

# AstroWheel

Pannonhalmi Főapátság

Szegedi SOB Technikuma



**Szakképesítés:**

Szoftverfejlesztő és -tesztelő  
- (506131203)

**Témavezető:** Rédai Dávid

**Készítők:**

Imre Zsófia  
Pappné Balogh Bernadett Katalin  
Réperger Patrícia

Szeged, 2025



# Tartalom

## Tartalomjegyzék

Tartalom.....	3
Projekt bemutatása.....	5
Projekt részletes bemutatása .....	5
Asztali alkalmazás .....	5
Adattáblák Az adatok tárolásában a következő modellek/entitások segítenek: .....	6
Weboldal .....	9
Projekt feladatainak megosztása .....	10
Projekt ütemezés és alkalmazott módszerek.....	11
Projekt tervezése .....	11
Ötletek szűkítése, konkretizálása.....	11
Unity tanulmányozása, 2D játék, szerkezet kitalálása.....	11
Relációs táblák készítése .....	11
Stílus, vizuális megjelenés megbeszélése .....	12
Elemek létrehozása Unity-ben.....	12
Adattáblák.....	12
API készítés .....	12
Egyéni munka, hibajavítás .....	12
StateMachine alkalmazása .....	12
Refaktorálás .....	13
Hearth - inventory .....	13
Játékokban gyűjtött anyagokat.....	13
Tesztelés .....	13
Nyomonkövetés: .....	14
Projekt részeinek technikai kifejtése és a létrehozásukra alkalmazott eszközök.....	14
Asztali játék .....	14
Regisztráció, bejelentkezés.....	15
Helyi adatok tárolása.....	16
Játékmenet és mechanikák .....	16
A szigetek és a játékmenet struktúrája.....	17
Fontos Unity-metódusok .....	17
Craft rendszer.....	18
Endgame .....	18
Weboldal .....	19

Az oldal struktúrája és funkcionálitása .....	20
Folder struktúra .....	20
Bejelentkezés: .....	22
Home oldal.....	24
Felhasználói felületek leírása .....	36
Asztali játék, bejelentkezés .....	36
Asztali játék Főmenü .....	37
Asztali játék, szigetek .....	38
Asztali játék, inventory.....	39
Kirakós játékok.....	40
1. Nyilas szigete - Puzzle.....	40
2. Bak szigete - Match3.....	41
6. Bika szigete – Hidden Object Game.....	41
7. Ikkrek szigete – Memória játék .....	41
Adatbázis részletes bemutatása .....	42
Kapcsolatok:.....	44
Asszociációk:.....	44
A projektben alkalmazott DTO-k:.....	46
Funkcionálitás/CRUD műveletek:.....	46
Design bemutatása.....	53
Projekt minőségbemutatása, tesztelés leírása .....	56
Asztali alkalmazás - State machine működése: .....	56
Backend:.....	57
Frontend:.....	61
Tesztelési eljárások.....	61
Tesztelési stratégiák és módszerek.....	61
Debug-olás folyamata.....	62
SWOT .....	63
Erősségek (Strengths).....	63
Gyengeségek (Weaknesses) .....	64
Lehetőségek (Opportunities) .....	64
Veszélyek (Threats) .....	64
Összegzés .....	65
Források .....	66

# Projekt bemutatása

A projekt megbeszélésének első alkalmával gyorsan kirajzolódott, hogy mindenkorban szívesen kipróbalnánk magunkat a játékfejlesztés világában. Egyhangú döntés született arról, hogy egy olyan egyszerűbb, kirakós játék gyűjteményt készítünk, amely a számunkra is érdekes és kikapcsolódást nyújtó élményt jelentene. A több kisebb játék biztosította, a projekt moduláris jellegét.

Ezen kisebb játékok rendszerezéséhez egy közös asztrológiai, misztikus tematikát választottunk.

A projekthez készítettünk egy weboldalt, ahol rangsor listákat és a játékos karakterekhez tartozó ismereteket találhatunk.

Mind a játék, mind pedig a weboldal egy közös relációs adatbázisból kapja a játékosok adatait.

## Projekt részletes bemutatása

### Asztali alkalmazás

A játékot a Unity keretrendszerével foglaltuk technikai keretbe. Nemcsak azért, mert játékok gyártásához használatos, hanem mert a C# objektumorientált nyelvet tudtuk használni.

A Játék szerkezetét State Machine technológiával kezeljük, aminek segítségével szabályozni tudjuk, hogy csak akkor léphet a játékos a következő szigetre, amikor az előző szigetek kirakását már teljesítette. Egy ún. PersistentScene vagyis egy folyamatosan élő jelenethez kötött GameManager.cs script kontrollálja, hogy a játékos melyik szigetekhez és játékokhoz fér hozzá, illetve melyik dialógus zajlott már le.

#### Kirakós játékok:

- Puzzle: Egy kép több darabból való kirakása.
- Match3: Randomizált színes elemekből 3-as kapcsolatokat kell létrehozni. Ezek a tábláról eltűnnék, helyükre új elemek kerünek.
- Memória játék: A letakart kártyák közül felfordítással meg kell találni a párokat.
- Hidden Object: Egy háttérképen elrejtett tárgyakat kell megtalálni egy lista alapján.

## Adattáblák

Az adatok tárolásában a következő modellek/entitások segítenek:

- **ApplicationUser:** A felhasználói adatokat tárolja (Microsoft Identity).
- **Character:** A játékban választható karaktereket tárolja.
- **Inventory:** A játékos tárgyait és pontszámát tárolja.
- **InventoryMaterial:** Kapcsolótábla az inventory és a material között.
- **Island:** A játékban lévő szigeteket tárolja.
- **LoginModel:** A bejelentkezési adatokat tárolja.
- **Material:** A játékban lévő anyagokat, tárgyakat tárolja.
- **Player:** A játékos adatait tárolja (pl. játékos neve, karakter, inventory).
- **RecipeBook:** A játékosok receptjeit tárolja.
- **RegisterModel:** A regisztrációs adatokat tárolja.

Először az **alaptáblákat** gondoltuk ki, hogy miket fogunk használni a játék során:

Player,      Inventory,      Character,      Material,      RecipeBook,      Island.

Ahhoz, hogy a **redundanciát** elkerüljük szükséges volt még **kapcsolótáblát** (**InventoryMaterial**) is készítenünk: Van egy "Inventory" (leltár) táblánk, amely a játékosok tárgyait tárolja, és egy "Material" (anyag) táblánk, amely a játékban lévő anyagokat tárolja. Egy játékos leltárában több anyag is lehet, és egy anyag több játékos leltárában is előfordulhat. Ez egy tipikus sok-sok kapcsolat. Ha közvetlenül próbáltuk volna összekapcsolni a két táblát, akkor vagy több anyagot kellett volna tárolnunk egy leltárban (ami nem felel meg az 1NF-nek), vagy több leltár azonosítót kellett volna tárolnunk egy anyagban (ami szintén nem lenne helyes).

Az adatbázisunk **megfelel az első három normálformának**: Az adatok konzisztensek és megbízhatóak, az adatredundancia minimális és az adatbázis könnyen karbantartható és bővíthető.

- **1NF:** Az adatbázisban minden tábla oszlopai atomi értékeket tárol. Például a Player táblában a PlayerName oszlop csak egy játékos nevét tartalmazza, nem pedig több

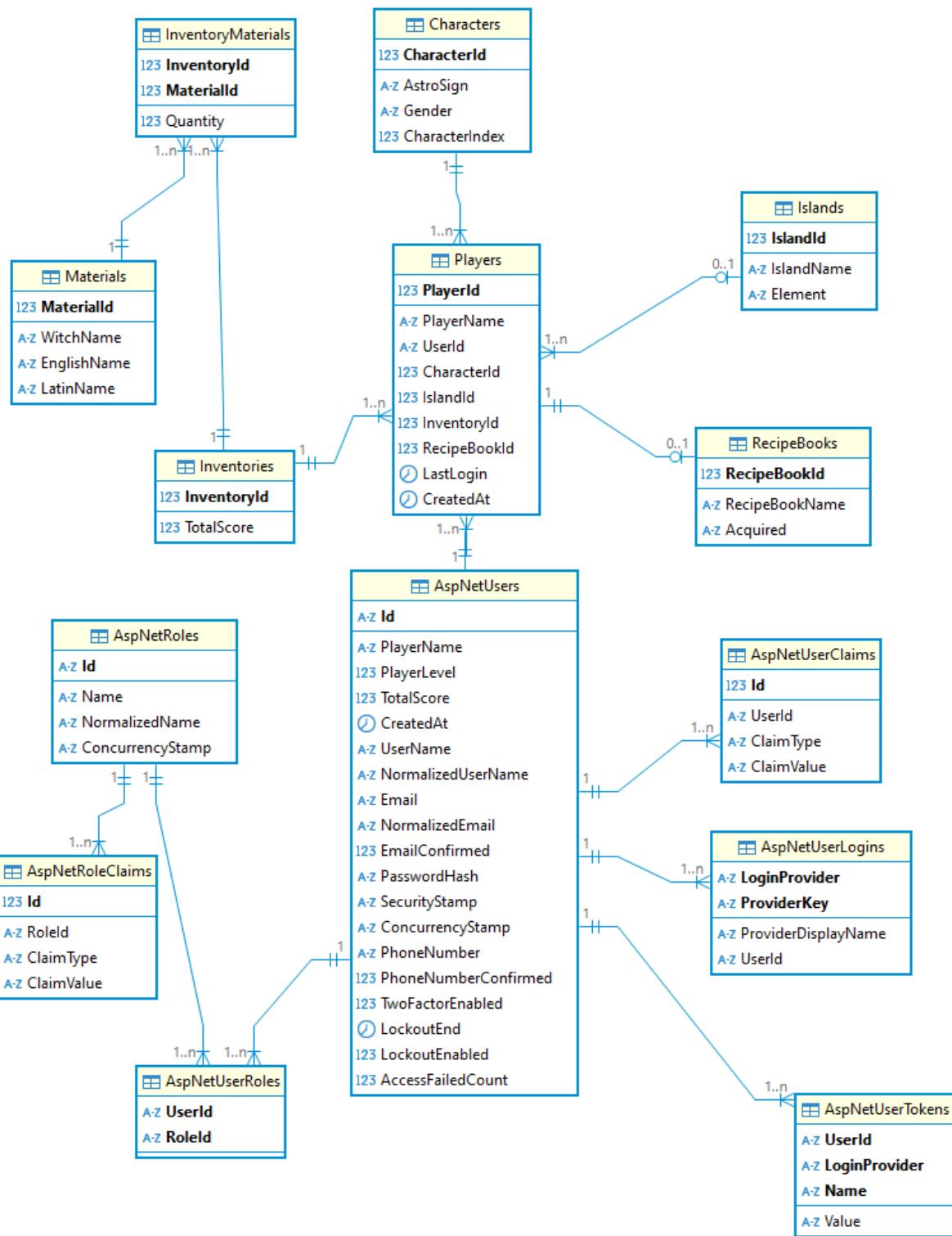
nevet vesszővel elválasztva. Az elsődleges kulcsok (pl. PlayerId, MaterialId) biztosítják, hogy minden sor egyedi legyen.

- **2NF:** minden tábla megfelel az 1NF-nek. A nem kulcs attribútumok (pl. PlayerName, MaterialName) teljes mértékben függnek az elsődleges kulcsoktól. Például a PlayerName egyértelműen azonosítja a PlayerId-hez tartozó játékost.
- **3NF:** minden tábla megfelel a 2NF-nek. Nincsenek tranzitív függőségek. Például a Player táblában a CharacterId nem függ a IslandId-től, hanem minden közvetlenül a PlayerId-től függ. Ahol a Microsoft Identity táblák vannak, azok is megfelelnek a 3NF követelményeinek.

A felhasználók kezelése miatt még több tábla létrehozására volt szükség: ApplicationUser, LoginModel, RegisterModel.

Az ER diagram, azaz az Entitás-kapcsolat ábra, egy vizuális eszköz, ami az adatbázis logikai szerkezetét ábrázolja. Bemutatja az adatbázisban szereplő entitásokat, azok tulajdonságait és az entitások közötti kapcsolatokat. Segíthet az adatbázis tervezésében is, mi a "Code First" módszert alkalmaztuk, az adatbázis kialakítása nálunk a kód fejlesztésével párhuzamosan történt. DBeaver alkalmazással készítettük a grafikont. Az általunk választott aiven.io - felhőalapú adatplatform - kapcsolatának beállításával elérjük a MySQL adatbázist és ez alapján generálódik a diagram. Akármikor módosul a REST API, módosul a diagram is.

Az ábra vizsgálata még a projektünk vége felé is hozott ki az API-ban megoldandó feladatokat, például egy opcionálisnak ábrázolt kapcsolatot: Player és Inventory, amit a módosítások során időközben kötelezővé tettünk.



1. Ábra: ER diagram

## Weboldal

A weboldal **SvelteKit** keretrendszer használ, ami egy hatékony és gyors frontend eszköz webalkalmazások fejlesztéséhez.

Az oldal felépítése *komponens alapú*, ahol a különböző UI elemek különálló, újra felhasználható komponensek formájában lettek megvalósítva.(.svelte fájlok)

Az alkalmazás alap HTML struktúráját, metaadatokat és a SvelteKit direktívákat (%sveltekit.head%, %sveltekit.body%) az **app.html** tartalmazza.

Oldalak és funkcionális szempontjából az alábbi oldalakat szerepeltettük:

**Bejelentkezés (Login):** A felhasználói bejelentkezési felület.

**Főoldal (Home):** a felhasználó bejelentkezés után ide érkezik, itt láthatók a legfontosabb adatok (név, utolsó sziget, pontszám).

**Anyagkészlet (Inventory):** A játékban gyűjthető anyagok listája, illetve a felhasználó által birtokolt anyagok listázása számokkal.

**Karakterek (Characters):** A játékban elérhető karakterek listája.

**Szigetek (Islands):** A játékban található szigetek listája.

**Ranglista (HighScore):** A játékosok TOP10-es rangsorolása pontszám alapján.

Az adatkezelés a komponensek betöltésekor az **onMount** lifecycle hook segítségével történik az adatok lekérdezésére.

A szerverrel való kommunikációhoz a **fetch API**-t használjuk.

A komponensek állapotát **változók** segítségével kezeljük, például **userData**, **character**, **scores**, **materials**.

Az oldalak közötti navigációt a **goto** függvényel valósítottuk meg.

A fő navigációs menü az **oldalsávban (sidebar)** található, ahol a felhasználó a különböző oldalak között válthat.

A **Card** és **CardContainer** komponensek a listák (képek) megjelenítésére szolgálnak. (anyagok, karakterek, szigetek)

A **Modal**-t a részletes információk megjelenítésére használtuk, ami egy felugró ablak formájában jelenik meg.

**Háttérzene** egy dinamikusan változó gomb megnyomása után ki-be kapcsolható.

A különböző képernyőmérethez alkalmazkodást, tehát az oldal reszponzivitását a stílusok @media-val biztosítottuk.

A CSS stílusokkal egyedi megjelenést és stílust adtunk az alkalmazásnak, például háttérképek, betűtípus, színek.

A felhasználók **tokenekkel** azonosítják magukat, amelyeket a **sessionStore** tárol.

A felhasználó a **logoutAndRedirectToLogin** függvényel jelentkezhet ki, ami törli a tokent és átirányítja a bejelentkezési oldalra.

Fejlesztési célra **mock szervert** alkalmaztunk, mely lehetővé tette a frontend fejlesztését a backendtől függetlenül. (Szimulálta az API végpontokat és válaszokat.)

## Projekt feladatainak megosztása

Imre Zsófia	Unity keretrendszer kezelése, Játék általános szerkezetének tervezése, játék: Memória játék, Match3, Puzzle
Pappné Balogh Bernadett	Adatbázis, REST API, játék: Hidden Object játék, kapcsolat diagramok
Réperger Patrícia	Frontend létrehozása, ppt alapszerkezete, skiccek és moodboardok

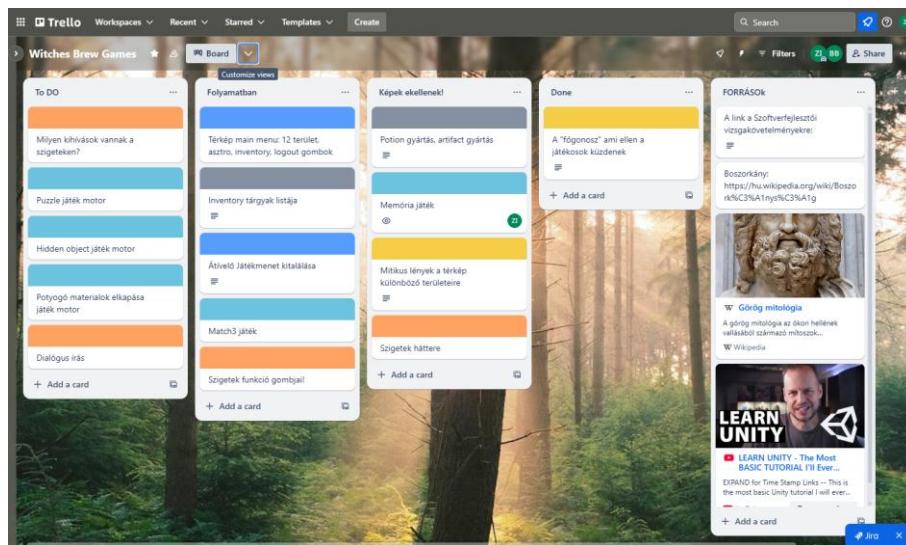
# Projekt ütemezés és alkalmazott módszerek

1. hét	<p><u>Projekt tervezése</u></p> <p>Miután tisztáztuk, hogy játékot szeretnénk készíteni, egy erre alkalmas keretrendszer választottunk, amely segíti és kiszolgálja a céljainkat. Ezért esett választásunk a Unity-re, annak ellenére, hogy az nem a kötelező tananyag része.</p> <p>A megbeszélések során felmerült a multiplayer opción kidolgozása is, de úgy tűnt meg fogja haladni a tudásunk, ezért ezt elvetettük. A játékosok közös felületének a projekthez tartozó weboldalt szándékoztunk megtenni, ahol rangsor listákat és a karakterükhez tartozó ismereteket találhatnak.</p>
2. hét	<p><u>Ötletek szűkítése, konkretizálása</u></p> <p>Ez alkalommal igyekeztünk szűrni, hogy a sok fajta ötletünkben koherens egységet alkossunk.</p> <p>Ekkor véglegesedett az ötlet, hogy szervezzünk minden történet köré, amelyben van egy nagy legyőzendő cél, ami a játékost motiválhatja. Ez lett Kronosz, a mitológiai alak, aki el akarja nyelni a világot.</p> <p>Ez ellen dolgozik a játékos.</p>
3. hét	<p><u>Unity tanulmányozása, 2D játék, szerkezet kitalálása</u></p> <p>Le kellett ellenőrizni, hogy a Unity-vel való játék alkotás ötletünk mennyire kivitelezhető. Ehhez ennek a keretrendszernek a működését kellett tanulmányozni.</p> <p>Láttuk, hogy 2D-s játék készítése számunkra tökéletes sokkal reálisabb számunkra.</p> <p>Ekkor kristályosodott ki az is, hogy a 12 asztrológiai jelnek megfelelően fogjuk a kis játékokat és a történetet rendezni.</p> <p>Így fejezetekre osztva, kezelhető lépésekre oszlott fel a játékkészítés folyamata.</p>
4. hét	<p><u>Relációs táblák készítése</u></p> <p>Miután egyre tisztábbá vált a kép a játék szerkezetről, le tudtunk ülni megbeszélni, hogy milyen adatokra lesz szükségünk.</p> <p>Ekkor gyűjtöttük össze, hogy az egyes játékok során milyen pontokat és alapanyagokat szeretnénk adni a játékosnak, és azokat hogyan szeretnénk használni.</p> <p>Továbbá azt is kitaláltuk, hogy milyen adatokat fogunk a regisztrációhoz és bejelentkezéshez használni.</p>

5. hét	<p><b><u>Stílus, vizuális megjelenés megbeszélése</u></b></p> <p>A játékhoz sok vizuális elemre volt szükségünk. Ezek létrehozásához szerettük volna hozzáfogni még időben. Úgy találtuk, hogy az Art Nouveau stílus olyan megjelenéssel rendelkezik, ami illik a 2D játékhoz és mindenkiunknak tetszik.</p> <p>Mivel sok vizuális elemre volt szükségünk, a DALL-E (chat gpt) képgenerálót vettük igénybe. Egyszerre többen is tudtunk képeket készíteni ugyanahhoz a témahez, miután kikísérleteztük a megfelelő prompt-okat.</p>
6. hét	<p><b><u>Elemek létrehozása Unity-ben</u></b></p> <p>A Unity-ben ún. Jelenetek (Scenes) alapján lehet a játékmenetet megalkotni. A funkcionálisan összefüggő részeket csoportba lehet szedni, és jelenetekbe rendezni.</p> <p>Ez nálunk a 12 szigetet és a rajtuk található kirakós játékokat is jelenti. Ezek készültek el, illetve a gombok, amikkel köztük tudunk majd navigálni. Ezen felül készült még egy jelenet a regisztrációhoz.</p>
7. hét	<p><b><u>Adattáblák</u></b></p> <p>A C# és a .NET Entity Framework segítségével nekiláttunk az adattáblák elkészítésének. A táblák közötti kapcsolatok kialakítása több próbálkozást is igényelt, de végül sikerült megfelelően megvalósítani őket.</p>
8. hét	<p><b><u>API készítés</u></b></p> <p>Ahogy haladtunk előre a C# programozással, és adattábla készítéssel, megtörténtek az első próbálkozások az API végpontok elérésére a játék felületéről.</p>
9. hét	<p><b><u>Egyéni munka, hibajavítás</u></b></p> <p>Az előző héten talált hibákat javítottuk.</p> <p>Elkészült az első kirakós pontozással együtt, ez volt a Memória játék.</p>
10. hét	<p><b><u>StateMachine alkalmazása</u></b></p> <p>Kész a szövegdoboz, elindult a dialógusok gyártása. Amikor a játékos először jut be egy szigetre, más szöveg köszönti a játékost, mint mikor már visszatér oda.</p> <p>Frontend: Mock szerver helyett itt is az API végpontok elérésén dolgoztunk.</p> <p>Bejelentkezést sikeresen teljesen működővé tennünk.</p>

11. hét	<b><u>Refaktorálás</u></b> A szigetek, funkciók és játékok szaporodásával gyűltek az ismétlődő forráskódok, ezért az asztali alkalmazásban refaktoráltuk azokat. Innentől kezdve minden szigetet, annak mentési állapotait, gombjainak működését 1-1 script kezeli.
12. hét	<b><u>Hearth - inventory</u></b> Az Inventory-val együtt elkészül a Crafting system, mely lehetővé teszi, hogy ne csak a játékok során kapunk növényeket. Már azok kombinálásával is kaphatunk tárgyakat. Továbbá elkészült a Match3 játék teljes pontozással.
13. hét	<b><u>Játékokban gyűjtött anyagokat</u></b> Elkezdtük beosztani, mely szigeten lehet őket gyűjteni. Nem minden alapanyagot lehet mindenhol gyűjteni, de próbáltuk kiegyensúlyozni úgy a craft rendszert, hogy ne szenvedjen hátrányt a játékos, ha kifejezetten csak néhány típusú kirakót szeret használni.
14. hét	<b><u>Tesztelés</u></b> A manuális tesztelés után, automata unit teszteket írtunk a programjainkra. mindenki a saját területére írt a teszteket. <b>Játék</b> - Unity beépített tesztelője <b>Backend</b> - Swagger <b>Frontend</b> - Manuális tesztelés
15. hét	Finomítás, tesztelés, grafikai elemek tisztázása és igazítása. Elkészült a klasszikus Puzzle játék, és a Hodden Object játék.
16. hét	Utolsó javítások, dokumentáció befejezése, és a dokumentáció alapján mindenki beletette a saját részét a prezentációs power point-ba.

## Nyomonkövetés:



2. Ábra: Trello

A projekt csoportos nyomon követésére két módszert alkalmaztunk. A feladatok nyilvántartására és követésére a Trello alkalmazás ingyenes változatát használtuk. A részletesebb és közvetlenebb megbeszélésekhez pedig a Discord alkalmazást vettük igénybe, mivel ott azonnal értesültünk róluk, ha valaki üzent valamelyik chatszobában.

## Projekt részeinek technikai kifejtése és a létrehozásukra alkalmazott eszközök

### Asztali játék

A játék a **Unity 5** keretrendszerben készült asztali alkalmazás, amelynek működtető kódja **C# nyelven** íródott. A Unity biztosítja azt a környezetet, amely összekapcsolja a C# kódot a játék felülettel.

Amikor egy **GameObject**-et vagy **UI-elemet** kóddal szeretnénk befolyásolni, az adott kódot tartalmazó osztálynak örökölnie kell a **MonoBehaviour** osztály tulajdonságait. Ezt a Unity biztosítja számunkra. Az így létrehozott scriptet egy **jelenetben található objektumhoz** (**GameObject**) kell komponensként hozzáadni, különben a kód nem fog lefutni, és az objektumokra mutató változók nem lesznek megfelelően beállíthatók.

Az alábbi példában a jelenet **Hierarchyia nézete** látható, amely tartalmaz egy **LoginManager** nevű GameObject-t:

A játék működését vezérlő **scripteket** bármely aktív objektumhoz hozzáadhatjuk, de az átláthatóság érdekében érdemes ezeket egy jól megnevezett, különálló objektumhoz kapcsolni.

Ha a **LoginManager** objektumra kattintunk, a **fejlesztői felületen** (Inspector ablak) láthatjuk a hozzáadott **scriptet**. Ha még nincs hozzáadva, itt is megtehetjük.

A képen látható **Input mezők** és **gombok** a **Canvas** objektum alatt találhatók a Hierarchiában.

Az Inspector ablakban rendeljük hozzá ezeket a megfelelő változókhöz.

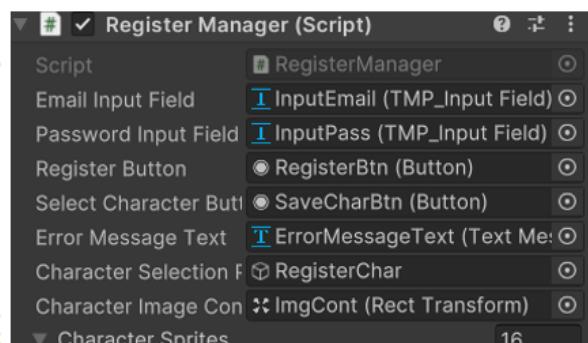
A változók az Inspectorban azért láthatók, mert a C# kódban **public** módon lettek definiálva, vagy ha **privát** változók, akkor kaptak egy **[SerializeField]** attribútumot.

Az Inspector és a kódbeli megfelelője:

```
public class RegisterManager : MonoBehaviour {
    public static RegisterManager Instance; // Singleton pattern

    public TMP_InputField emailInputField;
    public TMP_InputField passwordInputField;
    public Button registerButton;
    public Button selectCharacterButton;
    public TMP_Text errorMessageText;
    public GameObject characterSelectionPanel;
    public Transform characterImageContainer;

    [SerializeField] private Sprite[] characterSprites;
    3 references
    public Sprite[] CharacterSprites => characterSprites;
    private int selectedCharacterIndex = 0; // Kiválasztott karakter index
}
```



3. Ábra: Publikus mezők és az Inspektorbeli

Mint látható, a **selectedCharacterIndex** változó nem látható az inspectorban, mert **egyszerű privát** változó. A változók típusai, mint **Button**, **TMP\_text**, **GameObject**, stb, mind a Unity keretrendszere által vannak számunkra biztosítva.

## Regisztráció, bejelentkezés

A játék kezdetén a játékosnak **be kell jelentkeznie vagy regisztrálnia**. A hozzá tartozó adatokat helyileg is tároljuk a **PlayerPrefs** osztály elemeiben, így a játék működéséhez **nem szükséges állandó internetkapcsolat**. A **PlayerPrefs** a Unity beépített, egyszerű adatmentési megoldása, amely lehetővé teszi az alapvető játékosadatok tárolását.

Az adatokat emellett **egy Cloud szolgáltatásban futó MySQL-adatbázisban** is eltároljuk. Ez lehetővé teszi, hogy a játékos **adatvesztés nélkül** válthasson eszközt, és biztosítja, hogy a későbbiekben a weboldalon is megtekinthesse saját adatait.

## Helyi adatok tárolása

A játék során gyakran váltunk **jelenetek között**, ami azt jelenti, hogy egyes **játékelemek és scriptek** időszakosan aktívvá vagy inaktívvá válnak. Ez segít elkerülni a felesleges memóriahasználatot, és optimalizálja a teljesítményt.

Van azonban egy **állandóan aktív jelenet**, amelyet **PersistentScene** névre kereszteltünk. Ehhez kapcsolódnak a következő fontos komponensek:

- **GameStateMachine** és **GameManager** scriptek
- A REST API végpontokkal kommunikáló Unity-kód

Ezekre a játék bármely pontján szükség van, mivel ők felelősek a **játékosadatok frissítéséért** és **létrehozásáért**. Bár a felhasználó számára ezek a háttérben futnak, a játék egészéhez nélkülözhettetlenek, hiszen a **State Machine** rendszer biztosítja a **játékmenet stabil kereteit**.

## Játékmenet és mechanikák

A játékos egy **Idő Boszorkány**, akinek feladata, hogy **megakadályozza Kronoszt**, a főgonoszt, hogy **felfalja a világot**. Amíg a játékos **elég Idő Pontot** gyűjt, addig a világ nem omlik össze.

Az Idő Pontokon kívül a játékos különböző **alapanyagokat** is gyűjthet, amelyeket kombinálva új tárgyakat készíthet. Ezek a tárgyak:

- **Megkönnyítik a játékmenetet**, vagy
- **Előremozdítják a történetet**, amely a kirakós játékokat összefűzi.

A gyűjthető pontok és alapanyagok **motivációt adhatnak** a játékosnak a **későbbi visszatérésre**, még egy alapvetően alkalmi kirakós játék esetében is. **Hosszú távon** a játékos **számszerűsítve láthatja**, mennyi minden sikerült összegyűjtenie a játékban eltöltött idő alatt.

## A szigetek és a játékmenet struktúrája

A játékmenet **12 szigetre** oszlik, melyek a **12 asztrológiai jegyet** szimbolizálják. minden szigeten az adott **jegyhez tartozó Karakter** mutatja be a játékosnak az ott elérhető lehetőségeket. minden sziget saját **kirakós játékot tartalmaz**, de mivel az összes szigeten sok **azonos funkciójú gomb és mechanika** található, egyetlen **IslandManager** script kezeli minden a 12 szigetet.

A szigeteken belül egy gombbal indíthatók el a **különböző kirakós játékok**, amelyekhez egyedi script tartozik.

## Fontos Unity-metódusok

A szigeteken és a játék többi jelenetében használt scriptek több fontos metódust örökölnek a **MonoBehaviour** osztályból:

- **private void Start() {}**
  - Ez a metódus fut le **elsőként**, amikor egy script meghívódik.
  - **Feladata:** A jelenetek felépítése, például a **játék táblák** és **egyéb játékelemek létrehozása**.
- **private void Update() {}**
  - Ez a metódus **minden képkockánál egyszer lefut**, így frissíti a jelenetet és reagál a **felhasználói interakcióra**.
- **private void Awake() {}**
  - Ezt a metódust általában arra használjuk, hogy **Singletont (Instance) hozzunk létre**.
  - A Singleton biztosítja, hogy egy adott osztály példánya **megmaradjon**, még akkor is, ha a jelenet, amely létrehozta, **betöltésre kerül vagy elhagyjuk azt**.
  - Normál esetben egy jelenetből való kilépéskor  **minden hozzá kapcsolódó objektum törlődik a memóriából**, de a **Singletonok megőrzik az adataikat**.
    - Példák Singletonokra:
      - **GameManager** – a játékos előrehaladásának kezelése.

- **InventoryManager** – a játékos gyűjteményének és tárgyainak nyilvántartása.

## Craft rendszer

A játékos által összegyűjtött növényeket és craftolt tárgyakat két külön Dictionary-ben tároljuk. Ezeket a PlayerPrefs-be is elmentjük, hogy az adatok két játékalkalom között is megmaradjanak.

### **Inventory és Crafting helyszíne**

- Az Inventory-t a **Hearth** nevű jelenet tartalmazza, amely a főmenüből érhető el.
- Itt található a **crafting rendszer** is, amely lehetővé teszi a játékos számára, hogy az összegyűjtött alapanyagokból új tárgyat készítsen.

### **Craftolás folyamata**

1. A játékos kiválaszthat egy **craftolható tárgyat** a megfelelő gomb megnyomásával.
2. Ekkor megjelenik a **tárgy elkészítéséhez szükséges alapanyagok** listája.
3. A kód ekkor ellenőrzi, hogy a játékos rendelkezik-e a szükséges mennyiséggel.
  - Ha igen, a rendszer kiírja, hogy **craftolható**.
  - Ha nem, akkor figyelmeztet, hogy hiányoznak alapanyagok.
4. Ha a játékos megnyomja a „Craft It!” gombot:
  - A szükséges alapanyagokat **levonjuk** az inventoryból.
  - A játékos **megkapja az új tárgyat**.
  - A változásokat **azonnal mentjük a PlayerPrefs-be**, hogy a létrehozott tárgyak és a maradék alapanyagok a következő játékalkalom során is elérhetők maradjanak.

Ezzel a rendszerrel a játékos **tartós előrehaladást** érhet el, és az alapanyagok gyűjtése, valamint a tárgyak craftolása hosszú távú motivációt biztosít.

## Endgame

A 12. sziget teljesítésével a játékos valódi végzett időboszorkává válik. Feladata Kronosz mindenkorai visszatartása.

A játékos **bármelyik szigetre beléphet**, és a már ismert kirakós játékok teljesítésével **tovább gyűjtheti az alapanyagokat**. Ez lehetőséget ad arra, hogy **további tárgyakat craftoljon**, ezzel könnyítve a játékmenetet vagy **új narratív elemeket fedezzen fel**. A kirakósok egyszerűsége lehetővé teszi, hogy a játék élvezhető legyen jelentős időbefektetés nélkül is.

### Online bővítési lehetőségek

A **MySQL adatbázis** használata lehetővé teszi a **tartalom bővítését** a későbbiekben, így a játék nem ér véget a fő történeti szál befejeztével. Az online adatkezelésnek köszönhetően:

- **Új craftolható tárgyak, szigetek vagy események** adhatók a játékhoz frissítésekkel.
- **Szezonális kihívások** szervezhetők, amelyek időszakosan elérhető **extra jutalmakat** kínálnak.
- A játékos **statisztikáit** és **előrehaladását** könnyen szinkronizálhatjuk több eszköz között.

Ezzel a rendszerrel az **endgame tartalom** hosszabb távon is vonzó marad, és a játékos folyamatosan találhat új motivációt a visszatérésre.

## Weboldal

### Fejlesztési környezet:

**IDE:** Visual Studio Code

**Keretrendszer:** Svelte Kit

**Backend kapcsolat:** Mock szerverrel tesztelve

**Nyelv:** Javascript, Typescript, HTML, CSS

**Tesztelés:** Állandó fejlesztés közbeni tesztelés különböző képernyő méreteken.

A weboldal SvelteKit alkalmazással készült, mely egy modern frontend keretrendszerre épül, amely lehetővé tette a dinamikus felhasználói felületek létrehozását, mely egy külső API-n keresztül kommunikál a backenddel az adatok lekérdezéséhez és tárolásához.

```
PS C:\Users\Patricia\Astrowheel> npm run dev
> astrowheel@0.0.1 dev
> vite dev

VITE v6.1.0 ready in 50285 ms

→ Local: http://localhost:5173/
→ Network: use --host to expose
→ press h + enter to show help
```

4. Ábra: Svelte telepítés

Fejlesztése a különböző elemek figyelembevételével történt: felhasználói interakciók, hitelesítési folyamatok, dinamikus tartalomkezelés, valamint reszponzív megjelenés. Az oldal kizárolag a játékban regisztrált felhasználók számára érhető el!

A projekt során sikeresen létrejött egy letisztult, funkcionális és felhasználóbarát felület, amely képes megfelelően kezelni a játékosok adatait és statisztikáit.

A rendszer főbb oldalait a felhasználó az oldalsávon keresztül érheti el, beleértve a karakterek, szigetek, ranglista és a felhasználó anyagkészletének oldalait.

A folyamatos tesztelés biztosította a kód stabilitását és hibamentességét. Az utolsó lépések közé tartozott a valódi backend kapcsolat integrálása.

## Az oldal struktúrája és funkcionálitása

### Folder struktúra

#### \src\app.html

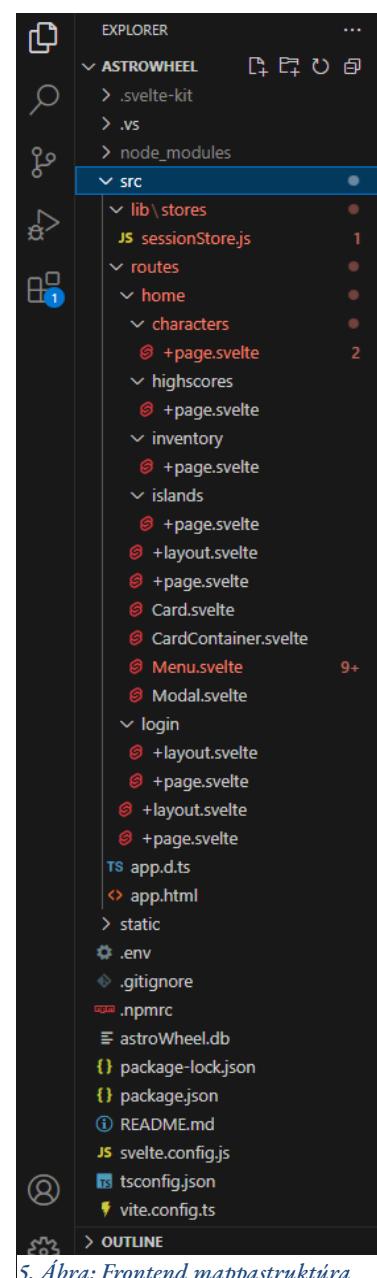
Az app.html az alkalmazás fő HTML struktúráját határozza meg. Tartalmazza a metaadatokat (charset és viewport), a favicon beállítását (link tag segítségével történik) és a SvelteKit által generált tartalmat (%sveltekit.head%, %sveltekit.body%).

#### \.env

Az .env egy környezeti változókat tároló fájl, amely a későbbiekben a program bármely részén használható. Itt található a Mock server és a valós backend kapcsolathoz szükséges kódrészlet.

#### \static

Ebben a mappában találhatjuk az oldalon felhasznált képeket, audio-t.



5. Ábra: Frontend mappastruktúra

## \src\lib\stores\sessionStore.js

A sessionStore.js egy egyszerű *munkamenet-kezelőt* valósít meg, amely a böngésző sessionStorage-jét használja a felhasználói token tárolására.

A **goto** függvény importálása segít az oldalak közti átirányításban.

Munkamenet-kezelő függvények:

- **setToken(token)**: beállítja a felhasználói tokenet a sessionStorage-ban.
- **getToken()**: visszaadja a felhasználói tokenet, ha nincs token átirányítja a felhasználót a főoldalra.
- **clearToken()**: törli a felhasználói tokenet a sessionStorage-ból.

A böngésző sessionStorage-ja ideiglenes tárolást biztosít, amely a böngésző bezárásával törlődik. A **goto(" /")** hívás garantálja, hogy ha nincs érvényes token akkor a felhasználó a főoldalra kerüljön.

```
1 import { goto } from "$app/navigation";
2
3 const createSessionStore = () => {
4
5   return {
6     setToken: (token) => {
7       sessionStorage.setItem('token', token);
8     },
9     getToken: () => {
10       try {
11         const token = sessionStorage.getItem('token')
12         if (!token) {
13           goto('/');
14         }
15         return token;
16       } catch (e) {
17         // handle url based search without login
18       }
19     },
20     clearToken: () => {
21       sessionStorage.removeItem('token');
22     }
23   };
24 }
25
26 export const sessionStore = createSessionStore();
```

6. Ábra: Munkamenetet kezelő függvények

## \src\routes

A SvelteKit-ben a routes mappa szerepe a weboldal URL struktúrájának felépítése. Az ebben szereplő almappák nevei lesznek később az URL címek részei. Például a routes mappában szereplő home mappa a weboldal futtatásakor “<host cím>/home”.

## \src\routes\+layout.svelte

Ez a fájl az alkalmazás legkülső elrendezését határozza meg. Tartalmazza a fő <main> blokkot, amelybe beilleszthetők a különböző oldalak tartalmai.

A **\$props()** függvény lehetővé teszi a komponens tulajdonságainak elérését. Ebben az esetben a **children** tulajdonságot használjuk.

A <main> blokk tartalmazza a **@render children()**-t, amely a különböző oldalak tartalmát jeleníti meg.

A CSS stílusok meghatározzák az oldal kinézetét, beleértve a háttérképet, a betűtípust és a fő tartalom elrendezését.

A `@render` lehetővé teszi a dinamikus tartalom megjelenítését.

A `:global(body)` garantálja, hogy a stílusok az egész oldalra vonatkozzanak, ne csak a jelenlegi komponensre.

### \src\routes\+page.svelte

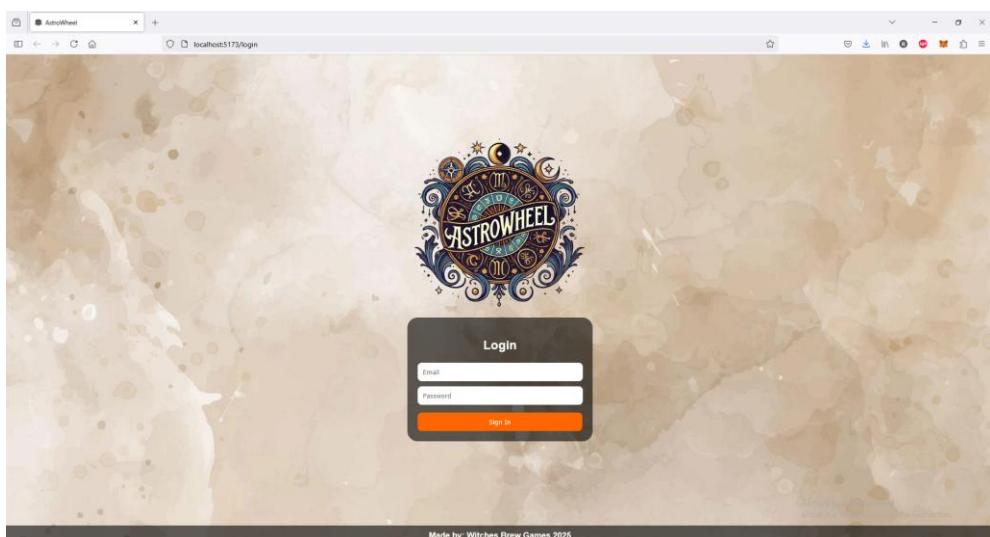
Amikor a felhasználó megnyitja a weboldalt a “`<host cím>/`” elérési útvonalon, akkor azonnal átirányítja a felhasználót a bejelentkezési oldalra. Ezt az `onMount` függvény importálása tette lehetővé azzal, hogy az oldal betöltődésekor meghívjuk a `goto (“/login”)` függvényt, amely átnavigálja a felhasználót.

```
1 <script>
2   import { onMount } from "svelte";
3   import { goto } from "$app/navigation";
4
5   onMount(() => {
6     |   goto("/login");
7   });
8 </script>
9 |
```

7. Ábra: Login függvény meghívása

### Bejelentkezés:

- Csak a Unity játékban regisztrált felhasználók számára elérhető a játékban használt email címmel, illetve jelszóval.
- A hitelesítéskor JWT tokent kapunk vissza a backend-től, amit sessionStore-ban tárolunk a frontenden. minden request tartalmazza ezt a tokenet az Authorization headerben, amelyet a backend fog ellenőrizni.
- Sikeres bejelentkezés után a felhasználót a főoldalra irányítja.



8. Ábra: Weboldal login page

## \src\routes\login

Ebben a mappában található a bejelentkező oldal +page.svelte, +layout.svelte fájlja.

Az oldal lehetővé teszi a felhasználók számára, hogy e-mail címmel és jelszóval hitelesíték magukat.

Tartalmazza a felhasználói adatok validálását, a bejelentkezési kérés kezelését, valamint a felhasználói élményt javító vizuális elemeket.

## \src\routes\login\+page.svelte

*A felhasználói adatok validálása:* az email és password változók tárolják a felhasználó által megadott adatokat.

A validate() függvény ellenőrzi, hogy az e-mail cím érvényes formátumú-e és, hogy a jelszó nem üres-e.

A handleSignIn() függvény felelős a bejelentkezési folyamat kezeléséért. Először validálja a felhasználói adatokat, majd, ha azok érvényesek, elküld egy POST kérést az /api/Auth/login végpontra. A válasz alapján beállítja a sessionStorage-ban a tokent, ha a bejelentkezés sikeres, vagy megjelenít egy hibaüzenetet, ha a bejelentkezés sikertelen.

Az isLoading változó szabályozza, hogy a bejelentkezés gomb aktív-e vagy sem, így megakadályozza a többszörös kattintást.

A CSS stílusok biztosítják a felhasználóbarát és esztétikus megjelenést.

A goto függvény lehetővé teszi az oldalak közötti átirányítást.

A sessionStorage egy tároló, amely a felhasználói munkamenet adatait kezeli.

PUBLIC\_SERVER\_URL egy környezeti változó, amely a szerver URL-jét tárolja.

A fetch API-t használtuk a szerverrel való kommunikációhoz, mely POST kérést küld az adatok hitelesítésére.

Amennyiben a rendszer nem elérhető vagy hitelesítési hibák történnek akkor különböző hibaüzenetek jelennek meg.

## \src\routes\login\+layout.svelte

```
21 |     async function handleSignIn() {
22 |         let errorMsg = validate();
23 |         if (errorMsg) {
24 |             alert(errorMsg);
25 |             return;
26 |         }
27 |         try {
28 |             isLoading = true;
29 |             const res = await fetch(PUBLIC_SERVER_URL + "/api/Auth/login", {
30 |                 method: "POST",
31 |                 headers: { "Content-Type": "application/json" },
32 |                 body: JSON.stringify({ email, password })
33 |             });
34 |
35 |             const data = await res.json();
36 |             if (res.ok) {
37 |                 sessionStorage.setItem(data.token);
38 |                 goto("/home");
39 |             } else if (res.status === 401) {
40 |                 alert(data.error);
41 |             } else {
42 |                 alert("Unexpected error!");
43 |             }
44 |         } catch (e) {
45 |             alert("System is down. Please try again later.");
46 |         }
47 |         isLoading = false;
48 |     }
```

9. Ábra: Bejelentkezési folyamat

Egy általános elrendezést biztosít az oldal számára. Ez a fájl felelős a weboldal alapvető kinézetéséért és struktúrájáért. Tartalmazza a logót, a fő tartalmat és a láblécet.

A **page store** importálása lehetővé tette az adott oldal URL-jének elérését.

A **logo Src** változó tárolja a logó kép forrását.

A **<svelte:head>** blokkban állítottuk be a böngészőben megjelenő weboldal ikonját.

A **<main>** blokk tartalmazza az oldal fő tartalmát.

A logó csak a bejelentkezési oldalon jelenik meg.

A **<slot />** helyettesíti a különböző oldalak tartalmát.

A **lábléc** tartalmazza a weboldal készítőinek információját.

A **\$page store** segítségével tudjuk eldönteneni melyik oldalon vagyunk jelenleg.

Az **#if** feltételesen jeleníti meg a logót.

A **<slot />** a SvelteKit routing rendszerének része, amely lehetővé tette, hogy az egyes oldalak tartalma beilleszthető lehessen a layoutba.

A háttérkép beállítása és a flexbox használata biztosítja a tartalom elhelyezését.

A **lábléc** pozicionálása abszolút, hogy minden az oldal alján legyen.

## Home oldal

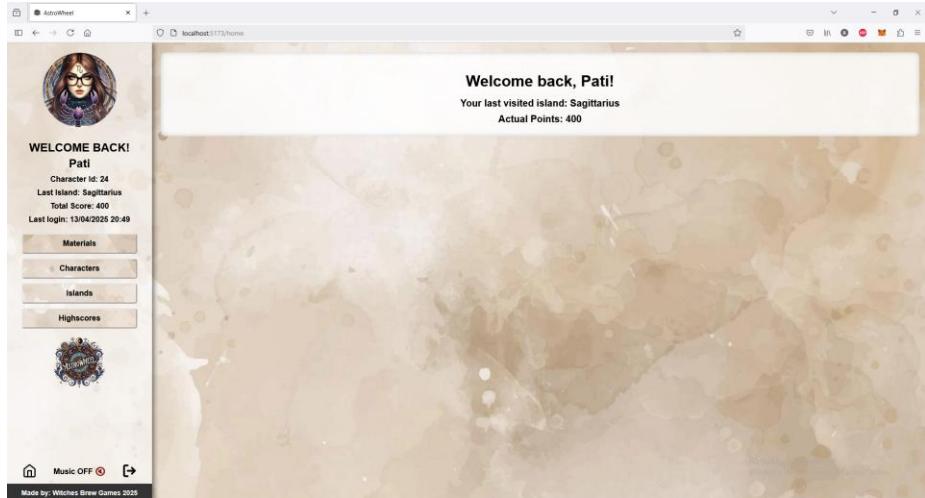
Bal oldali fix Sidebar, amely tartalmazza:

- Játékos által kiválasztott profilkép
- Névhez szóló Welcome Back! üzenet
- Character Id
- Last Island
- Total score
- Last login
- Játék logó

Menüelemek:

- Materials (az összes material képe, kattintás után felugró ablakkal, mely tartalmaz leírást és érdekességet)
- Characters (12 horoszkóp jelképpel, kattintás után felugró ablakkal, két különböző képet ad be, mely adott gombbal váltható)
- Islands (12 sziget képei, kattintás után felugró ablakkal, további információval)
- Highscores (Top 10 játékos listája adatbázisból)
- Home gomb

- Zene ki-be kapcsoló gomb
- Kijelentkezés gomb (visszairányít a bejelentkező oldalra)



10. Ábra: Home page

## \src\routes\home

Ebben a mappában található a Főoldal +page.svelte, +layout.svelte, Card.svelte, CardContainer.svelte, Menu.svelte, Modal.svelte fájlja, illetve az aloldalak és azok fájljai.

## \src\routes\home\Card.svelte

Ez a fájl egy Card komponenst valósít meg, amely egy képet és információt jelenít meg.

A Card tartalmazhat egy képet és leírást.

A kattintásra a Card információi megjelennek egy **Modal (felugró)** ablakban.

## \src\routes\home\CardContainer.svelte

Ez a fájl több Card megjelenítését teszi lehetővé egy container-ben.

Minden Card külön **Modal** ablakot nyit, amelyben további információk láthatók.

A Modal ablakban a Card információi jelennek meg, lehetőség van a képek közötti váltásra.



11. Ábra: Tooltip 1



12. Ábra: Tooltip 2

## \src\routes\home\Menu.svelte

Ez a fájl egy **oldalsávot (sidebar)** hoz létre, amely a felhasználói profil információit jeleníti meg. Navigációs gombokat tartalmaz és lehetővé teszi a zene be- és kikapcsolását, valamint a kijelentkezést.

Az **onMount** függvény segítségével a felhasználó adatait (**userData**) és a karakter adatait (**character**) lekéri a szerverről. A lekérdezett adatok közé tartozik a felhasználó neve, karaktere, utolsó látogatott sziget, pontszám és az utolsó bejelentkezés időpontja.

A dátum formázása **toLocaleString** függvény segítségével történik.

Az **audio** változó egy **HTMLAudioElement** típusú elem, amely a háttérzene lejátszásáért felelős.

A **toggleMusic** függvény segítségével lehet a zenét be- és kikapcsolni.

A **goto** függvény segítségével lehet a különböző oldalak között navigálni (inventory, characters, islands, highscores, home).

A **logoutAndRedirectToLogin** függvény segítségével lehet kijelentkezni és átirányítani a felhasználót a bejelentkezési oldalra.

A felhasználói profil képe, neve és adatai jelennek meg az oldalsáv tetején. A navigációs gombok segítségével lehet a különböző oldalak között navigálni.

A zene be- és kikapcsolására szolgáló gomb, valamint a kijelentkezés gomb azt oldalsáv alján található.

### Technikai részletek:

**Svelte Lifecycle Hook:** Az **onMount** függvény a Svelte lifecycle hook, amely akkor fut le amikor a komponens fel lett csatolva a DOM-hoz.

**Fetch API:** A fetch API-t használja a szerverrel való kommunikációra.

**Session Store:** A **sessionStore** segítségével tároljuk a felhasználói tokent, amely ellenőrzi, hogy a felhasználó be van-e jelentkezve.

A **.profile-image** osztály stílusai a felhasználó profilkép megjelenítését szabályozzák, beleértve a méretet, a lekerekítést és a paddingot.

A **.settings-button-container** osztály stílusai a beállítás gombjainak elrendezését szabályozzák, biztosítva, hogy azok egy sorban helyezkedjenek el.

A `.settings-button` osztály stílusai a beállítások gombjainak alapértelmezett megjelenését szabályozzák, beleértve a kurzort, a hátteret és a paddingot.

A `.settings-button svg` osztály stílusai a beállítások gombjaiban található SVG ikonok megjelenését szabályozzák.

A `.sidebar-text h2` és `.sidebar-text p` osztályok stílusai az oldalsávban megjelenő szövegek (címek és bekezdések) megjelenését szabályozzák, beleértve a margókat.

A `button` elem stílusai az oldalsávban található gombok általános megjelenését szabályozzák, beleértve a margókat, a paddingot, a kurzort, a hátteret, a színt, a betűtípusit és a lekerekítést.

A `button:hover` stílus a gombok hover állapotának megjelenését szabályozza.

A `.sidebar` osztály stílusai az oldalsáv általános megjelenését szabályozzák, beleértve a pozíciót, a méretet, a hátteret, a színt, a paddingot, az elrendezést és az árnyékot.

A `.music-toggle` osztály stílusai a zene be- és kikapcsolására szolgáló gomb megjelenését szabályozzák beleértve a margókat, a paddingot, a kurzort, a hátteret, a színt, a betűtípusit és a lekerekítést.

A `.music-toggle:hover` stílus a zene kapcsoló hover állapotának megjelenését szabályozza.

A `.footer` osztály stílusai az oldalsáv stílusai az oldalsáv alján található lábléc megjelenését szabályozzák, beleértve a pozíciót, a méretet, a hátteret, a színt, a szöveg igazítását és a paddingot.

A `@media` szabályok segítségével különböző képernyőméretekhez igazítjuk az oldalsáv megjelenését, beleértve a profilkép méretét, a gombok szélességét, a logó és a szövegek megjelenítését.

## \src\routes\home\Modal.svelte

Ez a fájl egy **Modal** felugró ablakot hoz létre, amely a felhasználói felületen jelenik meg és lehetővé teszi a felhasználó számára, hogy interakcióba lépjen a tartalommal.

**showModal:** egy bindable tulajdonság, amely meghatározza, hogy a modal ablak látható-e vagy sem.

## Tulajdonságok:

**toggleImage:** egy bindable függvény, amely a képek közötti váltást valósítja meg.

**showFirstImage:** egy bindable tulajdonság, amely meghatározza, hogy a modal ablakban az első kép jelenik-e meg vagy sem.

```
18  <Modal  
19  |  bind:showModal  
20  |  toggleImage={() => toggleImage()}  
21  |  bind:showFirstImage  
22  |  bind:isMultiPage  
23  |>  
24  |  {#if showFirstImage}  
25  |  |  <img src={info1} alt="" class="img-pop-up" />  
26  |  {:#else}  
27  |  |  <img src={info2} alt="" class="img-pop-up" />  
28  |  {/if}  
29  </Modal>
```

13. Ábra: Modal

**isMultiPage:** egy bindable tulajdonság,

amely meghatározza, hogy a modal ablakban több oldal van-e vagy sem.

**children:** egy tulajdonság, amely a modal ablakban megjelenő tartalmat tartalmazza.

## Modal ablak:

A `<dialog>` elem a modal ablakot valósítja meg.

Az `onClose` eseménykezelő a `showModal` tulajdonságot false értékre állítja, amikor a modal ablak bezárul.

Az `onClick` eseménykezelő bezárja a modal ablakot, amikor a felhasználó a modal ablakon kívülre kattint.

A `children` tulajdonságban található tartalom a modal ablakban jelenik meg.

Ha az `isMultiPage` tulajdonság true értékű, akkor egy gomb jelenik meg, amely lehetővé teszi a felhasználó számára, hogy a képek között váltsan.

A CSS stílusok a modal ablak megjelenését szabályozzák, beleértve a méretet, a hátteret, a szegélyt és a paddingot.

Az animációk segítségével lehet a modal ablak megjelenését és eltűnését simábbá tenni.

## Svelte LifeCycle Hook

Az `$effect` függvény a Svelte lifecycle hook, amely akkor fut le, amikor a komponens fel lett csatolva a DOM-hoz.

## HTML Dialog Element

A <dialog> elem egy HTML elem, amely modal ablakot valósít meg.



15. Ábra: Modal megjelenése 1



14. Modal megjelenése 2

\src\routes\home\+page.svelte

Ez a fájl a főoldal tartalmát határozza meg, amely a felhasználó számára üdvözlő üzenetet és információkat jelenít meg.

Az **onMount** függvény segítségével a felhasználó adatait lekéri a szerverről és megjeleníti az oldalon.

A felhasználó neve, utoljára látogatott sziget és pontszáma jelenik meg.

\src\routes\home\+layout.svelte

Ez a fájl az oldalak elrendezését határozza meg, beleértve a bal oldali menüt is.

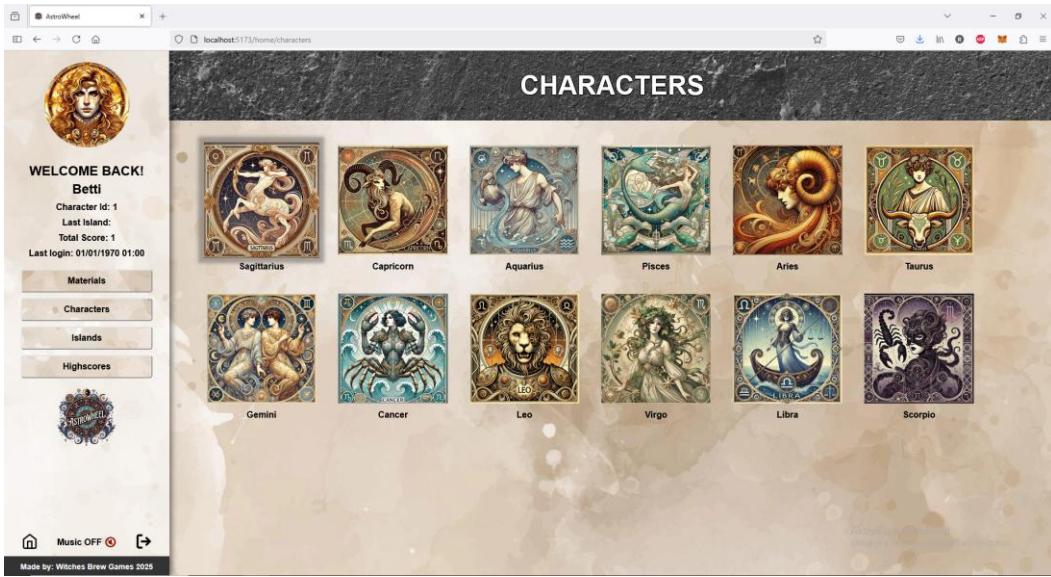
A **Sidebar** komponens importálása és megjelenítése az oldal bal oldalán.

A **children** segítségével a különböző oldalak tartalma beilleszthető a layoutba.

Amennyiben a felhasználó nincs bejelentkezve, akkor “**Unauthorised 403**” üzenet jelenik meg.

\src\routes\home\characters

A karakterek oldal 12 horoszkópot jelenít meg, amelyek képére kattintva további információkat kínálnak.



16. Ábra: Nem játékos karakterek

\src\routes\home\characters\+page.svelte

A komponens a rendelkezésre álló karakterek listáját jeleníti meg a Card-okon, amelyekre kattintva további információk jelennek meg.

Az **images** tömb tartalmazza a karakterek adatait, beleértve a képeket (**src**), az alternatív szöveget (**alt**) és a részletes információkat (**info1**, **info2**).

A home-container és content div-ekkel egy alap elrendezés jön létre, amely tartalmazza a “CHARACTERS” címet.

A **CardContainer** komponens felelős a karakterek adatainak Card formájában történő megjelenítéséért. A **bind:images** attribútummal adja át a karakterek adatait a komponensnek.

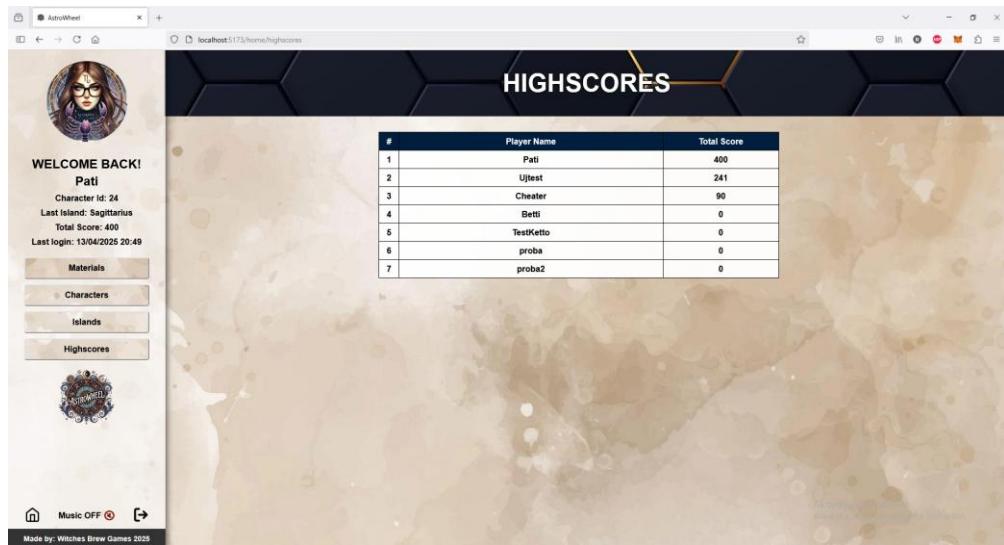
A CSS stílusok határozzák meg az oldal megjelenését, beleértve az elrendezést, a hátteret, a betűtípus és a szöveg árnyékolását.

```
import CardContainer from "../CardContainer.svelte";
let images = [
  {
    src: "/npc Sagittarius.webp",
    alt: "Sagittarius",
    info1: "/Sagittarius_info1.webp",
    info2: "/Sagittarius_info2.webp",
  },
  {
    src: "/npc Capricorn.webp",
    alt: "Capricorn",
    info1: "/Capricorn_info1.webp",
    info2: "/Capricorn_info2.webp",
  },
  {
    src: "/npc Aquarius.webp",
    alt: "Aquarius",
    info1: "/Aquarius_info1.webp",
    info2: "/Aquarius_info2.webp",
  }
]
```

17. Ábra: Képek kezelése

## \src\routes\home\highscores

Ez az oldal a top 10 játékos listáját jeleníti meg az adatbázisból.



18. Ábra: Highscores

## \src\routes\home\highscores\+page.svelte

A komponens betöltésekor az **onMount** függvény segítségével lekérdezi a ranglista adatokat a szerverről. A lekérdezett adatokat a játékosok nevét és pontszámát tartalmazzák. A ranglista adatokat táblázatban jeleníti meg. A **táblázat** tartalmazza a játékosok helyezését, nevét és pontszámát.

A **CSS** stílusok a ranglista oldal megjelenítését szabályozzák, beleértve a hátteret, a betűtípus és a táblázat stílusait.

Az **onMount** függvény egy **Svelte lifecycle hook**, amely akkor fut le, amikor a komponens fel lett csatolva a DOM-hoz.

A **fetch** függvény segítségével a komponens HTTP kéréseket küld a szervernek.

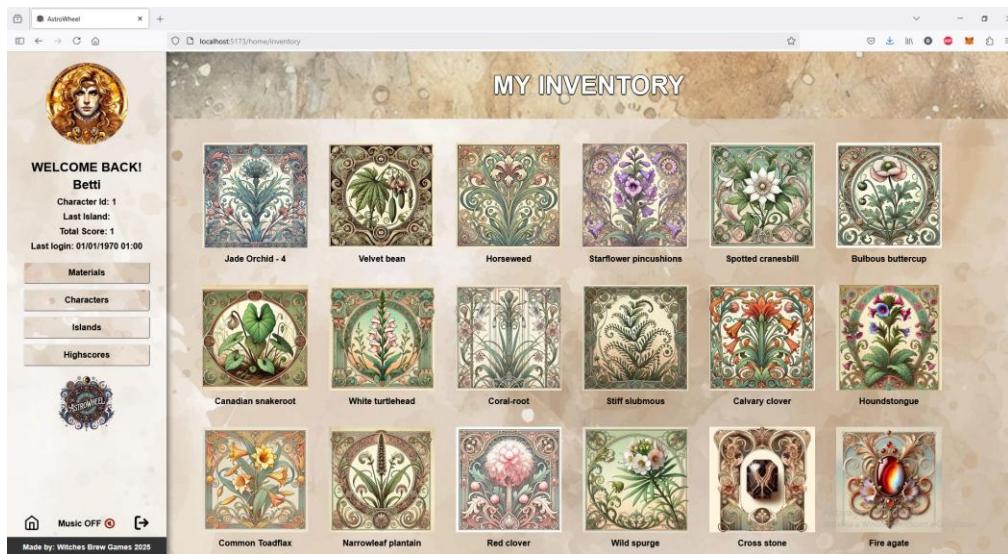
A **sessionStore** segítségével a komponens tárolja és kezeli a felhasználói munkamenetet.

```
onMount(async () => {
  const response = await fetch(PUBLIC_SERVER_URL + "/api/inventory", {
    method: "GET",
    headers: {
      "Content-Type": "application/json",
      Authorization: `Bearer ${sessionStore.getToken()}`,
    },
  });
  scores = await response.json();
});
```

19. Ábra: onMount függvény

## \src\routes\home\inventory

Ez az oldal jeleníti meg az összes material elemet képekkel, illetve a megnevezés mellett jelzi, hogy a játékosnak jelenleg mennyi eleme van.



20. Ábra: Inventory a weboldalon

## \src\routes\home\inventory\+page.svelte

A komponens betöltésekor az **onMount** függvény segítségével lekéri a felhasználó anyagkészletét a szerverről.

A lekérdezett adatok a felhasználó által birtokolt elemek számát és listáját jeleníti meg.

A komponens a **CardContainer** komponens segítségével jeleníti meg az elemeket.

A **CardContainer** komponens a **Card** komponenst használja az egyes elemek megjelenítésére.

Az **onMount** függvény egy Svelte lifecycle hook, amely akkor fut le, amikor a komponens fel lett csatolva a DOM-hoz.

A **fetch** függvény segítségével a komponens HTTP kéréseket küld a szervernek.

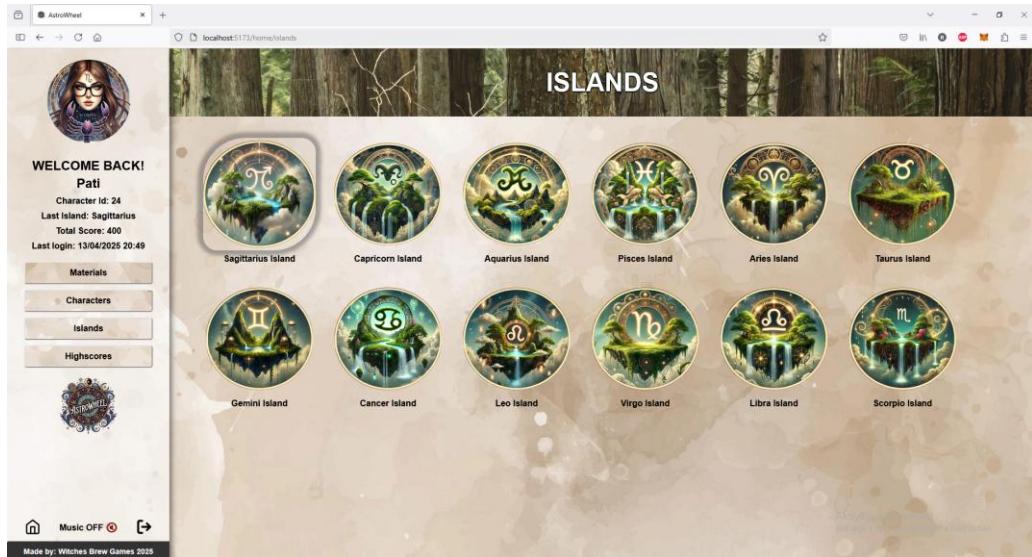
A **sessionStorage** segítségével a komponens tárolja és kezeli a felhasználói munkamenetet.

A **CardContainer** komponens egy másik Svelte komponens, amely az elemek megjelenítéséért felelős.

A **Card** komponens egy másik Svelte komponens, amely az egyes anyagok megjelenítéséért felelős.

\src\routes\home\islands

Ez az oldal jeleníti meg a 12 sziget képeit, amelyekre klikkelve további információkat jelenítenek meg.



21. Ábra: Weboldalak szigetek

\src\routes\home\islands\+page.svelte

Az **images** tömb tartalmazza a szigetek adatait, beleértve a képeket (**src**), az alternatív szöveget (**alt**) és a részletes információkat (**info1**)

A home-container és content div-ekkel egy alap elrendezés jön létre, amely tartalmazza az "ISLANDS" címet.

A **CardContainer** komponens felelős a szigetek adatainak card formájában történő megjelenítéséért. A **bind:images** attribútummal adja át a szigetek adatait a komponensnek.

A **CSS** stílusok határozzák meg az oldal megjelenését, beleértve az elrendezést, a hátteret, a betűtípusát és a szöveg árnyékolását.

A kód importálja a **Card** és **CardContainer** komponenseket, amelyek felelősek a Card megjelenítéséért és a Card elrendezéséért.

A **bind:images** attribútummal adja át a szigetek adatait a **CardContainer** komponensnek.

A **roundedSquare="True"** attribútummal azt jelzi, hogy a Card képei lekerekített négyzetekként jelenjenek meg.

A **CardContainer** komponens dinamikusan generálja a Card-okat az images tömbben található adatok alapján.

## Mock szerver

**Node.js** szerver használatával Express.js segítségével valósítottuk meg.

A mock szerver egy **szimulált szerver**, amelyet fejlesztés és tesztelés során használtunk a valós szerver helyett. Lehetővé tette, hogy teszteljük az alkalmazásunkat anélkül, hogy a valós szervezhez kellett volna csatlakoznunk.

**A szerver a következő funkciókat látja el:**

**Importálás:**

**express:** az Express.js keretrendszer importálása, amely a Node.js-ben futó webalkalmazások és API-k létrehozására szolgál.

**cors:** A CORS (Cross-Origin Resource Sharing) middleware importálása, amely lehetővé teszi a különböző domainekről érkező kérések kezelését.

**Alkalmazás létrehozása:**

`const app = express();` Létrehozza az Express alkalmazást.

**Port beállítása:**

`const port = 3005;` Beállítja a portot, amelyen a szerver futni fog.

**CORS beállítások:**

`const corsOptions = { ... };` Definiálja a CORS beállításokat, amelyek meghatározzák, hogy mely domainekről fogadjon el kéréseket a szerver.

**origin:** A megengedett domain (<http://localhost:5173>).

**optionsSuccessStatus:** A sikeres OPTIONS kérés státusz kódja (200).

**methods:** A megengedett HTTP metódusok (GET, POST, DELETE, PUT, PATCH).

**preflightContinue:** A preflight kérés folytatásának engedélyezése (false).

**CORS middleware használata:**

`app.use(cors(corsOptions));` Alkalmazza a CORS middleware-t az alkalmazásra a definiált beállításokkal.

### **JSON middleware használata:**

`app.use(express.json());`: Alkalmazza a JSON middleware-t az alkalmazásra, amely lehetővé teszi a JSON formátumú kérések feldolgozását.

### **Útvonalak definiálása:**

`app.post("api/Auth/login", (req, res, next) => { . . . });`:

Definiál egy POST útvonalat az /api/Auth/login végpontra, amely a bejelentkezási kéréseket kezeli.

Ha a jelszó “errorTest”, akkor egy 401-es hibakódöt ad vissza “Invalid email or password!” üzenettel.

Egyébként egy sikeres választ ad vissza egy “validMockToken” tokennel.

`app.get("api/character/:characterId", (req, res, next) => { . . . });`:

Definiál egy GET útvonalat az /api/character/:characterId végpontra, amely egy karakter adatait adja vissza. Ez a :characterId egy szám lesz az URL-ben, pl. 1 vagy 17.

`app.get("api/inventory", (req, res, next) => { . . . });`:

Definiál egy GET útvonalat az /api/inventory végpontra, amely egy felhasználó készletének adatait adja vissza.

`app.get("api/TotalScore/:playerId", (req, res, next) => { . . . });`:

Definiál egy GET útvonalat az /api/TotalScore/:playerId végpontra, amely egy játékos összpontszámát adja vissza.

`app.get("api/player/me", (req, res, next) => { . . . });`:

Definiál egy GET útvonalat az /api/players/me végpontra, amely az aktuális felhasználó adatait adja vissza.

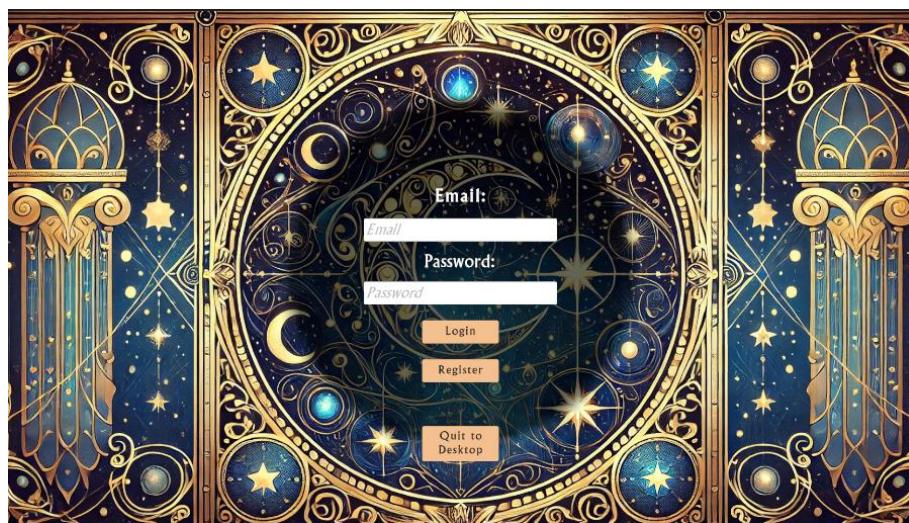
### **Szerver indítása:**

`app.listen(port, () => { . . . });`:

Elindítja a szervert a megadott porton és kiírja a konzolra, hogy a szerver fut.

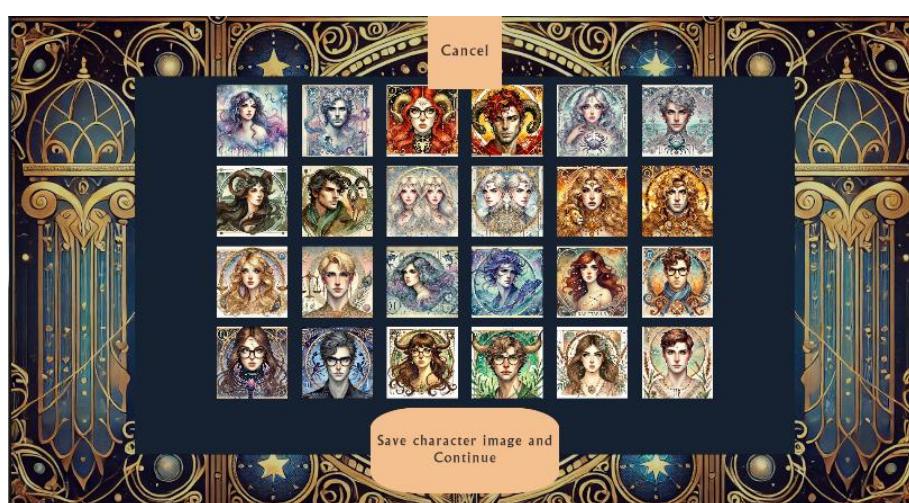
# Felhasználói felületek leírása

## Asztali játék, bejelentkezés



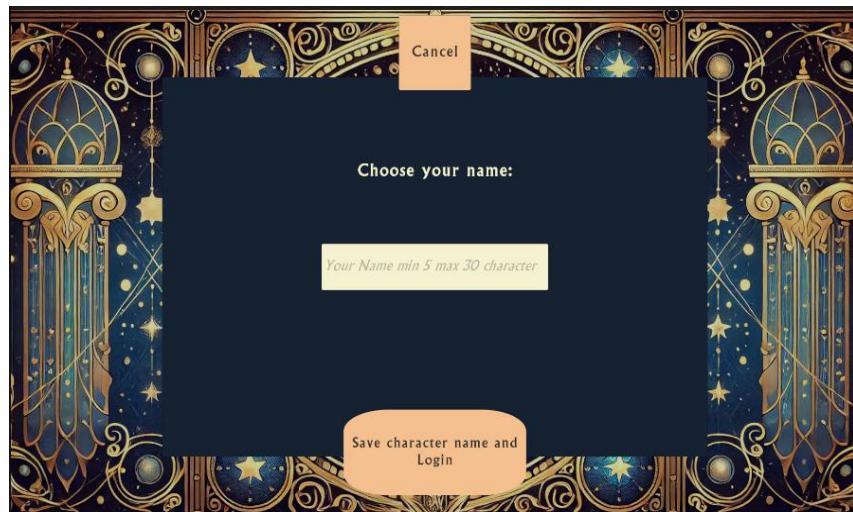
22. Ábra: Játék bejelentekzés

Első alkalommal **regisztrálni** kell. Email és jelszó megadása után a Karakterkép választó képernyő jelenik meg. A kód a Unity Grid layout opcionális segítségével a képeket automatikusan rendezи egymás alá és mellé. A képek egy elmentett listában találhatóak, kényelmesen változtathatóak vagy bővíthetőek. A játékunkban 24 választási lehetőség van, egy női és egy férfi kép minden asztrológiai jegy nevében.



23. Ábra: Karakterkép választó

A **Save character image** gomb lenyomásával elmentjük a képet, majd a Karakter név választó képernyőre jutunk.



24. Ábra: Karakternév választó

A **Save character name gomb** lenyomásával a nevet is elmentjük. Mivel első alkalommal jelentkezik be a játékos, őt rögtön az első sziget jelenetére visszük majd.

Már létező játékos az email és jelszó megadása **Login gomb** segítségével a Főmenübe jut. Innen tud majd navigálni minden olyan szigetre, melynek a kirakós játékát legalább egyszer megoldotta. Vagy arra a szigetre, mely nincs megoldva, de a sorban az következik.

A **Quit To Desktop** gomb kilép a játékból és menti adatainkat a MySQL táblába.

## Asztali játék Főmenü

A játék főmenüjéből juthatunk el a szigetekre, az Inventoryba, vagy ki is jelentkezhetünk.



25. Ábra: Játék főmenü

**A játékos neve:** A regisztrációkor választott név.

**Játékos összpontszáma:** Az egyes kirakósok megoldása során gyűjtött pontokat adjuk itt össze.

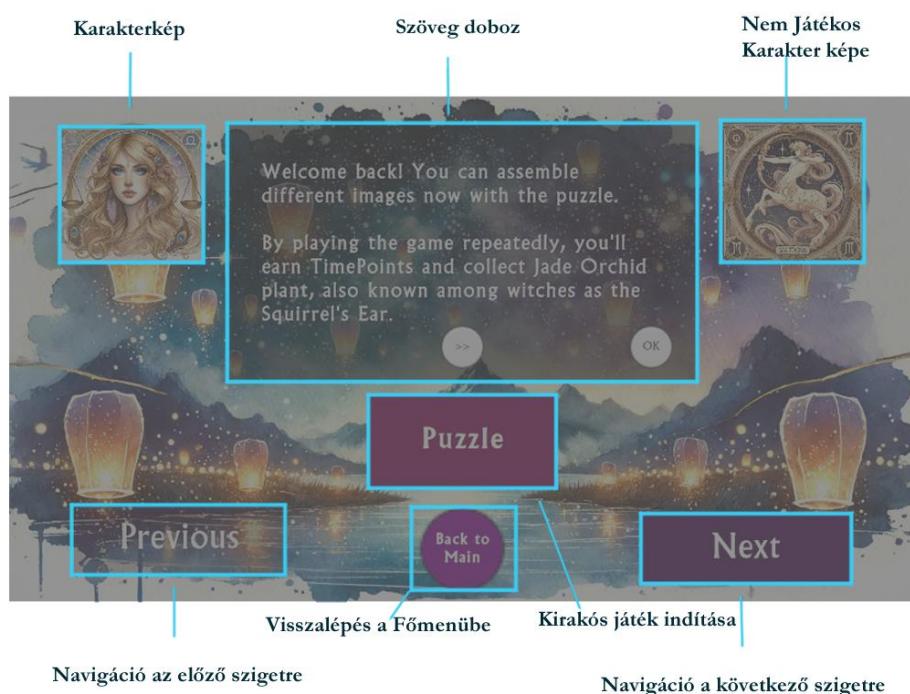
**Utolsó teljesített sziget:** Annak a szigetnek a száma, ahol legalább egyszer meg lett oldva a kirakósjáték.

**Szigetekre vezető gombok:** 12 gomb a szigetek mindegyikéhez. A nem elérhető szigetek pirossal jelennek meg és nem interaktívak.

**Hearth (inventory):** Az összegyűjtött növények és tárgyak listája. Itt lehet a meglévő alapanyagokból újakat is gyártani.

## Asztali játék, szigetek

A 12 szigeten hasonló elrendezésben találhatóak a gombok és panelek, az egységes kinézetért



26. Ábra: Sziget jelenet

**Karakterkép:** A játékos választotta regisztrációkor.

**Nem Játékos Karakter:** A sziget őrzője, a 12 asztrológiai jegynek felel meg.

**Szöveg doboz:** Ha a szigeten lévő kirakós még nincs teljesítve, üdvözlő szöveg fogadja a játékosat. Ellenkező esetben egy leírást látunk, melyben szerepel a szigeten nyerhető alapanyag is.

**Previous és Next gombok:** A szigetek között lehet lépkedni, anélkül, hogy a Főmenübe vissza kellene lépni.

A Previous gomb nem működik az első szigeten. A Next gomb nem működik, ha a következő szigetre még nincs a játékos beengedve vagy a 12. szigeten vagyunk.

**Back to Main gomb:** Főmenübe léphetünk

**A Játék gomb:** Megnyitja a szigethez tartozó kirakós jelenetét

Ami változik szigetenként az a dialógus szöveg, a szigethez tartozó Nem Játékos Karakter, a sziget háttere, és a játék, amihez a szigetről jutni lehet.

## Asztali játék, inventory

A Hearth jelenetben felsoroljuk a játékos összes tárgyát, és lehetőséget adunk újabbak gyártására is.



27. Ábra: Játék Inventory

**Plant Inventory:** A gyűjthető növények felsorolása névvel, képpel és mennyiséggel.

**Crafted Items:** Az elkészíthető tárgyak listája.

Ez utóbbi kettőt a kódból hozzuk létre. Prefabokat használunk ehhez, melyeket annyiszor tesz fel a képernyőre a program, ahány létező növény és készíthető tárgy van. Ezzel a játékot könnyebben lehet bővíteni. A képernyő egyes elemeit nem egyenként helyeztük el.

**Crafting window:** Jobb oldalon a gombokon vannak felsorolva a **receptek**, egy görgethető sávon. Ugyanúgy prefabból jönnek létre, ahogy a tárgyak slotjai.

Bal oldalon pedig egy **ablak**, amin láthatjuk a hozzávalókat, hogy azokból mennyi van nálunk és hány darab tárgyat fogunk kapni eredményül.

A **Craft it!** gombbal tudjuk végrehajtani a tárgykészítés műveletét. A tárgyak mennyiségének változását azonnal mentjük.

## Kirakós játékok

### 1. Nyilas szigete - Puzzle

Ha a játékos először jár itt, akkor mindenkiépp az első képet kell kirakni, ellenkező esetben már egy képválasztó képernyő várja, ahol az első képet, illetve az összes sziget hátteréből bármelyiket kiválaszthatja játék céljából.

A képek egy listából hívódnak meg, így később rugalmasan bővíthető a választék a kód megváltoztatása nélkül. Négyzetes darabokra vágásukról is a kód gondoskodik, így mindegy milyen képarányúak a lista új elemei.



A szerezhető pontmennyiség 3-szorosa az összes puzzle darabkának. A végén levonjuk a lépések számát a pontszámból. Ha ez 0, vagy kisebb akkor is kap 3 pontot a játékos. A nyerhető nyersanyag ennek harmada, minimum 1.

## 2. Bak szigete - Match3

Ez a játék egy klasszikus **match-3 kirakós**, ahol a játékos célja, hogy legalább három azonos ikont illesszen egymás mellé egy rácsszerkezetű táblán. A játék minden körben új elemeket generál, automatikusan eltávolítja az egyezéseket, és feltölti az üres helyeket. Az elemek prefab

alapján készülnek. A játékosnak **15 lépése van**, minden manuális egyezés pontot ér, a pontok pedig egy növényi alapú jutalomként kerülnek a játékos **inventory-jába**. A játék a pontszámot és az előrehaladást elmenti, és a végén lehetőség van újra játszásra is.



29. Ábra: Match3 játék

**6. Bika szigete – Hidden Object Game**

A **Hidden Object** játék során a játékos feladata, hogy egy adott jelenetben elrejtett tárgyakat megtaláljon. A cél a randomizáltan kiválasztott 10 kép kiválasztása 15 közül, melyek listáját a képernyőn is nyomon követheti. minden megtalált tárgy után frissül a hátralévő elemek listája. Amikor a játékos sikeresen összegyűjt az összes céltárgyat, egy növényt kap az inventoryjába, valamint pontszámai és előrehaladása automatikusan mentésre kerül a szerverre. A játék végén lehetőség van az újra játszásra.



30. Ábra: Hidden Object Game

## 7. Ikrek szigete – Memória játék

Ebben a memóriajátékon alapuló minijátékban a játékos célja, hogy az összes **kártyapárt** megtalálja korlátosztott számú próbálkozással. A pálya elején a lapok lefordítva jelennek meg, a játékos pedig minden egyszerre két kártyát fordíthat fel.

A játékos összesen **24 próbálkozást** kap a pálya teljesítésére. minden sikeres találat után pontokat kap – **minél kevesebb próbálkozásból találta meg az adott párt, annál több pont jár érte.** A pontszám és a hátralévő próbálkozások száma folyamatosan frissül a képernyőn.

A játék végén lehetőség van az újra játszásra.



31. Ábra: Memória Játék

## Adatbázis részletes bemutatása

Az *AstroWheelAPI* a felhasználói regisztráció, bejelentkezés, a játékosok adatainak, pontszámának menedzselését és a játékban lévő leltárak, tárgyak, receptek, szigetek kezelését szolgálja a Unity-ben fejlesztett játékhoz. A backend API létrehozásához .NET 8 keretrendszeret használtunk.

Az ASP.NET Core Web API-t azért választottuk adatbázis létrehozására és kezelésére, mert azon túl, hogy hatékony és rugalmas technológia, sokszor foglalkoztunk vele ebben a tanévben, biztosak voltunk benne, hogy a teljesítmény, a modern architektúra és a fejlett fejlesztői eszközök kombinációja alkalmas lesz a célunk elérésére. A játékos regisztrációs adatait, pontszámait, készletét, kellékeit MySQL adatbázisban mentjük el. A Unity nem támogatja a MySQL-hez való közvetlen csatlakozást a biztonsági és platformfüggetlenségi problémák miatt. A leggyakoribb megközelítés az, hogy egy köztes réteget, jelen esetben egy ASP.NET Core Web API-t használunk. Ez a programozási felület az Entity Framework Core objektum-relációs leképező keretrendszer segítségével kezeli az adatbázissal való kommunikációt, amelynek MySQL támogatásához a MySql.EntityFrameworkCore package-t használtuk, a Unity pedig HTTP-kéréseken keresztül kommunikál az API-val. A magas erőforrásigény ellenére, azért esett végül erre a nyílt forráskódú relációs adatbázis-kezelő rendszerre a választásunk, hogy az adatbázisunk interneten keresztül elérhető legyen, amihez az aiven.io menedzselt

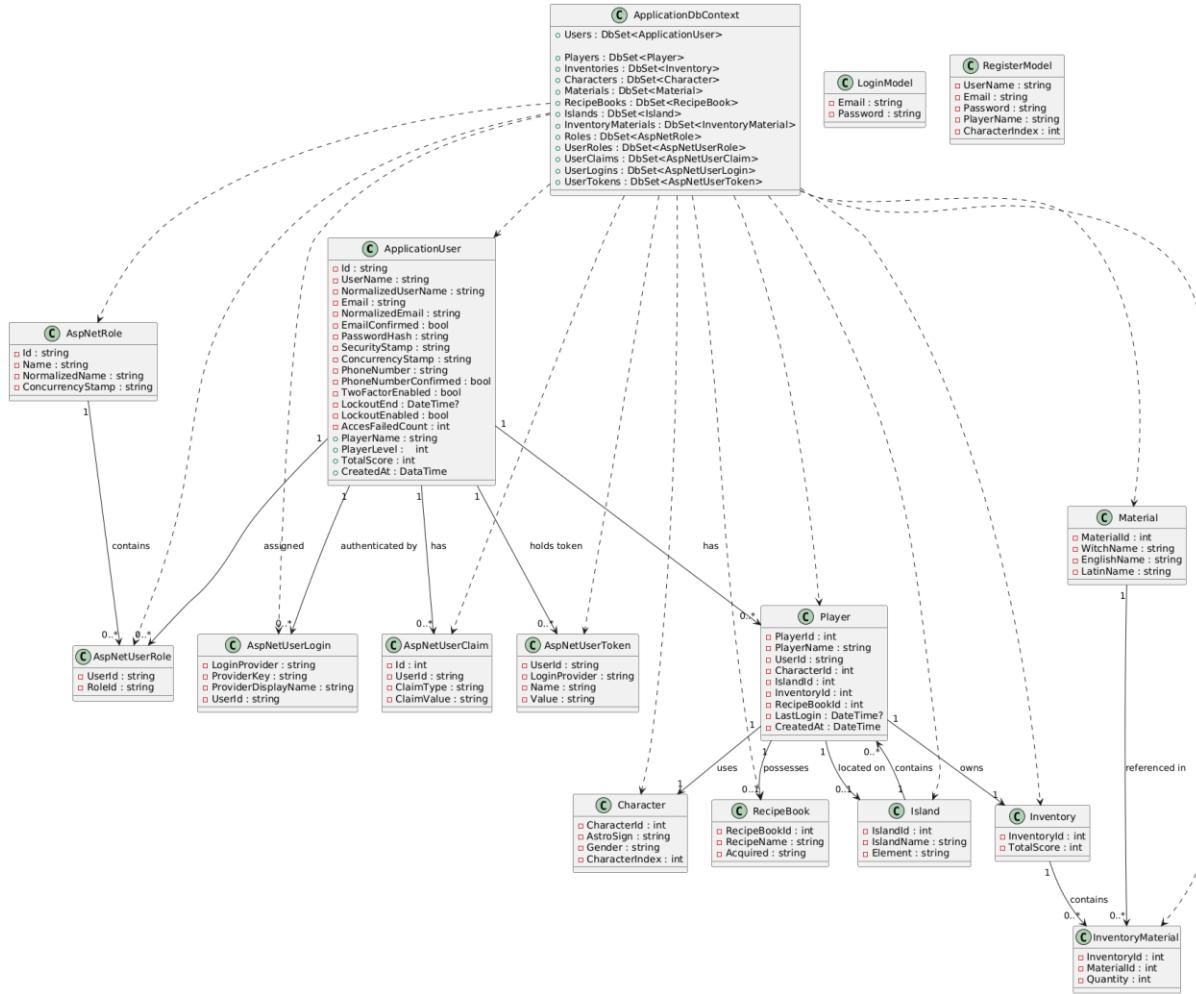
**adatbázis platformot** használtuk. Az adatokat maga az asztali alkalmazás is látja, megjeleníti és frissíti. Mivel az adattárunkat egy felhő alapú alkalmazás platformon, a **render.com**-on üzemeltetjük, a **GitHub**-on tárolt API kód bázis alapján, ez lehetővé teszi, hogy a projektünk bárhonnan kommunikáljon az adatokkal, biztosítva a folyamatos működést. A Webes felületen ezeknek a táblázatoknak a segítségével készítünk ranglistákat. A játékosok láthatják egymás legmagasabb pontszámait és tudomást szerezhetnek olyan alapanyagokról, gyűjthető elemek létezéséről, amelyek esetleg még nem állnak rendelkezésükre.

**Microsoft Identity** használatával - amelynek Entity Framework Core implementációjához a **Microsoft.AspNetCore.Identity.EntityFrameworkCore** package-t használtuk - kezeljük a felhasználói hitelesítést és engedélyezést.

JWT token-ekkel - amelyek generálásához és menedzseléséhez a **Microsoft.AspNetCore.Authentication.JwtBearer** és a **System.Identity.Model.Tokens.Jwt** package-t alkalmaztuk - biztosítjuk a biztonságos API hozzáférést.

A backend eseményeinek és hibáinak naplózására a **Microsoft.Extensions.Logging** package-t használtuk. A loggolás konzolra történik.

Az API végpontjainak manuális teszteléséhez a **Swagger UI**-t integráltuk a projektbe, a **Swashbuckle.AspNetCore** package segítségével. A Swagger dokumentáció automatikusan generálódik az API kódjából. Az adatbázis sémájának kezeléséhez és migrációk létrehozásához a **Microsoft.EntityFrameworkCore.Tools** package eszközeit használtuk.



32. Ábra: UML diagram

A plantuml.com-on plum fájl alapján készült az UML diagram, ami az "AstroWheelAPI" projekt osztályainak és azok kapcsolatainak statikus szerkezetét mutatja be. Ez megtalálható az API-ban.

## Kapcsolatok:

### Asszociációk:

- Az *ApplicationContext* osztály (az Entity Framework Core kontextus osztálya) tartalmazza a DbSet tulajdonságokat az egyes entitásokhoz. **Kezeli és tárolja** az alábbi osztályok adatait:  *ApplicationUser, Character, Inventory, InventoryMaterial, Island, Material, RecipeBook, Player, AspNetRole, AspNetUserRole, AspNetUserClaim, AspNetUserLogin, AspNetUserToken*.

- A *Player* osztály **tartalmazza** az *ApplicationUser*, a *Character* és az *Inventory* osztályt, azaz minden játékoshoz tartozik egy felhasználói fiók, karakter és leltár.
- A *Player* osztály **hivatkozik** az *Island* osztályra, tehát a játékos szigeteken **tartózkodhat**, és **hivatkozik** a *RecipeBook* osztályra, tehát **lehet** neki receptkönyve.
- Az *Inventory* osztály **tartalmazza** az *InventoryMaterial* osztályt, a leltárban lévő tárgyak *InventoryMaterial* objektumokként vannak rögzítve.
- A *Material* osztályt az *InventoryMaterial* osztály **hivatkozza**, az *InventoryMaterial* osztály tárolja, hogy mely *Material* objektumok találhatók a leltárban.
- Az *AspNetUserRole*, *AspNetUserLogin*, *AspNetUserClaim*, *AspNetUserToken* osztályok a Microsoft Identity keretrendszer részei és az  *ApplicationUser* osztályhoz kapcsolódnak. Ezek az osztályok a felhasználói fiókokkal kapcsolatos kiegészítő információk tárolására szolgálnak, mint például a felhasználó szerepkörei, bejelentkezési adatai, jogcímei és tokenjei.
- **Öröklődés:** Az  *ApplicationUser* osztály az *IdentityUser* osztályból származik.

Az API kérések és válaszok kialakítása során a **DTO**-k rendkívül hasznosnak bizonyultak. Elsőként a játékos saját adatainak lekérdezése miatt volt szükség erre az API modellre.

A Data Transfer Object mappával el tudjuk különíteni a külső API kéréseket a belső adatbázis modellektől. Ez egy jó tervezési minta, ami tisztábban és biztonságosabban tartja a kódot. A DTO-k segítenek abban, hogy ne küldjük ki feleslegesen az összes entitás adatot a kliensnek, és hogy csak a szükséges adatokat adjuk vissza az API-k válaszaiban.

A `[JsonIgnore]` attribútumot és a DTO-kat akkor használtuk együtt, amikor meg szerettük volna akadályozni a navigációs tulajdonság szerializációját, ez segített elkerülni a körkörös függőségeket, a nem kívánt adatokat és teljesítmény problémákat, például az *Inventory* és *Material* models osztályokban lévő *InventoryMaterial* kapcsolótábla esetén.

Az adatátviteli objektumok előnyei közé tartozik még, hogy lehetővé teszik az adatok személyre szabását és az adatok összesítését több entitásból.

## A projektben alkalmazott DTO-k:

- **InventoryMaterialDTO:** Az Inventory és a Material entitások közötti kapcsolatot tárolja.
- **InventoryWithPlayerDTO:** Az Inventory tábla kiegészítve a PlayerId és PlayerName tulajdonságokkal.
- **PlayerDTO:** A játékos adatait tárolja, de az adatbázis belső részleteinek elrejtésével, illetve a többi entitásból szükséges adatokkal együtt. Például, a PlayerDTO tartalmazza a TotalScore tulajdonságot, amely több entitásból származó adatok alapján kerül kiszámításra.
- **PlayerMaterialDTO:** A játékos anyagainak átvitelére szolgál a kliens és a szerver között. Az Inventory és Material modellből a szükséges tulajdonságokat tárolja, kifejezetten azokat, amik a PlayerController GetPlayerMaterials metódusában lényeges a kliens számára: MaterialId, WitchName, EnglishName, LatinName, Quantity.
- **PlayerPutDTO:** A játékos két tulajdonságát, a PlayerId-t és a PlayerName-t, illetve az IslandId-t és a LastLogin-t tartalmazza, célzottan azt, amit a PlayerController UpdatePlayer metódusában számunkra szükséges.
- **UserDTO:** A felhasználói adatokat tárolja.

## Funkcionalitás/CRUD műveletek:

A felhasználói regisztráció és bejelentkezés kezelése az AuthController segítségével történik.

**POST /api/Auth/register:** A felhasználók regisztrálhatnak e-mail címmel, felhasználónévvel, jelszóval és játékosnévvel. RegisterModelt használunk. A regisztráció során létrejön egy Inventory és egy Player entitás is. A Player entitáshoz hozzárendelődik egy Character entitás, a felhasználó által Unityn belül kiválasztott karakterkép indexe által, ezt Swagger-ben a CharacterIndex megadásával tudjuk elérni. A regisztrációkor ellenőrizzük a bemeneti adatok helyességét. Részletes hibakezelés és loggolás került bevezetésre.

**POST /api/Auth/login:** A felhasználók bejelentkezhetnek e-mail címmel és jelszóval. Sikeres bejelentkezés esetén a szerver JWT token-t ad vissza. LoginModel osztályt használunk.

**Felhasználók kezelése:** A UsersController segítségével a felhasználókat (ApplicationUser) tudjuk kezelni.

**GET /api/Users:** Lekéri az összes felhasználót az adatbázisból.

200 OK: Sikeres lekérdezés esetén a UserDTO objektumok listáját adja vissza. 500 Internal Server Error: Belső szerver hiba esetén.

**GET /api/Users/{id}:** Lekérdez egy adott felhasználót az azonosítója alapján.

200 OK: Sikeres lekérdezés esetén a UserDTO objektumot adja vissza. 404 Not Found: Ha a felhasználó nem található. 500 Internal Server Error: Belső szerver hiba esetén.

**DELETE /api/Users/{id}:** Töröl egy felhasználót az azonosítója alapján.

204 No Content: Sikeres törlés esetén. 400 Bad Request: Ha a törlés sikertelen (pl. kapcsolódó adatok miatt). A válaszban a hibaüzenetek listája található. 404 Not Found: Ha a felhasználó nem található. 500 Internal Server Error: Belső szerver hiba esetén.

**Játékosadatok kezelése:** A PlayerController a játékosok adatainak kezelésére szolgál. Lehetővé teszi a játékosok adatainak lekérdezését, frissítését és törlését, valamint a játékosokhoz tartozó anyagok lekérdezését. Nincs játékos létrehozási végpont, mert azt az AuthController Post("register") metódusa végzi.

**GET /api/Player/me:** Az aktuálisan bejelentkezett játékos adatait kérdezi le. Ehhez a végponthoz a felhasználónak bejelentkezve kell lennie, és a JWT token-t kell használnia az azonosításhoz. Ez a végpont lekérdezi a játékos adatait a felhasználó azonosítója alapján, és visszaadja a PlayerDTO-t.

Sikeres lekérdezés esetén a 200 OK státuszkódot és a PlayerDTO-t adja vissza. Ha a játékos nem található, a 404 Not Found státuszkódot adja vissza.

**GET /api/Player/{id}:** Egy adott játékos adatait kérdezi le az azonosítója alapján.

Sikeres lekérdezés esetén a 200 OK státuszkódot és a PlayerDTO-t adja vissza. Ha a játékos nem található, a 404 Not Found státuszkódot adja vissza.

**PUT /api/Player/{id}:** Egy adott játékos adatait frissíti az azonosítója alapján. A kérés törzsében a frissítendő adatokat kell elküldeni PlayerPutDTO formátumban.

Sikeress frissítés esetén a 204 No Content státuszkódot adja vissza. Ha a játékos nem található, a 404 Not Found státuszkódot adja vissza. Ha a kérés érvénytelen, 400 Bad Request-et ad vissza.

***DELETE /api/Player/{id}:*** Egy adott játékost töröl az azonosítója alapján.

Sikeress törlés esetén a 204 No Content státuszkódot adja vissza. Ha a játékos nem található, a 404 Not Found státuszkódot adja vissza.

***GET /api/Player/{playerId}/materials:*** Egy adott játékoshoz tartozó anyagokat kérdezi le a játékos azonosítója alapján.

Sikeress lekérdezés esetén 200 ok kódot ad vissza, és egy listát az adott játékos anyagaival. Ha a játékos nem található, 404 Not Found-ot ad vissza.

**Karakterek kezelése:** A CharacterController a játékban található karakterek kezelésére szolgál. Lehetővé teszi a karakterek lekérdezését, létrehozását, frissítését és törlését. A controller naplázást használ az ILogger segítségével a hibák követésére.

***GET /api/Character:*** Lekérdezi az összes karaktert az adatbázisból.

200 OK: Sikeres lekérdezés esetén a Character objektumok listáját adja vissza. 500 Internal Server Error: Belső szerver hiba esetén.

***GET /api/Character/{id}:*** Lekérdezt egy adott karaktert az azonosítója alapján.

200 OK: Sikeres lekérdezés esetén a Character objektumot adja vissza. 404 Not Found: Ha a karakter nem található. 500 Internal Server Error: Belső szerver hiba esetén.

***POST /api/Character:*** Létrehoz egy új karaktert.

201 Created: Sikeres létrehozás esetén az új karakter adatait adja vissza. 500 Internal Server Error: Belső szerver hiba esetén.

***PUT /api/Character/{id}:*** Frissíti egy meglévő karakter adatait az azonosítója alapján.

Kezeli a DbUpdateConcurrencyException kivételt.

204 No Content: Sikeres frissítés esetén. 400 Bad Request: Ha a karakter azonosítók nem egyeznek. 404 Not Found: Ha a karakter nem található. 500 Internal Server Error: Belső szerver hiba esetén.

***DELETE /api/Character/{id}:*** Töröl egy karaktert az azonosítója alapján.

**204 No Content:** Sikeres törlés esetén. **404 Not Found:** Ha a karakter nem található. **500 Internal Server Error:** Belső szerver hiba esetén.

A MaterialController a játékban található anyagok (**Material**) kezelésére szolgál. Lehetővé teszi az anyagok lekérdezését, létrehozását, frissítését és törlését.

**GET /api/***Material***:** Lekérdezi az összes anyagot az adatbázisból.

**200 OK:** Sikeres lekérdezés esetén az Material objektumok listáját adja vissza. **500 Internal Server Error:** Belső szerver hiba esetén.

**GET /api/***Material***/**{*id***}****:** Lekérdez egy adott anyagot az azonosítója alapján.

**200 OK:** Sikeres lekérdezés esetén az Material objektumot adja vissza. **404 Not Found:** Ha az anyag nem található. **500 Internal Server Error:** Belső szerver hiba esetén.

**POST /api/***Material***:** Létrehoz egy új anyagot.

**201 Created:** Sikeres létrehozás esetén az új anyag adatait adja vissza. **400 Bad Request:** Ha a kérés érvénytelen. **500 Internal Server Error:** Belső szerver hiba esetén.

**PUT /api/***Material***/**{*id***}****:** Frissíti egy meglévő anyag adatait.

**204 No Content:** Sikeres frissítés esetén. **400 Bad Request:** Ha az anyag azonosítók nem egyeznek. **404 Not Found:** Ha az anyag nem található. **500 Internal Server Error:** Belső szerver hiba esetén.

**DELETE /api/***Material***/**{*id***}****:** Töröl egy anyagot az azonosítója alapján.

**204 No Content:** Sikeres törlés esetén. **404 Not Found:** Ha az anyag nem található. **500 Internal Server Error:** Belső szerver hiba esetén.

**Inventory kezelése:** Az InventoryController segítségével a játékosok leltárát lehet kezelní.

**GET /api/***Inventory***:** Lekérdezi az összes Inventory-t az adatbázisból, beleértve az Inventory-hoz tartozó anyagokat és a hozzájuk tartozó játékosokat. Az eredményt InventoryWithPlayerDTO objektumok listájaként adja vissza, ahol az eredmények TotalScore szerint csökkenő, majd PlayerName szerint növekvő sorrendben vannak rendezve.

Sikeres lekérdezés esetén a 200 OK státuszkódot és az InventoryWithPlayerDTO objektumok listáját adja vissza. **500 Internal Server Error:** Belső szerver hiba esetén.

**GET /api/Inventory/{id}:** Lekérdez egy adott Inventory-t az azonosítója alapján, beleértve az Inventory-hoz tartozó anyagokat és a hozzá tartozó játékos.

Sikeres lekérdezés esetén a 200 OK státuszkódot InventoryWithPlayerDTO objektumot adja vissza. Ha az inventory nem található, a 404 Not Found státuszkódot adja vissza. 500 Internal Server Error: Belső szerver hiba esetén.

**POST /api/Inventory:** Új Inventory létrehozása.

Sikeres létrehozás esetén 201 Created státuszkódot és az új Inventory adatait adja vissza az InventoryWithPlayerDTO formátumában. 400 Bad Request: Ha a kérés érvénytelen. 404 Not Found: Ha nem található játékos a megadott InventoryId-val. 500 Internal Server Error: Belső szerver hiba esetén.

**PUT /api/Inventory/{id}:** Meglévő Inventory adatainak frissítése azonosító alapján.

Sikeres frissítés esetén 204 No Content-et ad vissza. Ha az inventory nem található, a 404 Not Found státuszkódot adja vissza. Hibás adatok esetén 400 Bad requestet ad vissza. 500 Internal Server Error: Belső szerver hiba esetén.

**DELETE /api/Inventory/{id}:** Egy adott Inventory törlése.

Sikeres törlés esetén 204 No Content-et ad vissza. Ha az inventory nem található, a 404 Not Found státuszkódot adja vissza. 500 Internal Server Error: Belső szerver hiba esetén.

Az InventoryMaterialController az **inventory-k** és **anyagok közötti kapcsolat kezelésére** szolgál. Az InventoryMaterial kapcsolótábla és a Material entitás segítségével a játékosok tárgyait tároljuk. Az InventoryMaterialController és MaterialController segítségével a tárgyakat lehet kezelní. Az InventoryMaterialDTO használatával az inventory és tárgyak közötti kapcsolat, az InventoryWithPlayerDTO-val pedig az inventory és a játékos közötti kapcsolat kerül átvitelre.

**GET /api/inventoryMaterials:** Lekérdezi az összes InventoryMaterial objektumot az adatbázisból, beleértve az Inventory és Material objektumokat is. Az explicit betöltés segítségével biztosítja, hogy az Inventory és Material objektumok is betöltődjenek.  
200 OK: Sikeres lekérdezés esetén az InventoryMaterial objektumok listáját adja vissza. 500 Internal Server Error: Belső szerver hiba esetén.

**GET /api/inventoryMaterials/{inventoryId}/{materialId}:** Lekérdez egy adott InventoryMaterial objektumot az inventory és anyag azonosítója alapján.

200 OK: Sikeres lekérdezés esetén az InventoryMaterial objektumot adja vissza. 404 Not Found: Ha az InventoryMaterial nem található. 500 Internal Server Error: Belső szerver hiba esetén.

**POST /api/InventoryMaterial:** Hozzáad egy anyagot egy Inventory-hoz, vagy növeli a mennyiséget. Ellenőrzi, hogy az Inventory és Material létezik-e az adatbázisban. Naplózza a kivételeket az ILogger segítségével. A kérés törzsében az InventoryMaterialDTO-t kell elküldeni, amely tartalmazza az inventory azonosítóját, az anyag azonosítóját és a mennyiséget.

201 Created: Sikeres létrehozás esetén az új InventoryMaterial objektumot adja vissza. 200 OK: Sikeres frissítés esetén a frissített InventoryMaterial objektumot adja vissza. 400 Bad Request: Ha az InventoryId vagy MaterialId érvénytelen. 409 Conflict: Konkurencia hiba esetén (DbUpdateConcurrencyException). 500 Internal Server Error: Belső szerver hiba esetén.

**PUT /api/InventoryMaterial/{inventoryId}/{materialId}:** Meglévő InventoryMaterial adatainak frissítése, a leltár és az anyag azonosítója alapján.

Sikeres frissítés esetén 204 No Content-et ad vissza. Ha az inventorymaterial nem található, a 404 Not Found státuszkódot adja vissza. Hibás adatok esetén 400 Bad requestet ad vissza. 500 Internal Server Error: Belső szerver hiba esetén.

**DELETE /api/InventoryMaterial/{inventoryId}/{materialId}:** Töröl egy InventoryMaterial objektumot az inventory és anyag azonosítója alapján.

Sikeres törlés esetén 204 No Content-et ad vissza. Ha az inventorymaterial nem található, a 404 Not Found státuszkódot adja vissza. 500 Internal Server Error: Belső szerver hiba esetén.

**Receptkönyvek kezelése:** A RecipeBookController lehetővé teszi a receptkönyvek lekérdezését, létrehozását, frissítését és törlését.

**GET /api/RecipeBook:** Lekérdezi az összes receptkönyvet az adatbázisból.

200 OK: Sikeres lekérdezés esetén a RecipeBook objektumok listáját adja vissza. 500 Internal Server Error: Belső szerver hiba esetén.

**GET /api/RecipeBook/{id}:** Lekérdez egy adott receptkönyvet az azonosítója alapján.  
200 OK: Sikeres lekérdezés esetén a RecipeBook objektumot adja vissza. 404 Not Found: Ha a receptkönyv nem található. 500 Internal Server Error: Belső szerver hiba esetén.

**POST /api/RecipeBook:** Létrehoz egy új receptkönyvet.  
201 Created: Sikeres létrehozás esetén az új receptkönyv adatait adja vissza. 400 Bad Request: Ha a kérés érvénytelen. 500 Internal Server Error: Belső szerver hiba esetén.

**PUT /api/RecipeBook/{id}:** Frissíti egy meglévő receptkönyv adatait az azonosítója alapján.  
204 No Content: Sikeres frissítés esetén. 400 Bad Request: Ha a receptkönyv azonosítók nem egyeznek. 404 Not Found: Ha a receptkönyv nem található. 500 Internal Server Error: Belső szerver hiba esetén.

**DELETE /api/RecipeBook/{id}:** Töröl egy receptkönyvet az azonosítója alapján.  
204 No Content: Sikeres törlés esetén. 404 Not Found: Ha a receptkönyv nem található. 500 Internal Server Error: Belső szerver hiba esetén.

**Szigetek kezelése:** Az IslandController segítségével a játékban lévő szigetek lekérdezését, létrehozását, frissítését és törlését tudjuk elvégezni.

**GET /api/Island:** Lekérdezi az összes szigetet az adatbázisból.  
200 OK: Sikeres lekérdezés esetén az Island objektumok listáját adja vissza. 500 Internal Server Error: Belső szerver hiba esetén.

**GET /api/Island/{id}:** Lekérdez egy adott szigetet az azonosítója alapján.  
200 OK: Sikeres lekérdezés esetén az Island objektumot adja vissza. 404 Not Found: Ha a sziget nem található. 500 Internal Server Error: Belső szerver hiba esetén.

**POST /api/Island:** Létrehoz egy új szigetet.  
201 Created: Sikeres létrehozás esetén az új sziget adatait adja vissza. 400 Bad Request: Ha a kérés érvénytelen. 500 Internal Server Error: Belső szerver hiba esetén.

**PUT /api/Island/{id}:** Frissíti egy meglévő sziget adatait az azonosítója alapján.  
204 No Content: Sikeres frissítés esetén. 400 Bad Request: Ha a sziget azonosítók nem egyeznek. 404 Not Found: Ha a sziget nem található. 500 Internal Server Error: Belső szerver hiba esetén.

**DELETE /api/Island/{id}:** Töröl egy szigetet az azonosítója alapján.

204 No Content: Sikeres törlés esetén. 404 Not Found: Ha a sziget nem található. 500

Internal Server Error: Belső szerver hiba esetén.

**Pontszám kezelés:** A TotalScoreController lehetővé teszi egy adott játékos összpontszámának lekérdezését a játékos azonosítója alapján.

**GET /api/TotalScore/{playerId}:** A lekérdezés során betölti a játékoshoz tartozó inventory-t is, mivel az összpontszám az inventory tulajdonsága.

200 OK: Sikeres lekérdezés esetén a játékos összpontszámát (int) adja vissza. 404 Not Found: Ha a játékos vagy a játékos inventory-ja nem található. 500 Internal Server Error: Belső szerver hiba esetén.

## Design bemutatása

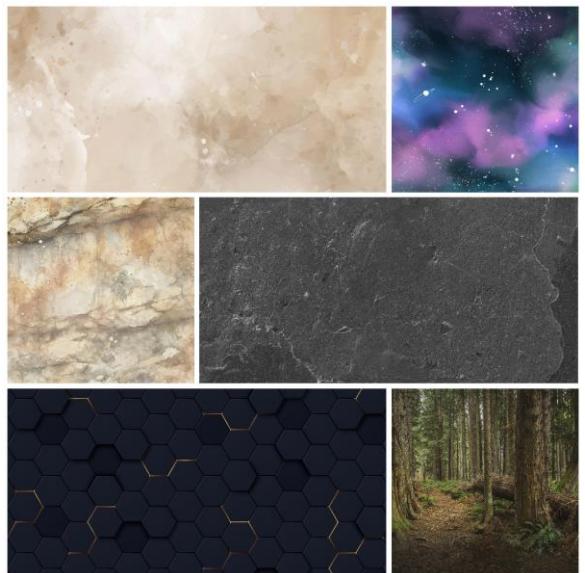
Kreatív anyag készítése, forrása:



33. Ábra: Skiccek

Fő színeknek és stílusnak a misztikus boszorkányos világban gyakran megjelenő színeket használtuk, figyelembe véve a 4 főelem stílusát és megjelenési formáját.

Kezdetben rajzokat, skicceket készítettünk a profilképek, karakterek megjelenítésére, hogy könnyebben el tudjuk képzelní a már kész projektet.



34. Ábra: Moodboard



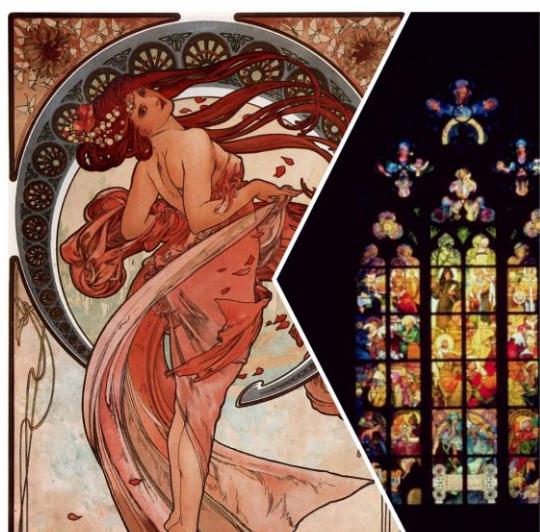
*35. Ábra: Színtáblák*

Kiindulási pontnak a szecessziós művészeti irányzatot vettük alapul. Festészet és építészeti síküveg szakirányban megjelent műveket tanulmányoztuk.

Ihletett merítettünk Alfons Mucha munkáiból.

- Alfons Mucha: Dance (1898),
- Stained-glass Window designed by Mucha at St. Vitus Cathedral, Prague (Early 1930s)

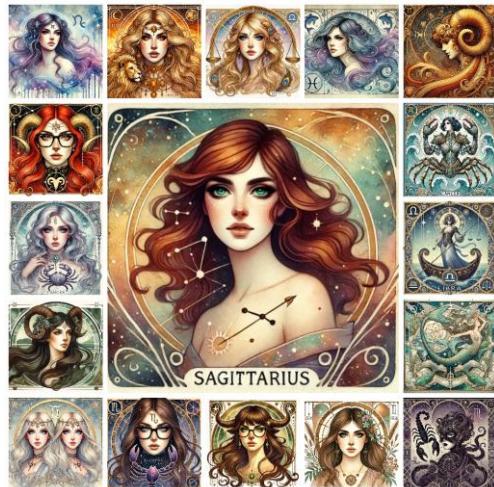
A weboldalon és a játékban felhasznált képek (horoszkóp, profil, szigetek, materials képei, a



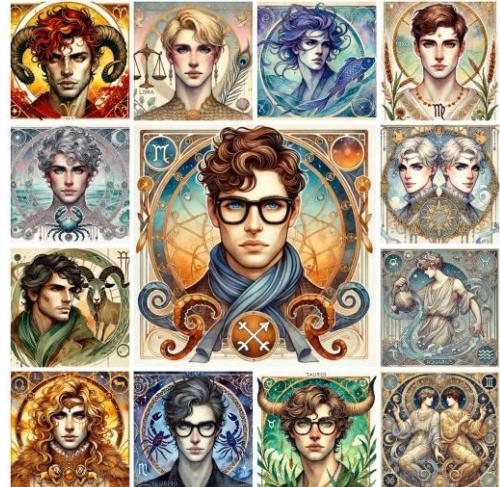
*36. Ábra: Ihlet*

játék logója a chatGPT kép generációjával jöttek létre.

A weboldalon látható felugró ablakokban megjelenő információs grafikai képeket a Canva oldalon készítettük a chatGPT képeit felhasználva.



39. Ábra: Női Játékos képek



38. Ábra: Férfi Játékos képek



40. Ábra: Alapanyagok



41. Ábra: Szigetek képei



37. Ábra: Tooltip designök



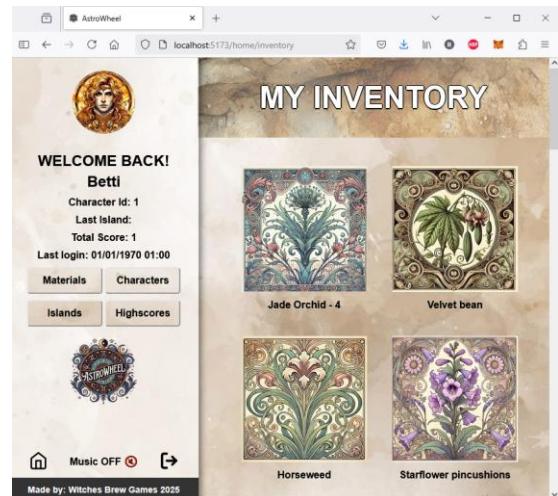
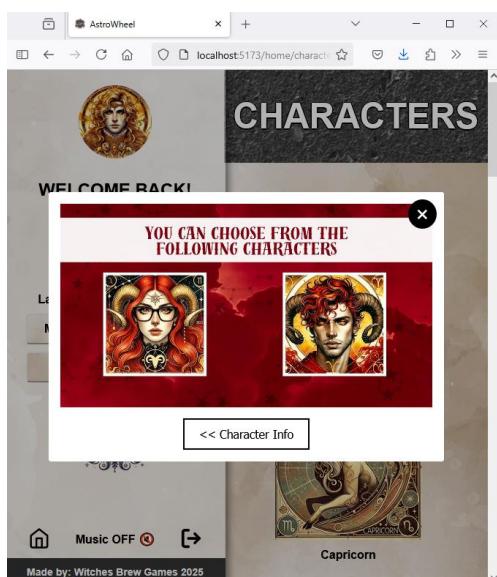
Az oldal gyorsabb betöltése érdekében az oldalon felhasznált képeket .png, illetve .jpg formátumról .webp kiterjesztésűre konvertáltuk.

### Reszponzív megjelenés

A weboldal különböző képernyő méretehez igazodik (kivéve mobil).



42. Ábra: Szigetek tooltip designje



43. Ábra: Képernyőhöz igazítás

## Projekt minőségbemutatása, tesztelés leírása

### Asztali alkalmazás - State machine működése:

Mivel a játék az ún State Machine típusú alap szerkezettel rendelkezik, unit tesztelése során egyértelmű volt, hogy érdemes leellenőrizni, a játékos csak akkor

```
private void Start()
{
    Debug.Log("Beléptél a Főmenübe!");
    // Gombok állapotának beállítása
    for (int i = 0; i < islandButtons.Length; i++)
    {
        int islandIndex = i + 1; // A szigetek indexelése 1-től kezdődik
        if (GameManager.Instance.IsIslandCompleted(islandIndex - 1))
        {
            islandButtons[i].interactable = true;
            Debug.Log($"gomb {i+1} él");
        }
        else
        {
            islandButtons[i].interactable = false;
            Debug.Log($"gomb {i+1} letiltva");
        }
    }
    //gomb lenyomás vizsgálat
}
```

44. Ábra: StateMachine

tud interakcióba lépni az egyes szigetekre vezető gombokkal a főmenüben, ha az azt megelőző jelenetekben található játékokat teljesítette.

Ezt a jobbra látható kód részlet biztosítja.

Az alábbi teszten a játékosnak harmadik szintet állítottunk be, ami azt jelenti, hogy az első három szigetre lehet vissza, plusz a negyedik sziget gombja is interaktívá kell váljon, hogy tovább tudjon haladni.

A tesztet a Unity beépített unit tesztelő motorja segítségével tudtuk vizsgálni. A teszt kód maguknak a gomboknak az interaktivitását vizsgálja.

Az jobb oldali ábrán láthatóan a Test runner

45. Ábra: Unity automata teszt

megjeleníti nekünk a sikeres eredményt zöld pipával. Plusz a játék kódjába ültetett Debug Log megjegyzéseinket, mellyel pontosabban tudjuk követni, hogy mely ágak futottak le.

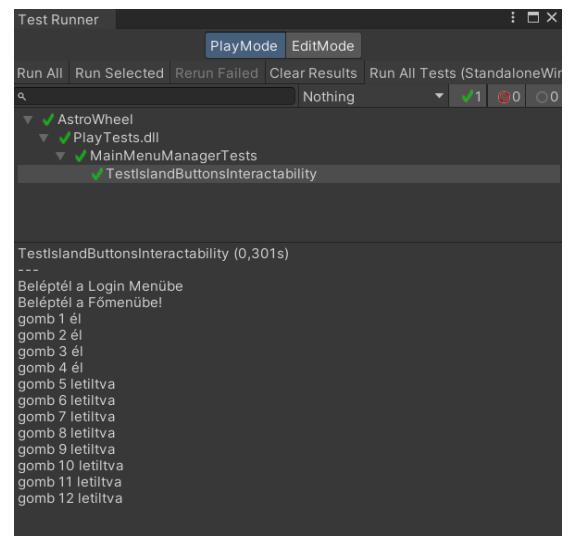
Manuális teszt

Nagyon sok manuális tesztelésen ment keresztül a regisztráció és a login, mire a küldött és kapott JSON file-ok adatai olyanok lettek, amire szükségünk volt. Talán ez volt a legnagyobb kihívása a projektnek a táblázatok kapcsolataival együtt.

## Backend:

Az AstroWheelAPI egységtesztjeit az NUnit keretrendszerrel készítettük, ahol a PlayerController viselkedését validáltuk hitelesítési és adatbázis-interakciós scenáriók mellett. A tesztek izoláltan futnak, a függőségeket (pl. adatbázis) egy in-memory SQLite adatbázissal szimuláltuk Entity Framework Core segítségével, nem pedig Moq-val, hogy valósabb környezetben teszteljünk. A tesztosztály: PlayerControllerTests és a metódusok nevei egyértelműen tükröztek a tesztelt funkciót és elvárt eredményt. A tesztszerkezet az Arrange-Act-Assert mintát követte: adatok előkészítése, metódushívás, eredmény ellenőrzése.

A GetMyPlayer végpont tesztjei két fő esetet vizsgáltak:



Érvényes token esetén a válasz tartalmazza a felhasználó adatait (HTTP 200 OK, PlayerDTO).

Érvénytelen token esetén a rendszer HTTP 401 Unauthorized hibát dob (korábban hibásan 404 Not Found-t adott vissza, amit a vezérlő kódjának módosításával javítottunk).

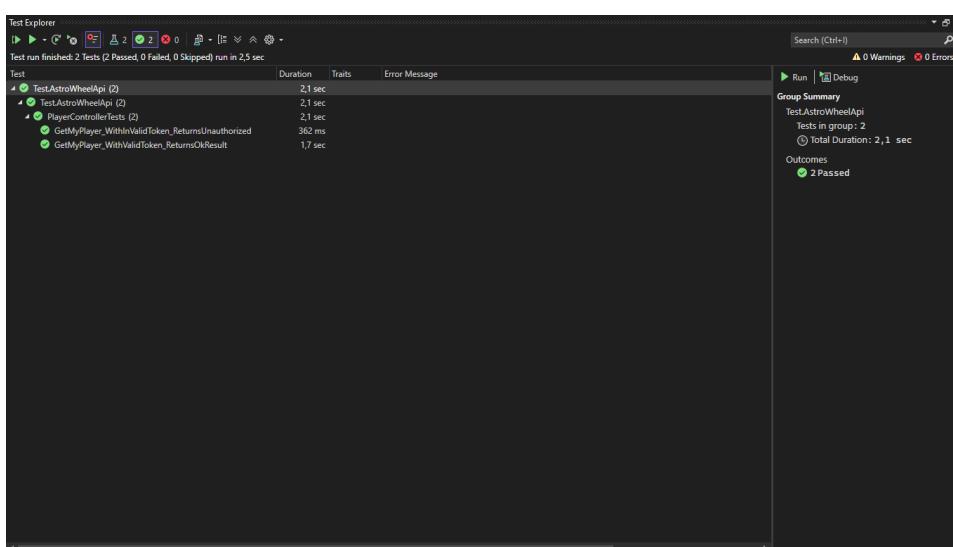
A hitelesítést a tesztekben manuálisan szimuláltuk: érvényes token esetén a felhasználóhoz tartozó ClaimTypes.NameIdentifier-t állítottuk be, érvénytelen token esetén üres ClaimsPrincipal-t használtunk. Az [Authorize] attribútumra nem támaszkodtunk a tesztekben, helyette a vezérlő kódjában explicit módon ellenőriztük a felhasználó azonosítóját.

Az adatbázis-interakciók teszteléséhez minden tesztfuttatás előtt egyedi in-memory adatbázist hoztunk létre (Guid-al generált névvel), és a teszt végén a [TearDown] metódusban felszabadítottuk az erőforrásokat. A tesztek eredményeit a Visual Studio Test Explorer segítségével validáltuk, az Assert.That metódusokkal ellenőrizve a stáruszkódokat, visszatérési értékek típusát és tartalmát. A dokumentáció karbantarthatóságra és refaktorálásbiztonságra fókuszál, és elősegíti a hibák korai felismerését.

Példa eredmények:

- Sikeres teszt: [PASS] GetMyPlayer\_WithValidToken\_ReturnsOkResult
- Sikertelen teszt (javítás előtt): [FAIL]

GetMyPlayer\_WithInValidToken\_ReturnsUnauthorized.



46 Ábra: Backend teszt

A tesztkörnyezet konfigurációja és lefuttatása ismételhető, és integrálható folyamatos integrációs (CI) folyamatokba.

A Swagger-rel való tesztelés elsősorban az **API integrációs és végpontok közötti tesztelés** kategóriájába esik. Segítségével ellenőriztük, hogy az API megfelelően kommunikál-e más rendszerekkel, és hogy az API-folyamatok a várt módon működnek-e minden esetben.

Sikeressé regisztráció alkalmával létrejön a felhasználó és a játékos:

The screenshot shows the Swagger UI interface for a POST request to `https://localhost:7178/api/auth/register`. The response code is 200, and the response body contains the message: "User and Player are registered successfully!". The response headers include `access-control-allow-origin: *`, `content-type: text/plain; charset=utf-8`, `date: Mon,14 Apr 2025 10:08:52 GMT`, and `server: Kestrel`.

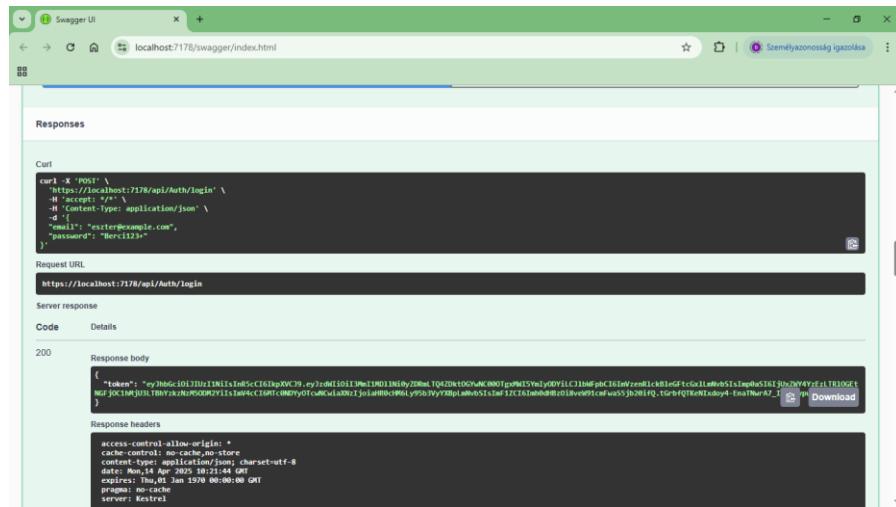
47. Ábra: Swagger - Regisztráció

Sikertelen regisztráció esetén, a jelszó hosszára vonatkozó hibaüzenet:

The screenshot shows the Swagger UI interface for a POST request to `https://localhost:7178/api/auth/register`. The response code is 400, and the response body indicates an error: "Error. response status is 400". The detailed error message is: "The Field Password must be a string or array type with a minimum length of '8'." The trace ID is also provided: "100-0187f0b68bfaf1843de196dd204db309-6d1374fc9d197ac2-00".

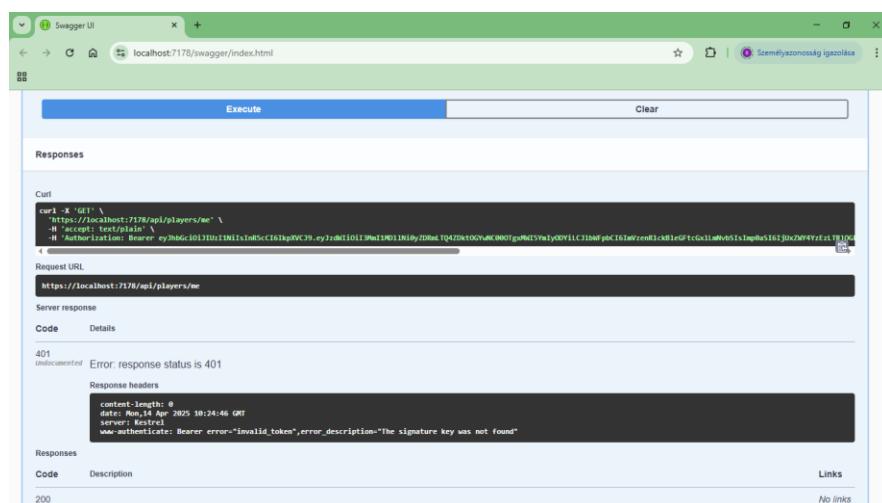
48. Ábra: Swagger- Sikertelen regisztráció

Sikeressé bejelentkezés esetén megjelenik a token, amit hitelesítésnél használunk:



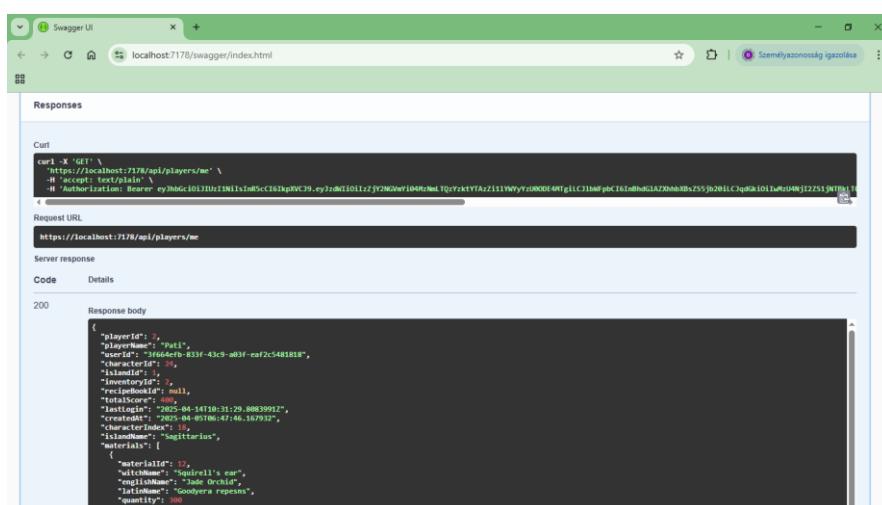
#### **49. Ábra: Swagger – Sikeres Autorizációs Token**

Sikertelen autorizáció esetén:



#### *50. Ábra: Swagger - Sikertelen Autorizáció*

Sikeres lekérdezés GetMyPlayer() végpontnál, bejelentkezés és hitelesítés után:



### 51. Ábra: Swagger - PlayerMe végpont

## Frontend:

### Tesztelési eljárások

A fejlesztés folyamán folyamatos tesztelés történt különböző aspektusokban:

Frontend tesztelés (Svelte Kit)

- Komponensek helyes megjelenítése
- Dinamikus tartalom betöltése mock szerverről.
- Token alapú autentikáció működése
- Navigáció tesztelése (menüpontok közötti váltás).
- Backend kapcsolat tesztelése (Mock szerverrel)
- Tokenek kezelése (bejelentkezés, kijelentkezés)

### Rezonansivitás tesztelése

- Különböző képernyőméretek kezelése (asztali gép, kisebb laptopok, tabletok)
- a developer tools segítségével teszteltük és fejlesztettük az esztétikus megjelenést különböző méreteken

### Hibakezelés tesztelése

- URL alapján történő navigáció megfelelő kezelése
- Token nélküli URL alapján történő átirányítás ellenőrzése

### Tesztelési stratégiák és módszerek

#### 1.Komponens-szintű tesztelés

- Ellenőriztük a rezponsív viselkedést különböző képernyőméretekben
- Interakciók tesztelése (hover, click állapotok)

#### 2.API integrációs tesztek

- Mock szerver használata
- Hibakezelések tesztelése (401, 500 státuszok)

#### 3.Hitelesítési folyamat tesztelése

- Token kezelés tesztesetei
- Session storage helyes működése

#### 4.Responsív design validálás

- Képernyőmrétek szerinti tesztelés
- Chrome DevTools "Device Toolbar" használata
- Cross-browser kompatibilitás ellenőrzése
- Touch események kezelése

#### **Bejelentkezési teszteset:**

-*Input:* Email+jelszó bejelentkezés gombra kattintás

-*Várt kimenet:* Post request küldése a backend-re a megadott e-mail cím és jelszóval. Helyes e-mail cím és jelszó esetén redirect —> /home, sessionStorage-ban megjelenik a token, bejelentkezés az oldalra, látható az üdvözlő üzenetek, character Id, last island és egyéb egyéni adat

-*Teszteset eredmény: sikeres*

#### **Hibás felhasználói teszteset:**

-*Input:* Nincs jelszó, nincs e-mail, helytelen e-mail cím formátum

-*Várt kimenet:* Felugró ablak hibaüzenettel, nem küld requestet a backend-hez, mert ez a validálás frontend-en történik

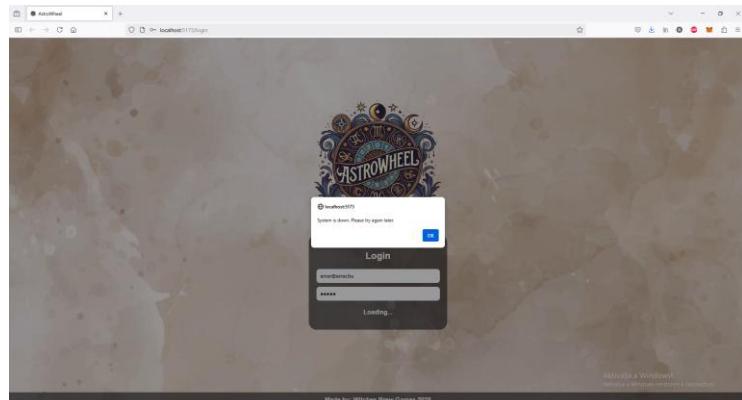
-*Teszteset eredmény: sikeres*

#### **Nem működő backend rendszer teszteset:**

-*Input:* helyes felhasználó

-*Várt kimenet:* lásd a képen

-*Teszteset eredmény: sikeres*



52. Ábra: Nem működő backend teszteset

#### **Debug-olás folyamata**

##### **Chrome DevTools "Device Toolbar"** használata

###### ○ **Vizsgáló**

- Itt található a DOM és a CSS elemek, itt látjuk a html kód felépítését, ami pl. segített abban, hogy megértsük a SvelteKit felépítését és dinamikus CSS osztályait a html tageken.

- Itt ideiglenes CSS adható az oldalhoz amely szignifikánsan csökkenti a fejlesztési és tanulási időt.
- **Konzol**
- Itt találhatóak a különböző error és warning üzenetek. Ezen felül ideiglenes *console.log()* használata amely segíti a hibakeresést.
- **Hálózat**
- Itt láthatóak a kimenő requestek és azoknak az eredményei. (response time, response body, használt headerek)
- **Alkalmazás**
- Itt látható a session storage és így a token
- Szintén itt is lehet cookie-kat és cache-t türíteni
  - **Mock szerver**
    - *console.log()* használata a szerver oldali hibák megértéséhez
    - a fentebb említett DevTools hálózat fül használata

## SWOT

### Erősségek (Strengths)

- **Rugalmas tematika:** Az asztrológiai és misztikus téma jellegzetes és ismert ugyanakkor nagyon rugalmas és sokfajta elemet tartalmazhat.
- **Több kis játék egy rendszerben:** A változatos puzzle típusok szélesebb közönség számára teszik vonzóvá a játékot.
- **Többplatformos elérhetőség:** A játék PC-n és mobilon is fut.
- **Modern technológiák használata:** Unity + ASP.NET Core Web API + MySQL stabil, skálázható architektúrát biztosít.
- **Jól szervezett adatbázis-kezelés:** A relációs adatbázis biztosítja az adatok konziszenciáját és a hatékony adatkezelést.

## Gyengeségek (Weaknesses)

- **Magas erőforrásigény:** A Unity és az ASP.NET Core együttes használata nagyobb teljesítményt igényelhet, főleg mobilon.
- **Biztonsági kihívások:** Az adatbázis és a webes API biztonságos kezelése kiemelt figyelmet igényel, különösen a mobilos hozzáférés miatt.
- **Komplex fejlesztés és karbantartás:** A többplatformos támogatás növeli a fejlesztési időt és a hibakeresés bonyolultságát.

## Lehetőségek (Opportunities)

- **Mobilos piac teljes kihasználása:** A mobiljátékok népszerűsége révén nagyobb közönség érhető el.
- **Bővítés további puzzle játékokkal:** Újabb kirakósokkal frissíthető a játék, növelve az újrajátszhatóságot.
- **Közösségi funkciók fejlesztése:** Multiplayer lehetőségek vagy ranglisták növelhetik a játékos-elköteleződést. Az asztrológiai téma különösen könnyen adja magát egész évben időszakos kihívásokhoz.

## Veszélyek (Threats)

- **Technikai kihívások mobilon:** Teljesítményoptimalizálás és kompatibilitási problémák különböző eszközökön.
- **Erős piaci verseny:** A mobilos puzzle-játékok szegmense zsúfolt, nehéz lehet kitűnni.
- **Adatbiztonsági kockázatok:** A mobilos és online adatok védelme kiemelt fontosságú, GDPR és egyéb szabályozások betartása szükséges.
- **Fejlesztési és kiadási csúszások:** A többplatformos megoldás növelheti a fejlesztési időt és a komplexitást.

## Összegzés

A projekt során együttműködve dolgoztunk a tervezéstől a megvalósításig. Igyekeztünk összehangolni mind a vizsgakövetelményeket, mind a saját igényeinket a játékkal szemben, emellett pedig tartani a határidőket.

A végeredmény egy összetett projekt lett, amely további bővítésre is alkalmas. A közös munka során hasznos tapasztalatokat szereztünk, rengeteget tanultunk, amelyek hozzájárultak a szakmai fejlődésünkhez.

# Források

**Grafikus felületek:** Canva, Clip Studio Paint

**Képek és kódolási problémák:** Chat gpt, Deep seek, Gemini

**Játék keretrendszer:** Unity, Unity asset store

**Weblap keretrendszer:** SvelteKit, Node.js

**Programozás:** Visual Studio 2022

1. Ábra: ER diagram .....	8
2. Ábra: Trello .....	14
3. Ábra: Publikus mezők és az Inspektorbeli .....	15
4. Ábra: Svelte telepítés.....	19
5. Ábra: Frontend mappastruktúra.....	20
6. Ábra: Munkamenetet kezelő függvények.....	21
7. Ábra: Login függvény meghívása .....	22
8. Ábra: Weboldal login page .....	22
9. Ábra: Bejelntkezési folyamat .....	23
10. Ábra: Home page .....	25
11. Ábra: Tooltip 1 .....	25
12. Ábra: Tooltip 2 .....	25
13. Ábra: Modal .....	28
14. Modal megjelenése 2 .....	29
15. Ábra: Modal megjelenése 1 .....	29
16. Ábra: Nem játékos karakterek .....	30
17. Ábra: Képek kezelése.....	30
18. Ábra: Highscores.....	31
19. Ábra: onMount függvény .....	31
20. Ábra: Inventory a weboldalon .....	32
21. Ábra: Weboldalak szigetek .....	33
22. Ábra: Játék bejelentekzés .....	36
23. Ábra: Karakterkép választó.....	36
24. Ábra: Karakternév választó.....	37
25. Ábra: Játék főmenü.....	37
26. Ábra: Sziget jelenet.....	38
27. Ábra: Játék Inventory .....	39
28. Ábra: Puzzle játék .....	40
29. Ábra: Match3 játék.....	41
30. Ábra: Hidden Object Game .....	41
31. Ábra: Memória Játék .....	42
32. Ábra: UML diagram.....	44
33. Ábra: Skiccek.....	53
34. Ábra: Moodboard.....	53
35. Ábra: Színtáblák .....	54
36. Ábra: Ihlet.....	54
37. Ábra: Tooltip designok .....	55
38. Ábra: Férfi Játékos képek .....	55
39. Ábra: Női Játékos képek.....	55
40. Ábra: Alapanyagok.....	55
41. Ábra: Szigetek képei .....	55
42. Ábra: Szigetek tooltip designja.....	56
43. Ábra: Képernyőhöz igazítás .....	56
44. Ábra: StateMachine.....	56
45. Ábra: Unity automata teszt.....	57
46 Ábra: Backend teszt .....	58
47. Ábra: Swagger - Regisztráció .....	59
48. Ábra: Swagger- Sikertelen regisztráció.....	59
49. Ábra: Swagger – Sikeres Autorizációs Token.....	60
50. Ábra: Swagger - Sikertelen Autorizáció .....	60
51. Ábra: Swagger - PlayerMe végpont.....	60
52. Ábra: Nem működő backend teszteset .....	62