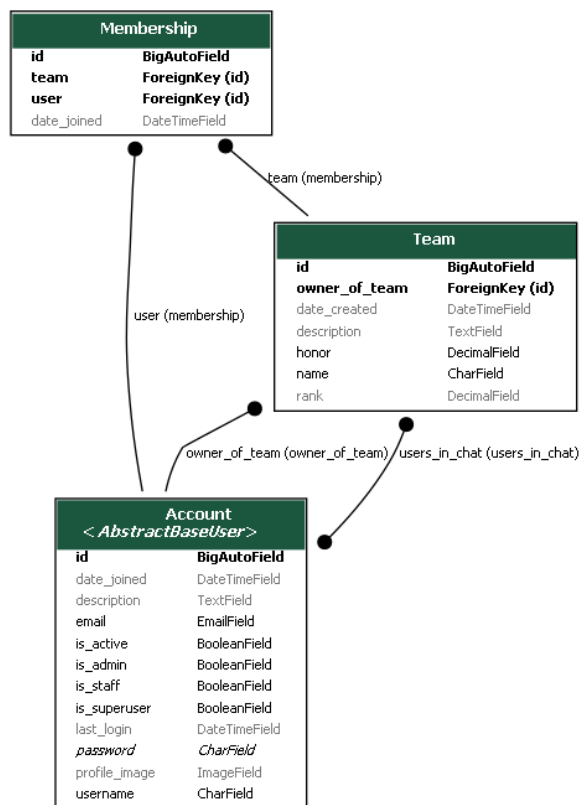
meg, már az új szoba ID-ját használva. Ezután megjelennek a megnyitott szobában található korábbi üzenetek. Mikor egy felhasználó először küld valakinek egy üzenetet, vagy csak egyszerűen a ranglistából kiválaszt egy másik felhasználót, és a profiljánál található üzenetküldés ikonra kattintva szeretné megnyitni a chatszobát, kezelni kell, hogy ilyenkor a kiválasztott felhasználó chatszobája jelenjen meg. Ezt szintén egy AJAX hívás segítségével valósítjuk meg az első lépésben. A hívás a 38. ábra által megjelenített függvényt fogja lefuttatni. Ilyenkor visszakapjuk a megtalált vagy létrehozott chatszoba objektumot. Ebből az objektumból a kliens oldalon kinyerjük a szoba ID-ját. Ezután ezt az értéket a privát chatszoba nézetfüggvényéhez tartozó URL-nek egy paramétereként átadjuk (39. ábra).

```
function onCreateOrGetPrivateChatRoomSuccess(privateChatRoomId){  
    var url = "{% url 'private_chat:private_chat_room' %}?szoba_id=" + privateChatRoomId  
    var win = window.location.replace(url)  
    win.focus()  
}
```

39. ábra
Paraméterátadás URL-en keresztül

A paramétert a rendes URL-től ?-el elválasztva adjuk meg, az ábrán látható módon. Ezt a paraméter értéket a *request.GET.get('szoba_id')* paranccsal kinyerjük a nézetfüggvényben, és ezáltal a program meg tudja jeleníteni a kiválasztott szobát a kliens oldalon.

A *team* egy összetettebb modul, hiszen a csapatok kezelése mellett a csapatchat működéséért is felel. Az *Account* tábla és a *Team* tábla kapcsolatát mutatja be a 40. ábra. Látható, hogy a csapattagsági viszony egy kapcsoló táblán keresztül (*Membership*) lett megvalósítva. Így könnyen hozzá lehet adni információkat a kapcsolathoz, például, hogy mikor csatlakozott valaki a csapathoz. Továbbá, ezzel a megvalósítással azt is meg lehetne oldani, hogy egy játékos több csapat tagja is lehessen. A játék tervezésekor azt szerettem volna, hogyha egy játékos csak egy csapat tagja lehet, viszont fejlesztés során úgy döntöttem, hogy fenntartom annak a lehetőségét, hogyha esetleg ez a későbbiekben változna. Ezen kívül a csapat tulajdonosa közvetlenül az *Account* táblához van kapcsolva egy idegen kulccsal. Továbbá látható egy *ManyToMany* kapcsolat is közvetlenül a két tábla között, ez a chatben jelenleg online lévő felhasználók jelzése miatt hoztam létre.



40. ábra

A *Team* és az *Account* modell kapcsolata

A csapatchat működése hasonló a publikus chatnél bemutatotthoz, azonban itt csak az adott csapat tagjait engedjük hozzáférni a chathez. A WebSocket kapcsolat kezdeti kiépítése mindenkinek engedélyezve van. Azonban ahogy ez létrejött, és a kientől beérkezett csatlakozási parancshoz tartozó függvény lefut, a program észleli, hogy a felhasználó nem a csapat tagja (41. ábra). Ezzel pedig a felhasználó objektum csatornája nem lesz hozzáadva a csoporthoz, és így üzenetet sem tud küldeni.

```

# ellenőrizzük, hogy a felhasználó abban a csapatban van-e amelyet épp megnyitott az oldalon
if user not in room.users.all():
    raise ClientError('ERROR', 'Ugyanabban a csapatban kell legyetek a chateléshez!')
  
```

41. ábra

Csatlakozási jogosultság ellenőrzése

Továbbá JavaScript segítségével a teljes chat felület is elrejtésre kerül. Így még ha csatlakozna is a chathez egy másik csapat tagja, maga a chat felület nem jelenne meg neki. Az app ezen kívül kezeli a csapat létrehozás függvényeit, az egyes csapatokba történő csatlakozási kérésekkel együtt. Ezek egyszerű nézetfüggvények segítségével vannak megvalósítva. A 42.

ábra mutatja be, mi történik, mikor a csapat tulajdonosa elfogad egy csatlakozási kérelmet. Látható, hogy a kérelem egy AJAX hívás segítségével jut el a szerverhez. Megkapjuk a klienstől a felhasználó ID-ját, valamint a csapatnak az ID-ját is. Ezeknek a segítségével lekérdezésre kerül az adott felhasználó és csapat objektum. Ezt követően, először ellenőrizzük, hogy az adott felhasználónak létezik-e csatlakozási kérelem objektuma. Hiszen előfordulhat, hogy időközben valaki más már elfogadta a felhasználó kérelmét egy másik csapatból. Abban az esetben minden más kérelem törlődik, így ezt az eshetőséget is kezelni kell. Ezután azt nézzük meg, hogy a felhasználó kérelmei között található-e olyan, ami a tulajdonos csapatába szól. Ha igen, először töröljük az összes csatlakozási kérelmét a felhasználónak, majd ezt követően hozzáadjuk a felhasználót a csapathoz a kapcsoló tábla segítségével. Ha bármilyen hiba lép fel, azt a kliens oldalon egy hibaüzenet formájában megjelenítjük a felhasználónak.

```
def accept_user_join(request):
    data = {}

    user_id = request.GET.get('user_id')
    user = Account.objects.get(id=user_id)

    team_id = request.GET.get('team_id')
    team = Team.objects.get(id=team_id)

    # ha van a felhasználónak függő csatlakozási kérelme
    if TeamJoinRequest.objects.filter(user=user).exists():
        # ha van a felhasználónak függő csatlakozási kérelme a csapathoz
        if TeamJoinRequest.objects.filter(user=user, team=team).exists():
            # töröljük az összes csatlakozási kérelmet ami a felhasználóhoz köthető
            TeamJoinRequest.objects.filter(user=user).delete()
            # hozzáadás a csapathoz
            Membership.objects.create(user=user, team=team)
            data['is_joined'] = 'joined'
        else:
            data['is_joined'] = 'not_joined'

    data['message'] = 'success'

    return JsonResponse(data)
```

42. ábra
Csatlakozási kérelem elfogadása

A csatlakozási kérelem elküldése és elutasítása hasonlóan működik az imént bemutatott folyamathoz. A *TeamJoinRequest* objektumok a kérelmek. Ez egy modell, amit azért hoztam létre, hogy tárolni lehessen az egyes kérelmeket az adatbázisban. Ezeknek a létrehozása történik meg, mikor egy felhasználó kérelmet küld, illetve ezek az objektumuk kerülnek törlésre, mikor a kérelmet elfogadják vagy elutasítják. Ennek a modellnek a használatával lehet kezelni a felhasználók jelentkezését egy csapatba. A kliens oldal AJAX hívásokkal kezeli ezeket a kérelmeket a felhasználóktól, és a szerver oldal ezeknek megfelelően kezeli a *TeamJoinRequest* objektumokat. A modellt a 43. ábra mutatja be. A kérelmek beérkezése modul az oldalon valós

időben frissül, beállított időközönként. Ennek a működése ugyanazzal a függvény segítségével valósul meg, amelyet a privát chatnél említettem, a korábbi chatszobák részénél.

```
class TeamJoinRequest(models.Model):
    user = models.ForeignKey(settings.AUTH_USER_MODEL, on_delete=models.CASCADE)
    team = models.ForeignKey(Team, on_delete=models.CASCADE)
    request_date = models.DateTimeField(verbose_name = 'request date', auto_now_add = True)

    def __str__(self):
        return f'Csatlakozási kérelem {self.team} csapathoz {self.user} felhasználótól'

objects = TeamJoinRequestManager()
```

43. ábra
A csatlakozási kérelem modellje

A felhasználók kirúgása vagy kilépése a csapatból szintén AJAX hívás segítségével valósul meg. Ekkor a `user_team.users.remove(user)` parancs fut le, ahol a `user_team` egy csapat objektum, a `user` pedig egy felhasználó. Ilyenkor törlődik a *Membership* objektum is, ezáltal megszűnik a kapcsolat a csapat és felhasználó között.

A *notification* modul felel az értesítések kezeléséért. A modell kialakítása során figyelembe vettem, hogy ennek a modellnek kell kiszolgáltatnia mindhárom chatnek az értesítési rendszerét. Ennek a megvalósításához a Django *ContentType* beépített modult, valamint generikus kapcsolatokat (*Generic Relation*) alkalmaztam. Az ilyen típusú kapcsolat segítségével valósítható meg, hogy egy kapcsolat nem csak egy adott táblához van kötve. Mikor a *ForeignKey* kulcsszóval csatlakoztatunk egy modellt egy másikhoz, akkor az a kapcsolat fixen a két modell között áll fent. Ebben az esetben, hogyha több modellt akarunk egy adott modellhez hozzákötni, több *ForeignKey* mezőt kell létrehozni. A generikus kapcsolatok segítségével nincs szükség az egyes *ForeignKey* mezők létrehozására. Ezáltal egyszerűbbé tudjuk tenni a modell felépítését, és kevesebbet kell kódolni is. Az idegen kulcs kiváltásra kerül három mezővel, amelyet együttesen generikus idegen kulcsnak nevezünk. Az első a *content_type* mező, amely egy *ForeignKey* mező, és a *ContentType* modellre mutat. Ez a mező felelős azért, hogy a későbbiekben az egyes táblákra mutasson, amelyet hozzákötünk a modellhez. A második mező az *object_id*, amely az elsődleges kulcsát tartalmazza annak a táblának, amire a *content_type* mező mutat. Ennek a típusa *PositiveIntegerField*, hiszen legtöbb esetben a modellekben az elsődleges kulcs az ID, amely egy pozitív egész szám. A *content_object* mező a harmadik, melynek értékekül a *GenericForeignKey()* mezőt kell adni. Ennek paraméterül kell adni az előbb említett két mezőt, azonban csak akkor, hogyha eltérő

nevet adtunk azoknak a mezőknek. Jelen esetben a szabványos elnevezés lett használva, a Django alapból ilyen nevű objektumokat fog keresni, ezért nem szükséges megadni paraméterként. A modell felépítését a gyakorlatban a 44. ábra mutatja be.

```
class Notification(models.Model):
    # felhasználó akinek küldjük az értesítést
    notified_user = models.ForeignKey(settings.AUTH_USER_MODEL, on_delete=models.CASCADE)

    # felhasználó akitől jön az értesítés
    sender_user = models.ForeignKey(settings.AUTH_USER_MODEL, on_delete=models.CASCADE, null=True, blank=True,
                                    related_name='sender_user')

    # az értesítés tartalma
    notification_text = models.CharField(max_length=100, unique=False, blank=True, null=True)

    # az értesítés keletkezési ideje
    sending_time = models.DateTimeField(auto_now_add=True)

    # számon tartja melyek azok az értesítések amelyeket a felhasználó már látott vagy nem
    read = models.BooleanField(default=False)

    # generikus idegen kulcs
    content_type = models.ForeignKey(ContentType, on_delete=models.CASCADE)
    object_id = models.PositiveIntegerField()
    content_object = GenericForeignKey()

    def __str__(self):
        return self.notification_text
```

44. ábra
Az értesítési modell

Ezzel bármelyik modellhez kapcsolhatjuk a táblát, az egyes *ForeignKey* mezők definiálása nélkül. Az egyes chat modulokban találhatók meg az ehhez a modellhez kapcsolódó modellek. Minden chat modellhez különálló modell van, ami az adott chatben lévő olvasatlan üzenetekről tartalmaz információt (45. ábra). Ez a modell objektum létrehozásra kerül minden felhasználó számára. Ezekből az objektumokból felhasználónként egy darab van, minden szobához külön-külön.

```
class UnreadPublicChatRoomMessages(models.Model):
    """
    Eltároljuk az olvasatlan üzeneteknek a számát adott felhasználónak az adott privát szobában
    Ahogy belépett a szobába a felhasználó, visszaállítjuk 0-ra a számlálót, 'olvasottra' állítjuk
    """
    user = models.ForeignKey(settings.AUTH_USER_MODEL, on_delete=models.CASCADE)
    room = models.ForeignKey(PublicChatRoom, on_delete=models.CASCADE)
    unread_messages_count = models.IntegerField(default=0)
    recent_message = models.CharField(max_length=200, blank=True, null=True)
    last_seen_time = models.DateTimeField(null=True) # a legutóbbi idő mikor a user olvasta az üzeneteket

    notifications = GenericRelation(Notification)

    def __str__(self):
        return f'{self.user.username}'\ olvasatlan üzenetei a publikus chatszobában'
```

45. ábra
Az olvasatlan üzenetek felépítése

Látható, hogy a *Notification* modell a *GenericRelation* mező használatával van a modellhez hozzákapcsolva. Ezek a modellek akkor sem törölődnek, hogyha a felhasználónak már nincs több értesítése, csak az olvasatlan üzenetek attribútuma változik meg. Az értesítések maguk a *Notification* modell példányai, azok kerülnek törlésre, amikor a felhasználó elolvassa az olvasatlan üzeneteket. Az üzenetek elolvasásának a jelzése a csatlakozási parancs során történik meg a consumer-ben. A *reset_notification_count* függvény kerül meghívásra, ahol lekérdezzük az adott szobához és felhasználóhoz tartozó *UnreadPublicChatRoomMessages* objektumot. (az ábrán ez a modell látható, ezért ezt használom példának) Beállítjuk az olvasatlan üzenetek számát 0-ra, illetve az időt, ami jelzi, hogy mikor látta az üzenetet. Ezt követően elmentjük a módosított objektumot. Abban az esetben, hogyha a felhasználó először lép be egy chatszobába, még nem lesz ilyen objektum a nevéhez párosítva. Ilyenkor egy *DoesNotExist* kivétel fog létrejönni. Ezt a kivételt kezeljük, és létrehozuk az objektumot. Mikor egy felhasználó üzenetet kap, a consumer-ben ellenőrzésre kerül, hogy az adott chatbe csatlakozva van-e jelenleg az, akinek szeretnénk küldeni az értesítést. Csak akkor frissül *UnreadPublicChatRoomMessages* objektum, hogyha a felhasználó nincs a chatben az üzenet küldésekor. Ilyenkor lefut az *add_or_update_unread_message*, amely során az olvasatlan üzenetek számát megnöveljük, valamint a legújabb üzenet mezőt módosítjuk, és értékül adjuk a kapott üzenetet az attribútumnak. Ezt követően elmentjük az objektumot. Az ilyen típusú objektumok mentése esetén egy úgynevezett *pre_save* metódus fut le, mielőtt az objektum ténylegesen mentésre kerülne. Ennek a során jönnek létre maguk a *Notification* modellnek a példányai, és ebben a metódusban kapcsoljuk össze a *Notification* modellt és azt, amelyik generálta az új értesítést. Jelen esetben az *UnreadPublicChatRoomMessages* modell lesz ez. Mikor a *reset_notification_count* függvény fut le, akkor is egy ilyen *pre_save* metódus kerül végrehajtásra, amely törli az értesítéseket az adott felhasználó-szoba páros esetében. Az előbbi leírás során a publikus chatszoba értesítési rendszerét használtam példának, viszont a másik kettő chatnél is teljesen ugyanígy működik, csak az elnevezésük más. Az értesítések jelzése a kliens oldalon hasonlóan Django Channels-t használva, consumer-en keresztül történik meg. Mint azt már korábban említettem, a kezdőlapon működik az értesítések megjelenítése. A WebSocket kapcsolat hasonlóan épül ki, mint ahogy az korábban már bemutatásra került. Az értesítési rendszer consumer-e három parancsot kezel: a három chathez tartozó értesítések számának lekérését. Ez három különálló függvény, amely lekérdezi az adott típusú chathez tartozó értesítéseket az adatbázisból, majd ezeknek a számát küldi vissza a kliens oldalra. A

kliens oldalról bizonyos időközönként érkezik egy kérelem, hogy küldje el neki az új értesítések számát. Így valós időben tudni fogjuk, ha valaki üzenet küldött az egyik chaten. A 46. ábra ábrázolja a *Notification* modellhez tartozó consumer egy részét, amely a beérkező parancsokat kezeli.

```
try:
    if command == 'get_unread_private_chat_room_messages_count':
        try:
            info_packet = await get_unread_private_chat_room_messages_count(user)
            if info_packet != None:
                info_packet = json.loads(info_packet)
                await self.send_private_chat_notifications_count(info_packet['count'])
        except Exception as e:
            print('Exception at get_unread_private_chat_room_messages_count consumer: ' + str(e))
            pass
    elif command == 'get_unread_public_chat_room_messages_count':
        try:
            info_packet = await get_unread_public_chat_room_messages_count(user)
            if info_packet != None:
                info_packet = json.loads(info_packet)
                await self.send_public_chat_notifications_count(info_packet['count'])
        except Exception as e:
            print('Exception at get_unread_public_chat_room_messages_count consumer: ' + str(e))
            pass
    elif command == 'get_unread_team_messages_count':
        try:
            info_packet = await get_unread_team_messages_count(user)
            if info_packet != None:
                info_packet = json.loads(info_packet)
                await self.send_team_chat_notifications_count(info_packet['count'])
        except Exception as e:
            print('Exception at get_unread_team_messages_count consumer: ' + str(e))
            pass
except:
    pass
```

46. ábra
A *NotificationConsumer* parancskezelő része

A *game* modul tartalmazza az összes olyan logikát, amely a játékhoz szükséges. Az egyes minijátékok JavaScriptben írt programok, azoknak a működése teljes mértékben a kliens oldalon zajlik. A kód a játék elején és végén lép kapcsolatba a szerverrel. Az elején a kiválasztott nehézségi szint alapján elküldi a kliensnek a pálya méretét, valamint a rendelkezésre álló időt. A játék végén pedig csakis győzelem esetén küldünk adatot a szervernek, hogy a felhasználó számára jóváírja a megszerzett aranyat és tapasztalati pontokat. Sikertelen teljesítés esetén nincs mit jóváírni, valamint az adatbázisba sem kerülnek elmentésre az egyes próbálkozások sem. Emiatt felesleges lenne az adatküldés. A 47. ábra egy példa arra, hogyan állítja be a program a játéknak a nehézségét. A *request.GET.get('jatek_tabla_meret')* segítségével a linkben átadott paraméterhez fér hozzá a program. Ezt a *?jatek_tabla_meret=4* formátumban adhatjuk át, ahol a szám a pálya méretét jelenti. A URL megnyitásakor ezt a

program automatikusan kezeli. A kiválasztott nehézségi szint alapján állítja be a számot, amely alapján a játéktábla legenerálásra kerül. Számon tartja azt is, mely számok lehetnek a megfelelő értékek, így ha a felhasználó át is írja a URL-t valami másra, nem fog a program hibára futni. Ebben az esetben egy beállított alapértelmezett értéket fog betölteni a program.

```
game_field_size = request.GET.get('jatek_tabela_meret')
game_field_size = int(game_field_size)

if game_field_size == 4:
    minutes = 0
    seconds = 45
elif game_field_size == 5:
    minutes = 1
    seconds = 10
elif game_field_size == 6:
    minutes = 2
    seconds = 10
elif game_field_size == 2:
    minutes = 0
    seconds = 10
else:
    minutes = 0
    seconds = 45

user = Account.objects.get(id=request.user.id)

context = {
    'user_id': user.id,
    'game_field_size': game_field_size,
    'minutes': minutes,
    'seconds': seconds,
}

return render(request, 'game/easy_game.html', context)
```

47. ábra

A játék inicializálása a szerver oldalon

A kliens oldalnak átadásra kerül a játéktábla mérete, valamint az adott mérethez a rendelkezésre álló idő. Ezeket JavaScript és Django *template tagek* segítségével állítjuk be.

A minijátékokon kívül az aréna harcokat is ez a modul kezeli. Ehhez egy külön modellt is létrehoztam, hiszen az egyes karaktereknek a profiljánál megjelenik statisztikaként a lejátszott aréna harcok száma. Ehhez pedig ezeket el kell tárolni az adatbázisban. A 48. ábra mutatja be az egyes aréna harcok felépítését. Két játékos van, aki megküzd egymással, továbbá eltárolásra kerülnek az egyes életerő értékek is. Ezeknek a szimuláció során van értelme, hiszen általában nem egy ütéssel győzik le az ellenfelet az adott játékosok. Így a köztes életerőket is tárolni kell. Ezen kívül a harc győztese, valamint a harc dátuma is tárolva van. A harc szimulációja egy

nézetfüggvényben történik a szerver oldalon. Ilyenkor meghatározásra kerül a karakternek a típusa, és a típus alapján az ahhoz tartozó fő tulajdonság értéke is.

```
class Arena(models.Model):
    attacker = models.ForeignKey(Character, on_delete=models.CASCADE, related_name='attacker')
    defender = models.ForeignKey(Character, on_delete=models.CASCADE, related_name='defender')

    attacker_health_values = models.TextField(null=True)
    defender_health_values = models.TextField(null=True)

    winner_of_fight = models.ForeignKey(Character, on_delete=models.CASCADE, related_name='winner')
    date_of_fight = models.DateTimeField(verbose_name='date of fight', auto_now_add = True)

    objects = ArenaManager()

    def __str__(self):
        return f'Arena fight between {self.attacker} and {self.defender}'
```

48. ábra
Az Aréna modellje

Ilyenkor kerül meghatározásra az ellenfél védekező attribútuma is. Ezeknek az értékeknek a figyelembevételével kiszámításra kerül egy sebzés érték, amely a teljes életerő pontból vonódik le. Ez a levonás addig zajlik, amíg az egyik karakter életerője 0-ra nem csökken. Ilyenkor a harc befejeződik, és az adatbázisba eltárolásra kerül. Ezután történik az adatok átadása a kliens oldalra, és a szimuláció lejátéssza JavaScript segítségével. Tehát a harc az már az indítás pillanatában eldől, így ha a felhasználó ki is lép az adott szimulációból, a harcot nem tudja nem megtörténté tenni. A csapat aréna hasonlóan működik, mint a rendes aréna. Azonban ezek a meccsek nem kerülnek eltárolásra az adatbázisban. Ilyenkor a csapat tagjai két listába lesznek rendezve, szintjük szerint növekvő sorrendben. Így mindig a két legalacsonyabb szintű játékos csap össze. A játékosok öröklük az életerőjüket a következő harcra. Tehát ha valaki legyőzi az ellenfelét, de az életerőjének a fele odalett, a következő harcban nem fog újratöltődni, így már sérülten kell kiálljon a következő karakter ellen. Ez két lista segítségével valósul meg. Addig tart a harc, amíg az egyik lista végére nem ér a program. Olyankor az a vesztes csapat, akinek már nem maradt több játékosa, tehát amelyik lista hamarabb kimerül. A két játékos közötti harc eldöntése ugyanazzal a logikával történik, mint a normál arénában. A különbség csak annyi, hogy nem egyszer hívjuk meg azt a függvényt, amely eldönti ki a győztes, hanem egy *while* ciklusban folyamatosan, amíg a lista végére nem érünk. JavaScriptben a jQuery animációk használatával alakítottam ki a harcnak a szimulációját, hangeffektekkel együtt. A csapatharc megjelenítése egy külön oldalon történik meg. Ehhez a *Django Sessions* technológiát alkalmaztam. Mivel egy teljesen új oldalon történik a csapatharcnak a szimulációja, ezért az

adatokat valahogyan át kell adni az egyik nézetfüggvényből a másikba. Ezt teszi lehetővé a *sessions* technológia. A `request.session['attacker_team_id'] = attacker_team.id` paranccsal kód szinten egy Python szótárba el lehet menteni bizonyos adatokat, amelyek aztán a teljes weboldalon bárhol elérhetőek. Ezeknek a visszaállítása a *None* értékkel történhet meg. A programban a két csapatnak az ID-ját mentem el ilyen módszerrel, és a csapatharc nézetfüggvényében ezeknek az ID-k felhasználásával kérem le végül a rendes csapatobjektumot.

Ahogy korábban említettem, az alkalmazásban a front-end kezeléséhez nem használok semmilyen keretrendszert. A felület kialakításánál a legtöbb esetben a CSS Flexible Box Module, röviden *Flexbox* technológiát alkalmaztam. Ezzel könnyen dinamikussá lehet tenni a felületet, és százalékos arányban megadni az egyes tároló elemek méretét, valamint elhelyezkedését is. Az egyes oldalak tartalmazznak egy *header* és egy *footer* részt. A *header*-ben általában az adott oldal címe található, valamint egy gomb, amivel vissza lehet térni a főoldalra. A *footer*-ben egyéb információk, esetleg néhány gomb található, amely az adott oldalhoz kapcsolódik. E kettő között helyezkedik el az oldalnak a tartalma. A 49. ábra egy kódrészletet mutat be, ahol ezt a technológiát alkalmaztam.

```
#team-header{
    height: 20%;
    display: flex;
    align-items: center;
    justify-content: center;
}

#team-body{
    height: 60%;
    display: flex;
    flex-direction: row;
    align-items: center;
    justify-content: center;
}

#team-footer{
    height: 20%;
    display: flex;
    align-items: center;
    justify-content: center;
}
```

49. ábra
Példa a *flexbox* használatára

Egy tároló elemben helyezkednek el az ábrán látható részei az oldalnak. Először megadtam százalékos arányban, hogy a szülő tárolóhoz viszonyítva hány százalékot foglalhatnak el az egyes részek az oldalon, magasság szempontjából. Ezután megadtam, hogy flexboxot szeretnék használni a további elemek rendezésére, a *display: flex;* sorral. Az *align-items*, valamint a *justify-content* kódrészletek a tárolóban lévő elemek középre igazítására szolgálnak, vertikálisan és horizontálisan is. Továbbá megadtam, hogy a *team-body* elemben elhelyezkedő elemek oszlopokat alkossanak. Itt becsapós lehet az elnevezés. Ha sorokat akarunk, tehát hogy az egyes elemek egymás alatt helyezkedjenek el, akkor a *column* kulcsszót kell megadni a *flex-direction* értékeként. Azonban ha azt akarjuk, hogy egymás mellett, oszlopokban legyenek az egyes tárolók, akkor a *row* kulcsszót kell használni.

Ezen kívül használtam még a CSS Grid Layout Module-t, amely különböző rácsszerkezetek megvalósítására szolgál. A kezdőlapon található csempés felület ennek a technológiának a használatával lett létrehozva. Hasonlóan működik a flexboxhoz, azonban ilyenkor a *display* értékeként a *grid* kulcsszót kell megadni, majd beállítani az egyes attribútumait a rácsoknak.

3.2.4. Továbbfejlesztési lehetőségek

Az alkalmazásban rendkívül sok továbbfejlesztési lehetőség rejlik. A minijátékok sikertelen megoldása során például egyszerűen csak láthatóvá válik a teljes játékmező a felhasználó számára. Ezeket meg lehetne valósítani úgy, hogy egy mesterséges intelligencia oldja meg helyettünk ezeket a játékokat. A mesterséges intelligencia tudománya napjainkban nagyon népszerű és felkapott. Különösen elterjedt az általam is megvalósított játéktípusok esetében, épp ezért egy remek ötlet lehet ennek az implementálása az alkalmazáson belül, vagy akár egy teljesen új projekt keretében.

Továbbá, jelenleg a játékosok csak szimuláción keresztül tudnak megküzdeni egymással. Lehetővé lehetne azt is tenni, hogy ténylegesen egymással tudjanak játszani. Például az egyik minijátékot át lehetne alakítani, hogy azt ne időre, hanem egymás ellen tudják lejátszani a felhasználók. Ezt egyébként a Django Channels lehetővé is teszi, tehát már nem is egy teljesen ismeretlen technológiával kellene szembenéznünk.

Ezen kívül a játék modellen is lehetne faragni, hogy még kiszámíthatatlanabbak, és ezáltal izgalmasabbak legyenek az egymás ellen vívott szimulációk. Különböző matematikai modelleket lehetne kitalálni erre, illetve még jobban lehetne randomizálni is.

Természetesen az új ötleteken kívül egy webalkalmazást karbantartani, és üzemeltetni kell. Bizonyos programrészek hatékonyabbá tételével a program méginkább gyorsítható, és ezáltal a felhasználói élményt is lehetne javítani. Továbbá a Django-nak és a Django Channels-nek is folyamatosan jönnek az újabb és újabb kiadásai, ezekre érdemes lehet idővel átváltani. Mint azt korábban említettem, a jelenlegi verzió egy LTS kiadás. Ez még sok ideig támogatva lesz, azonban el fog jönni az az idő, mikor egy újabb verzióra kell váltani.

4. ÖSSZEFOGLALÁS

A szakdolgozat célja egy webes szerepjáték elkészítése volt, amely két általam fontosnak tartott felhasználásra alkalmas. Egyrészt, hogy tartalmazzon különböző minijátékokat, amelyek alkalmasak a napjainkban keletkezett néhány szabad percrek a hasznos eltöltésére. Ezzel egy kicsit meg is különböztetve a szokásos szerepjátékoktól ezt az alkalmazást. Továbbá, az alkalmazást használva lehessen más felhasználókkal beszélgetni az alkalmazáson belül, ezzel segítve a kapcsolatteremtést kikapcsolódás közben. Ennek a két fő célnak a megvalósítása sikeresen megtörtént. További céljaim közé tartozott a Django keretrendszer még jobb elsajátítása egy nagyobb projekten keresztül, valamint a Django Channels technológia megértése és megtanulása is. Úgy érzem, ez sikeresen megtörtént.

A szakdolgozat elején bemutatásra kerültek az alkalmazás elkészítéséhez használt technológiák. Ezeknek a technológiáknak a felhasználásával bármikor létre lehet hozni egy éles weboldalt, bármilyen témában. Tartalmazzák az összes olyan funkciót, amelyek lehetővé teszik, hogy a mai elvárásoknak megfelelő weboldalt készítsünk el. Kezdve a bejelentkezés folyamatának a kezelésétől, az adatbázis kezelésén át, a kliens oldalon az erőforrások megjelenítéséig. Ezt követően részleteztem a futtatáshoz szükséges követelményeket. Ezután bemutatásra került az alkalmazás működése. Először felhasználói szemszögből ismertettem az egyes részeit a weboldalnak, hogy mi mire jó és hogyan lehet az egyes funkciókat használni a programban. Miután letisztáztuk, hogy az alkalmazás mire képes, következhetett a fejlesztői szemszögből történő ismertetése a programnak. Ebben a fejezetrészen ismertette lett az alkalmazás működése kód szinten, továbbá az egyes modellek felépítése és kapcsolata egymással.

Az alkalmazás fejlesztése során még jobban sikerült elmélyíteni a tudásomat a Python programozási nyelvben. A Django keretrendszerrel további hasznos információkat tudtam meg, és sikerült a már meglévő tudásomat még tovább bővíteni. Az alkalmazás fejlesztésének a kezdetekor már láttam, hogy mennyi mindenre képes a Django, milyen nagy eszközkészlet áll a rendelkezésemre. Ezek közül sokat sikerült az alkalmazásba is beépíteni. Megtanultam a Django Channels használatát, amely alkalmassá tette a weboldalt a chat működtetésére. Mivel ez egy elég új technológia, és még nem is annyira ismert, nagyon hasznos lehet a jövőben, hogy már ismerem ennek a technológiának az alapjait. Hasznos tapasztalatokat szereztem a WebSocket kapcsolatok megvalósításának a folyamatáról is. A JavaScript tudásom is

jelentősen fejlődött. Mikor elkezdtem a projektet nem gondoltam, hogy ennyi mindenre képes ez a nyelv. A HTML és CSS tudásom is felfrissült. Habár ezeknek a nyelveknek a használatát már ismertem korábbról elég jól, mégis sikerült újabb dolgokat megtanulnom.

Úgy érzem, hogy a projekt elkészítése során olyan hasznos tudásra tettem szert, amelyek a későbbiekben segíteni fognak, hogy a webfejlesztés területén helyezkedjek el.

5. IRODALOMJEGYZÉK

5.1. Irodalmi források

- [1] Dr. Andrey Bulezyuk - Django 3: For Beginners, önállóan kiadott, 2021, ISBN 9798571462471.
- [2] Adrian Holovaty, Jacob Kaplan-Moss - The Definitive Guide to Django: Web Development Done Right, Apress kiadó, 2. kiadás, 2009, ISBN 978-1-4302-1937-8.
- [3] Greg Lim, Daniel Correa - Beginning Django 3: Build Full Stack Python Web Applications, önállóan kiadott, 2021, ISBN 979-8768875657.
- [4] Federico Marani - Practical Django 2 and Channels 2: Building Projects and Applications with Real-Time Capabilities, Apress kiadó, 2019, ISBN 978-1-4842-4099-1.
- [5] Jonathan Hayward - Django JavaScript Integration: AJAX and jQuery: Develop AJAX applications using Django and jQuery, Packt Publishing kiadó, 2011, ISBN 978-1-849510-34-9.
- [6] Jon Duckett - JavaScript and jQuery: Interactive Front-End Web Development, Wiley kiadó, 2014, ISBN 978-1-118-53164-8.
- [7] Jon Duckett - HTML & CSS: Design and Build Websites, Wiley kiadó, 2011, ISBN 978-1-118-00818-8.
- [8] Sufyan bin Uzayr - Mastering Python for Web: A Beginner's Guide, CRC Press kiadó, 2022, ISBN 9781032135656.

5.2. Internetes források

- [1] <https://www.djangoproject.com/start/>
- [2] <https://docs.djangoproject.com/en/3.2/>
- [3] <https://www.geeksforgeeks.org/django-project-mvt-structure/>
- [4] <https://medium.com/@ksarthak4ever/django-request-response-cycle-2626e9e8606e>
- [5] <https://www.tutorialspoint.com/what-is-middleware-in-django>
- [6] <https://docs.python.org/3/tutorial/datastructures.html>
- [7] https://www.w3schools.com/python/python_json.asp

- [8] https://www.w3schools.com/python/python_dictionaries.asp
- [9] https://www.w3schools.com/js/js_json.asp
- [10] https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API
- [11] <https://javascript.info/websocket>
- [12] <https://channels.readthedocs.io/en/stable/index.html>
- [13] <https://testdriven.io/blog/django-channels/>
- [14] <https://blog.logrocket.com/django-channels-and-websockets/>
- [15] <https://api.jquery.com/animate/>
- [16] <https://api.jquery.com/jquery.ajax/>
- [17] <https://css-tricks.com/snippets/css/a-guide-to-flexbox/>
- [18] <https://css-tricks.com/snippets/css/complete-guide-grid/>

6. KÖSZÖNETNYILVÁNÍTÁS

Ezúton szeretném megköszönni mindazoknak, akik segítettek a szakdolgozatom elkészítésében:

Köszönettel tartozom a témavezetőmnek, Dr. Szathmáry László tanár úrnak, aki az egyetemi tanulmányaim alatt bevezetett a Python programozás alapjaiba, majd a szakdolgozat készítése során hasznos tanácsokkal és rengeteg segítséggel látott el.

Köszönettel tartozom még a Debreceni Egyetem Informatikai Karának, hogy megszerezhettem az itt eltöltött félévek során a szükséges alapokat, amelyek nagy hasznomra voltak a dolgozat elkészítése során, és lehetnek is a jövőben.

Végül, szeretném megköszönni a családomnak és a barátaimnak, akiktől mindenfajta támogatást megkaptam az egyetemi éveim alatt.