



**Eötvös Loránd Tudományegyetem**

**Informatikai Kar**

**Informatikatudományi Intézet**

**Média- és Oktatásinformatika Tanszék**

# Online absztrakt stratégiai társasjáték megvalósítása

**Szerző:**

Gyepes Gergő

Programtervező informatikus BSc.

**Témavezető:**

Korom Szilárd

Doktorandusz

**Budapest, 2023**

Ide kerül a hivatalos témabejelentő lap.

# Tartalomjegyzék

<b>1. Bevezetés .....</b>	<b>1</b>
<b>2. Felhasználói dokumentáció .....</b>	<b>2</b>
2.1. Megoldott probléma rövid leírása .....	2
2.2. Az oldal elérése .....	2
2.3. Az oldal felépítése. ....	2
2.3.1. Navigációs sáv .....	2
2.3.2. Bejelentkezési/regisztrációs felület .....	3
2.3.3. Fő oldal .....	5
2.3.4. Szoba .....	8
<b>3. Fejlesztői dokumentáció .....</b>	<b>14</b>
3.1. Technológiai Stack .....	14
3.2. Rendszer .....	14
3.3. Kezdeti lépések .....	14
3.4. Környezet felállítása a további fejlesztéshez .....	15
3.5. Adatbázis (MongoDB) .....	16
3.6. Socket.IO Integráció .....	18
3.7. Backend (Express.js) .....	19
3.7.1. Felépítés .....	19
3.7.2. Szerver-kliens kommunikáció .....	20
3.7.3. Kontrollerek .....	21
3.8. Frontend (React) .....	24
<b>4. Összefoglalás és további fejlesztési lehetőségek .....</b>	<b>43</b>
4.1. Összefoglalás .....	43
4.2. További fejlesztési lehetőségek .....	43
<b>5. Irodalomjegyzék, hivatkozások .....</b>	<b>44</b>

# 1. Bevezetés

A társasjátékok mindig is különleges helyet foglaltak el az életemben. Gyerekként mindig is szívesen töltöttem a szabadidőm barátaimmal és családommal különböző társasjátékok mellett, és felnőttként is szenvedéllyel hódolok ezen hobbinak. A játékok nem csak szórakoztatóak, hanem kiváló lehetőséget nyújtanak a stratégia gondolkodás, a problémamegoldás és csapatmunka fejlesztésére is. A dolgozatom témájának választott Azul az egyik kedvenc játékom és véleményem szerint kellő kihívást nyújt a játék megvalósítása.

Ez a projekt ötvözi a társasjátékok és webfejlesztés iránti érdeklődésemet. Célom, hogy létrehozzak egy olyan online platformot, amely lehetővé teszi a felhasználók számára, hogy valós időben játszhassanak egymással, bárhol is legyenek a világban. A dolgozat nemcsak a szakmai képességeimet fejleszti, teszi próbára, hanem egyúttal egy szórakoztató eszközt biztosít a társasjátékok kedvelői számára.

A projekt célja egy webalkalmazás fejlesztése, amely lehetővé teszi a felhasználók számára, hogy különböző társasjátékokat játszhassanak online, valós időben. A rendszernek biztosítania kell a következő funkciókat:

- **Felhasználói Regisztráció és Bejelentkezés:** A felhasználók regisztrálhatnak egy fiókot, és bejelentkezhetnek az alkalmazásba, hogy hozzáférjenek a játékokhoz.
- **Játékterek Létrehozása és Csatlakozás:** A felhasználók létrehozhatnak új játéktereket (szobákat), vagy csatlakozhatnak meglévő szobákhoz, ahol más felhasználókkal együtt játszhatnak.
- **Valós Idejű Kommunikáció:** A Socket.IO segítségével valós idejű kommunikáció valósul meg a játékosok között, lehetővé téve a zökkenőmentes játékmenetet.
- **Játéklogika Kezelése:** A szerver oldalon fut a játéklogika, amely biztosítja a játékszabályok betartását és a játékmenet folytonosságát.
- **Mentés:** A játék állapotának folyamatos mentése, visszatöltése és a felhasználók adatainak tárolása a MongoDB adatbázisban történik.

## 2. Felhasználói dokumentáció

### 2.1. Megoldott probléma rövid leírása

A dolgozatomban egy böngészőben játszható társasjáték megvalósítását tűztem ki célul. A weboldalra a felhasználók be tudnak lépni és vagy regisztrálni. Majd ezután elérik a fő oldalt, ahol egy szoba hosztolás – csatlakozás rendszeren került implementálásra. Egy szobához 1-4 ember tud csatlakozni. A szobákban, az egész életciklusuk alatt, használható egy chat ablak a felhasználók közti kommunikációra. A játékot akkor indíthatjuk, ha legalább ketten vagyunk a szobában és a szoba összes tagja jelezte, hogy készen áll.

Ezután a játék automatikusan elindul. Felváltva vehetünk le csempéket a piacokról és helyezhetjük azokat a táblánkra. A játék az Azul társasjáték szabályait követi. Ha egy felhasználó nincs tisztában a szabályokkal vagy esetleg a weboldal használata nem érthető számára, biztosítottam neki egy súgó-t, amelyet a „?” gombbal meg tud nyitni. Itt lehet elolvasni a szabálykönyvet a játékhoz és az oldal használatához biztosított segédletet.

### 2.2. Az oldal elérése

Sajnos a játék, amiről mintáztam az alkalmazást nem public domain, ezért a weboldal nem lett publikálva. Ha ki szeretnénk próbálni, akkor a 3.4-es lépéseknek megfelelően saját (labor) környezetünkben tehetjük meg azt.

### 2.3. Az oldal felépítése.

Az alkalmazás egy úgy nevezett SPA (single page application), de a könnyen érthetőség kedvéért, az egy böngésző oldalon, egy időben megjelenő felületet oldalként fogom nevezni. A fejlesztői dokumentációban részletesebben lesz róla szó.

A weboldal elsősorban nagyképernyőre lett optimalizálva, de kisebb kijelzőkkel is kompatibilis. Kis kijelzős üzemmódban bizonyos funkciókat elrejtettünk, ezeket navigációs gombokkal tudjuk majd elérni.

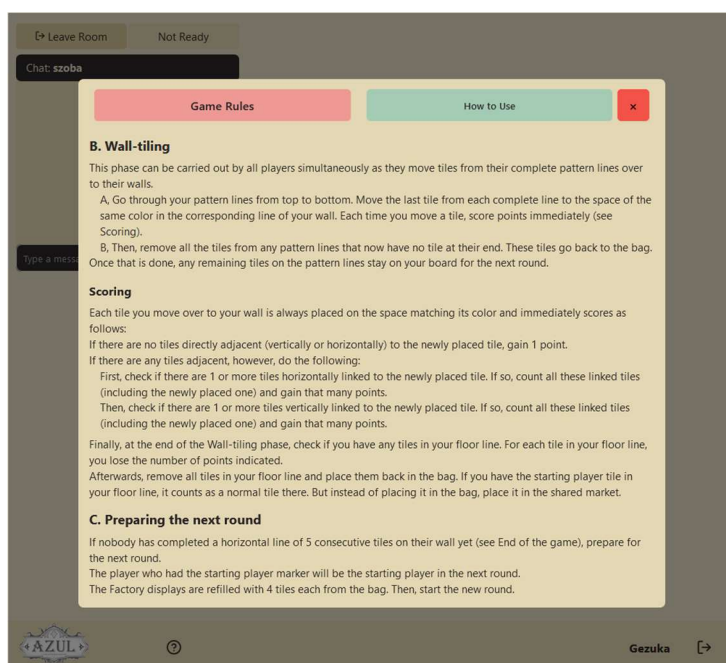
#### 2.3.1. Navigációs sáv

Minden oldalon a lap alján szerepel egy sáv, amelyen megjelenítjük a jelenleg bejelentkezett felhasználó nevét, egy kijelentkezés gombot, egy info/sugó gombot és az oldalunk logóját.

### 1. ábra navigációs sáv

A kijelentkezés gomb és a felhasználó neve csak a fő és játékszoba oldalakon érhető el.

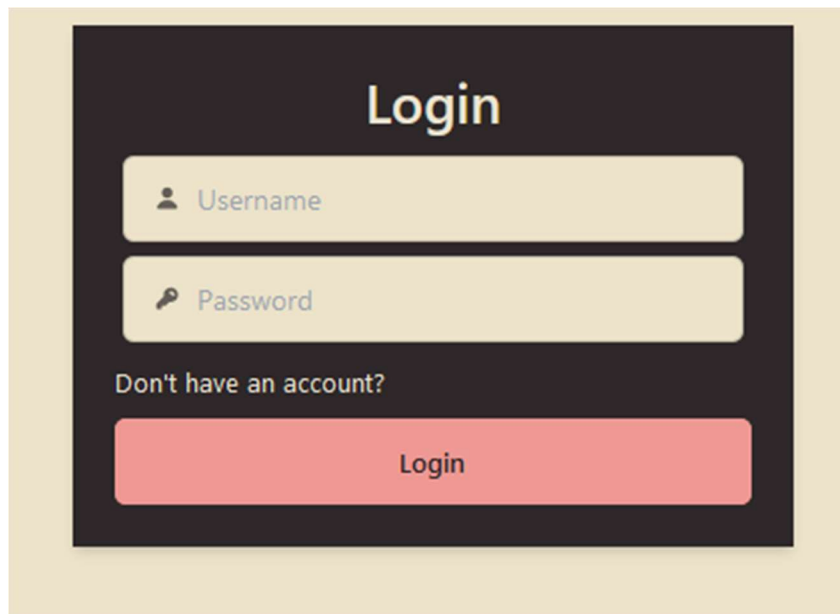
A súgóra kattintva megnyílik egy panel, ahol elolvashatjuk a játék (Azul) szabálykönyvét és az oldal használati útmutatóját is.



### 2. ábra súgó

#### 2.3.2. Bejelentkezési/regisztrációs felület.

Amikor ellátogatnak a weboldalra, a bejelentkezési felületet fogják először meglátni. Itt, ha már van létrehozva felhasználójuk, tudnak tovább lépni az alkalmazás fő oldalára.

A login form with a dark background and light-colored input fields. The title "Login" is centered at the top. Below it are two input fields: "Username" with a person icon and "Password" with a key icon. Below the password field is a link "Don't have an account?". At the bottom is a large red button labeled "Login".

Login

Username

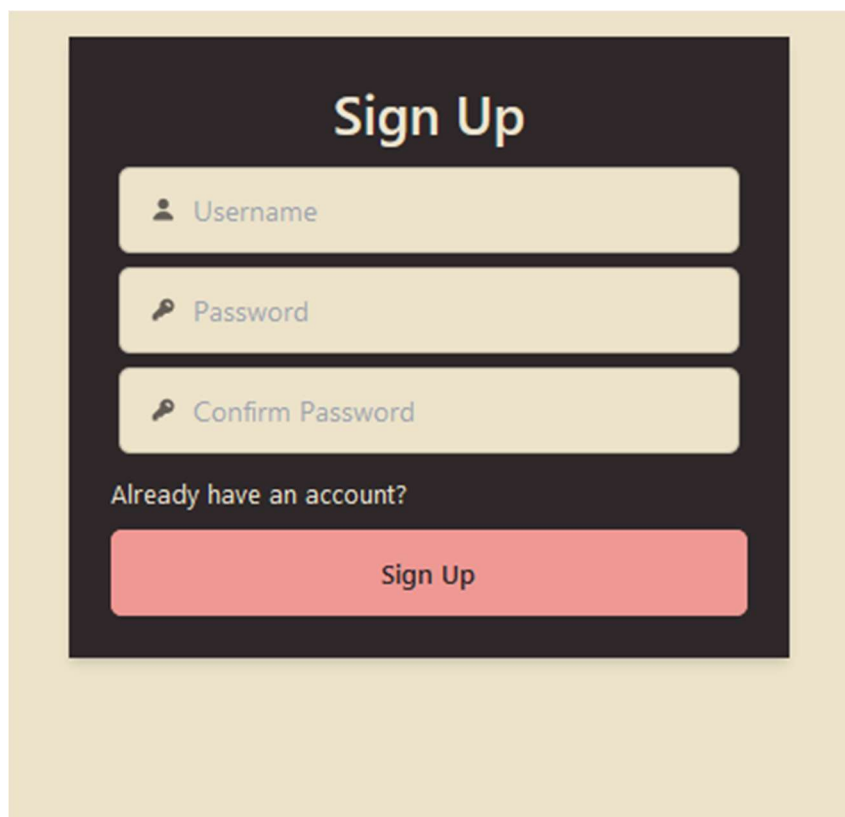
Password

Don't have an account?

Login

3. ábra bejelentkezési felület

Amennyiben nincsen még felhasználójuk, a bejelentkezés gomb fölötti “Don’t have an account?” linkkel tudnak átváltani a regisztrációs felületre.

A sign up form with a dark background and light-colored input fields. The title "Sign Up" is centered at the top. Below it are three input fields: "Username" with a person icon, "Password" with a key icon, and "Confirm Password" with a key icon. Below the password fields is a link "Already have an account?". At the bottom is a large red button labeled "Sign Up".

Sign Up

Username

Password

Confirm Password

Already have an account?

Sign Up

4. ábra regisztrációs felület

A regisztrációhoz szükséges megadni egy egyedi felhasználó nevet, egy legalább 6 karakter hosszú jelszót és a jelszót még egyszer, ezzel megerősítve azt.

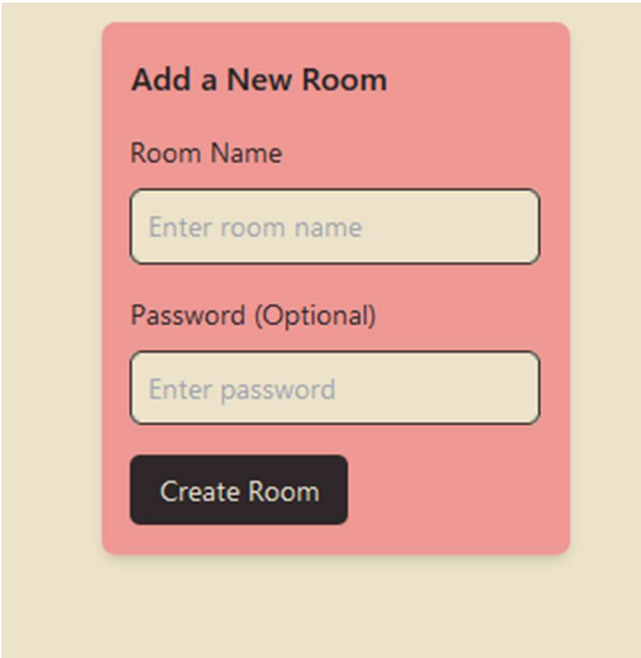
Ha a bejelentkezési adatainkat helyesen kitöltöttük () az alkalmazás a fő oldalra fog minket navigálni.

### 2.3.3. Fő oldal

A főoldal két fő komponensből áll. Egy űrlapból, ami kitöltésével egy új szobát tudunk nyitni és egy, a jelenlegi szobákat tartalmazó listából.

#### Új szoba felvétele

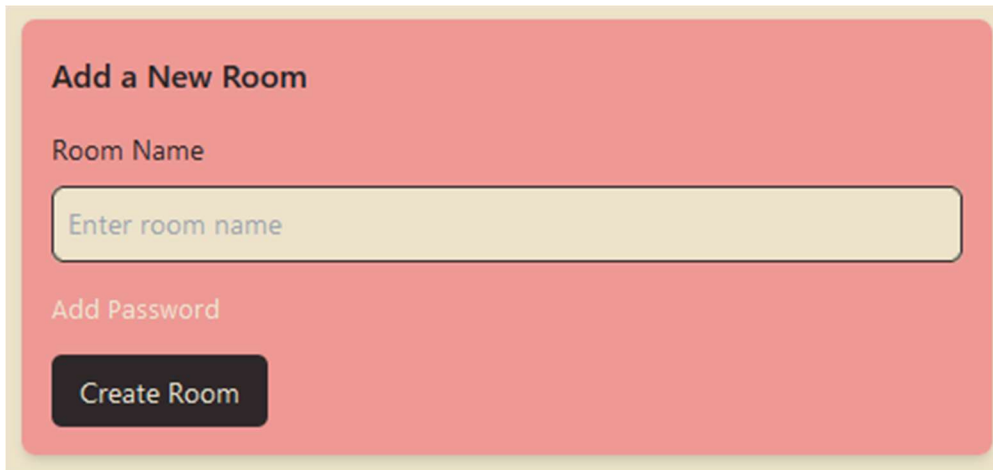
Az űrlapon elegendő csak a szoba nevét megadni, ekkor bárki csatlakozhat. A Jelszó mező kitöltésével korlátozhatjuk a szobánkhoz való hozzáférést.



5. ábra új szoba form

Kisebb kijelző esetén az “Add Password” és “Remove Password” gombokkal tudjuk ki és be kapcsolni a jelszó mező megjelenítését.



A screenshot of a web form titled "Add a New Room". The form has a light red background. It contains a label "Room Name" above a text input field with the placeholder text "Enter room name". Below the input field is a label "Add Password" and a dark red button with the text "Create Room".

**Add a New Room**

Room Name

Enter room name

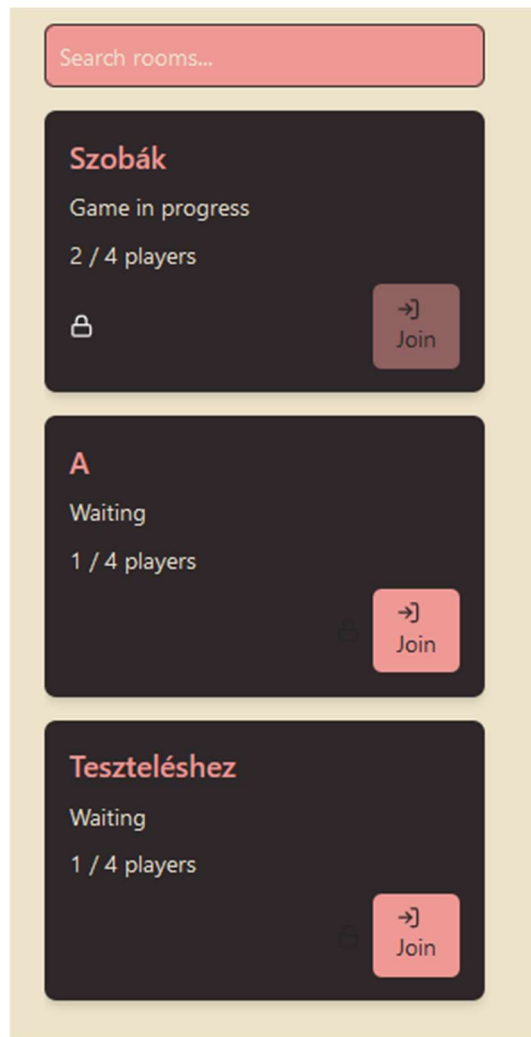
Add Password

Create Room

6. ábra új szoba from kis kijelzőhöz

### **Meglévő szobák**

A job oldali lista tartalmazza a jelenlegi szobákat és egy keresés mezőt, amivel a szobák nevére tudunk szűrni.



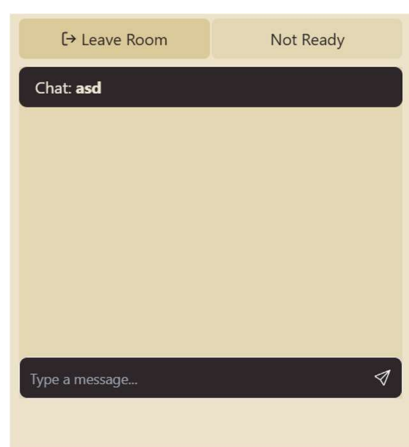
7. ábra szobák listája

Az itt megjelenő kártyák a szobákról tartalmazznak információkat és egy gombot a csatlakozáshoz. A szobáknak megjelenítjük a nevét, jelenlegi állapotát, a csatlakozott felhasználók számát és azt is mutatjuk, hogy szükséges-e jelszó a belépéshez. Amennyiben nem indult még el játék az adott szobában az állapota „Waiting”, ha már megkezdődött a játék, akkor a „Game in progress” feliratot láthatjuk. A szobához csak akkor tudunk csatlakozni, ha még nem indult el a játék és még nem telt be a férőhelyek száma (játékosok száma kevesebb, mint 4). Ha a szobát jelszó védi, azt egy lakat ikonnal jelezzük. Ha erre az ikonra rávisszük a mutatót vagy rákattintunk a csatlakozás gombra, akkor megjelenik a jelszó beviteli mező.

#### 2.3.4. Szoba

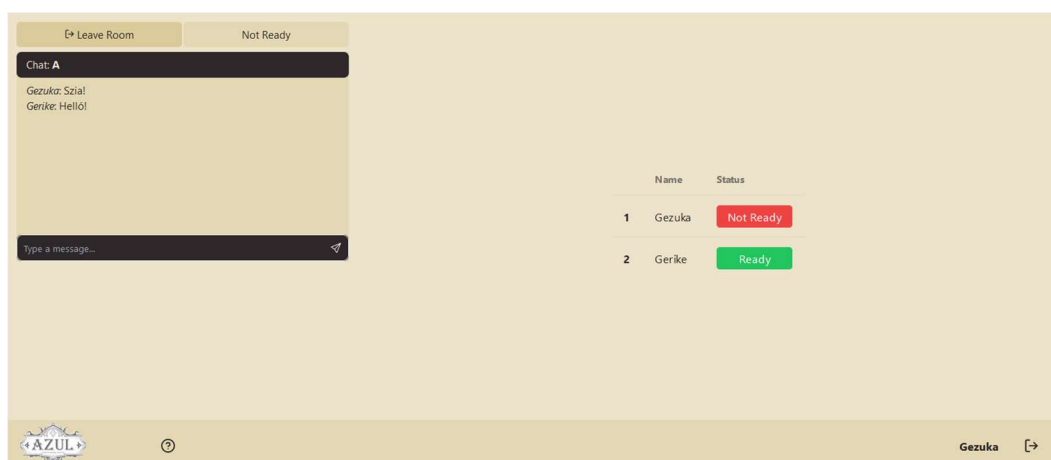
Amikor csatlakozunk egy szobához, akár mi hoztuk létre vagy a listából kiválasztva léptünk be, ugyan arra az oldalra navigál minket az alkalmazás. Ezt a szoba oldalt két részre oszthatjuk. A bal oldali panelre és a játéktérre.

Kezdeti állapotban a bal panel tartalmaz két gombot. Az egyikkel el tudjuk hagyni a szobát, a másik gombbal meg tudjuk jelezni játékosársainknak, hogy készen állunk a játék indítására. A gombok alatt található egy chat ablak, amellyel valós időben tudunk kommunikálni a szoba tagjaival.



8. ábra baloldali panel

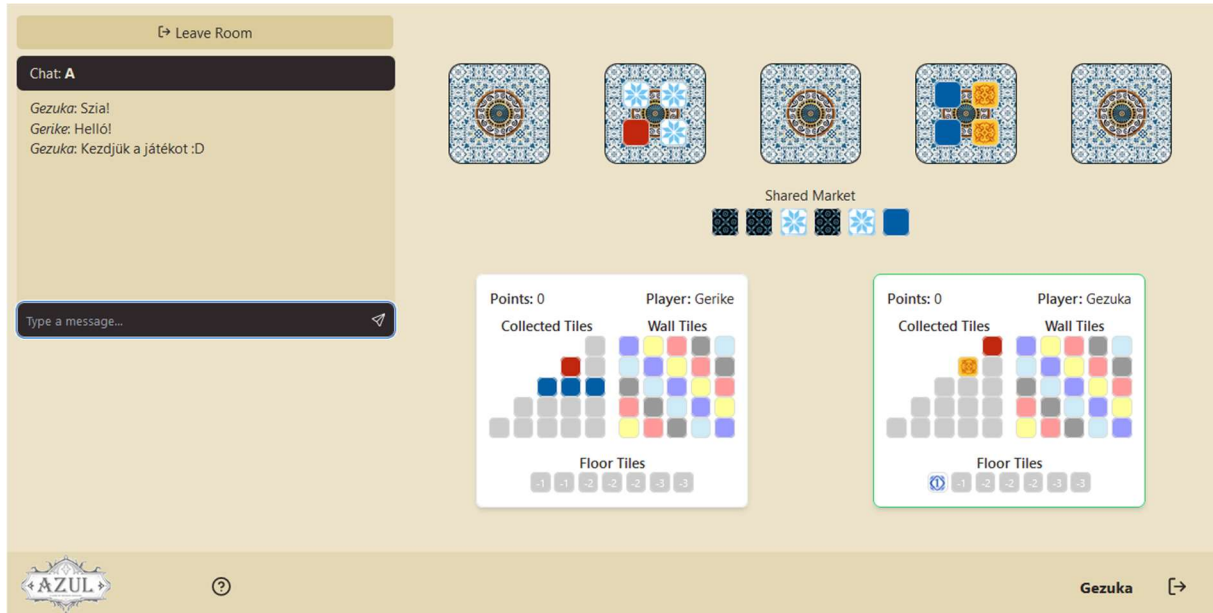
A játék indítása előtt a játéktéren nem a játéktáblát mutatjuk, hanem a szobához csatlakozott felhasználók állapotát.



9. ábra teljes szoba - várakozó

Miután minden játékos „Ready” állapotba került, a játék elindul. A bal oldalról eltűnik a „Ready” gomb, a szoba elhagyása gomb és a chat ablak megmarad.

A játéktéren a státusz panelt leváltja az aktuális játéktábla.



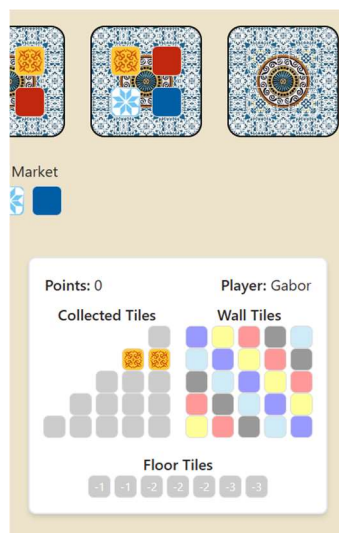
10. ábra teljes szoba - játék

#### 2.3.4.1. Játék

A játék a hagyományos Azul társasjáték szabályait követi. A felhasználók egymás után végzik el a lépéseiket, akcióikat. A soron következő játékos táblája mindig ki lesz emelve, piros színnel, ha az egyik ellenfelünk van soron és zölddel, ha a mi következzük. A saját körünkben választanunk kell egy csempét az egyik piacról (vagy a közös piacról) és a játék táblánk egyik Collected Tiles sorát.

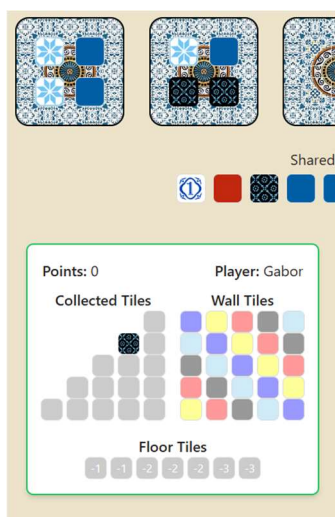


11. ábra csempék kiválasztása

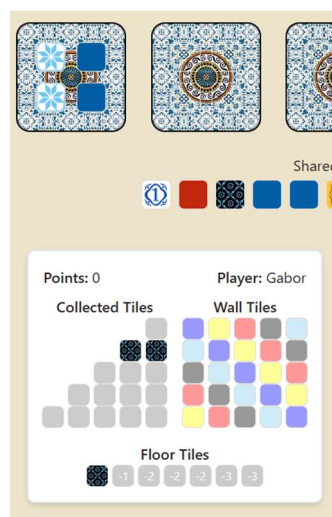


12. ábra csempék lehelyezése

A piacról az összes azonos színű csempét magunkhoz fogjuk venni és a választott sorunkba próbáljuk őket helyezni. Ha a sorban már van csempe, akkor csak vele azonos színűt próbálhatunk a sorba tenni. Ha megtelt a sor, de még nem raktuk le az összes csempénket, akkor a maradék csempe, a földre, a Floor Tiles mezőre fog esni.



13. ábra Floor példa1

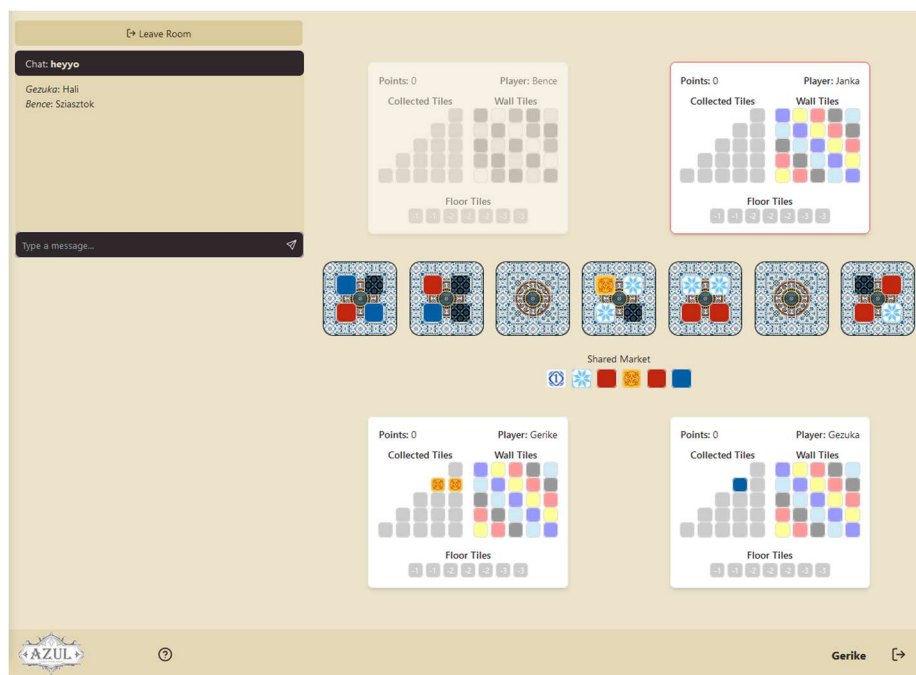


14. ábra floor példa2

A 13. ábrán kiválasztott 2 fekete csempét a 2. sorba helyezzük, a 14. ábra a lépés eredménye.

Csempét, akkor se helyezhetünk egy sorba, ha az ahhoz a sorhoz tartozó fal soron (Wall Tiles) már van a választott színű csempéből.

Ha egy felhasználó elhagyja játék közben a szobát, akkor az ő tábláját kiszürkítjük és a játék állapotból kitörökjük. A játékmenet úgy fog folytatódni, mintha eleve egyel kevesebben kezdtük volna a játékot. A következő kör elején a piacok számát is újra kalibráljuk.



15. ábra játék egy kilépett játékossal

Miután vége a játéknak a játéktér tartalmát ismét kicseréljük. Most egy játék vége táblát mutatunk a játékosok nevével és az elért pontszámukkal.

//a játék akkor is véget ér, ha túl sokan lépnek ki és nem tud folytatódni a játék

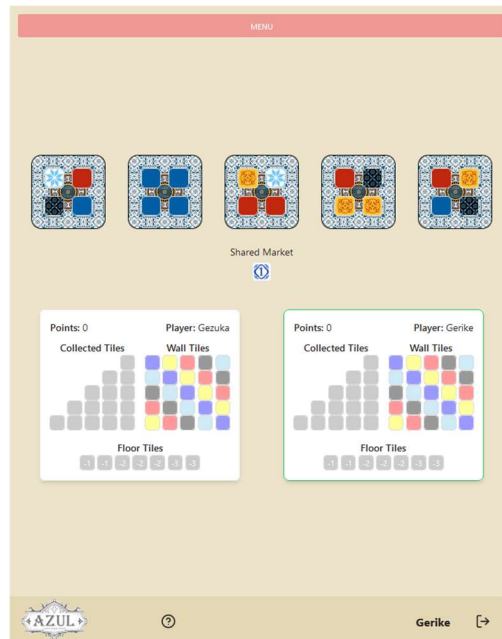
Game Over!		
	Player	Score
1	Gezuka	12
2	Gerike	6
Accept		

16. ábra játék vége panel

Az elfogadásra (“Accept”) gombra kattintva elhagyjuk a játékot és a szobát is. A szoba elhagyásáig a még használhatjuk a közös chat-et.

#### 2.4.5 Kiskijelzős szoba elrendezés

A kis kijelzős elrendezésben a bal oldali komponenseket elrejtjük és az oldal tetején megjelenítünk egy navigációs gombot, mellyel váltogathatunk a játéktér és a bal panel között.

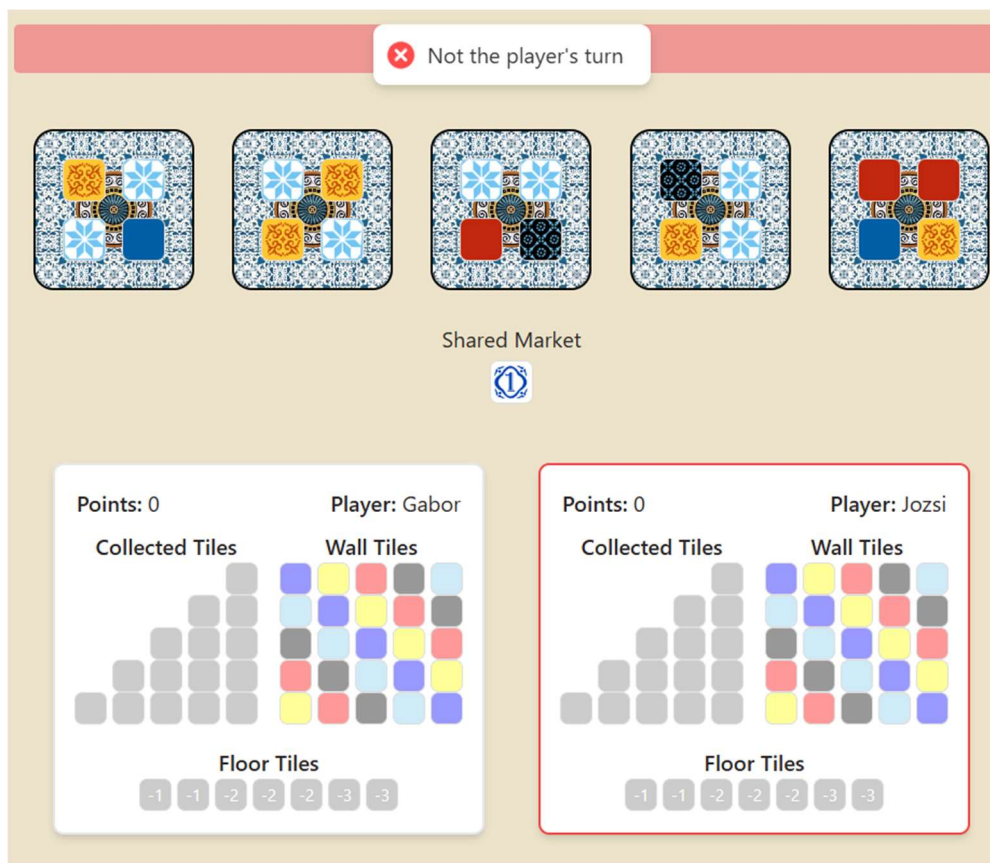


17. ábra kis kijelzős szoba - játék

A játék menet és egyéb funkciók nem változnak.

#### 2.4.6 Figyelmeztetések

Az weboldal használata közben az alkalmazás figyelmeztetéseket küld beúszó buborékok formájában.



18. ábra értesítés - rossz lépésről

Ezek csak információs jellegűek. Röviden, néhány szóban beszámol arról, ha a felhasználó valami hibát követett el. Például: Rossz adatokkal próbált meg bejelentkezni/regisztrálni, nem töltött ki egy fontos beviteli mezőt. Játék közben hibás lépést szeretett volna végrehajtani.



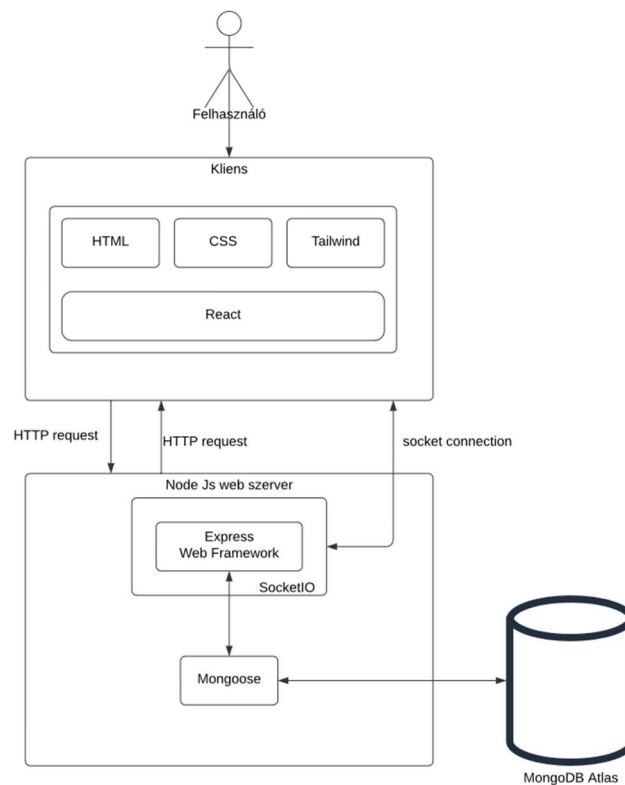
### 3. Fejlesztői dokumentáció

#### 3.1. Technológiai Stack

A szoftver a következő technológiákat használja:

- MongoDB: Dokumentum-orientált NoSQL adatbázis
- Express.js: Minimalista és rugalmas Node.js webalkalmazás keretrendszer
- React: JavaScript könyvtár felhasználói felületek készítéséhez
- Node.js: Szerveroldali JavaScript futtatási környezet
- Socket.io Valós idejű, két irányú kommunikációt tesz lehetővé

#### 3.2. Rendszer



19. ábra rendszer felépítése

A felhasználó a React App-el, a kliensünkkel kommunikál. A frontend és a web szerverünk http kérdés-válasz kommunikációt folytat (bejelentkezés/regisztráció, szobák). A játék és az real-time chat socket kapcsolaton keresztül kommunikál.

#### 3.3. Kezdeti lépések

A projekt létrehozásához a következő lépéseket végeztem el:

- Node.js (<https://nodejs.org/>) a legújabb LTS verzió telepítése
- A Projekt Struktúra kialakítása. Létrehozni a root könyvtárat, amiben dolgozni fogunk. Ebben a könyvtárban npm init-el inicializáljuk a Node környezetünket.
- Npm-mel további függőségek telepítése, mint az express, dotenv, cookie-parser, mongoose, bcryptjs, jsonwebtoken, socket.io
- Adatbázis létrehozása a MongoDB Atlas oldalán (<https://www.mongodb.com/cloud/atlas>)
- .env fájl létrehozása kapcsolat és más környezeti változók tárolására. Alkalmazásunknál elég a MongoDB connection string-jét és a PORT-ot ahonnan futtatjuk a szervert belerakni ebbe a fájlba.
- Npm segítségével Frontend (React) létrehozása Vite-tal.
- Frontendhez a socket.io-client telepítése npm-mel. Illetve a további megjelenítési/frontend dependenciák hozzáadása, mint framer-motion, react-dom, react-router-dom, react-hot-toast, react-icons.
- Stílus elemekhez a tailwindcss és daisyui hozzáadása a projekthez.
- Ezekkel a dependenciák, packagek használatával készítettem el a dolgozatomat.
- A felhasznált technológiákról és azok integrációjáról, összekötéséről a következő fejezetekben olvashatnak.

### 3.4. Környezet felállítása a további fejlesztéshez

- a dolgozat fájljainak letöltése
- Node.js telepítése (legfrissebb LTS verzió)
- MongoDB Atlas beállítása, connection string megkeresése (<https://www.mongodb.com/docs/guides/atlas/connection-string/>)
- dependenciák telepítése a gyökerkönyvtárban „npm install -y”, a frontend mappában „npm install -y”.
- környezeti változók: a gyöker mappában .env fájl létrehozása. A tartalma:  
PORT=<portszám ahonnan fut a szerver>  
MONGO\_URI=<Atlas connection string>  
SECRET=<egy string, jwt beállításához>
- Környezeti változók: a ./frontend mappában .env fájl létrehozása. A tartalma:  
VITE\_APP\_SERVER\_URL=<backend serverünk címe pl: http://localhost:5000>

- server elindítása: a gyökérmappából „npm run server”
- react app elindítása: ./frontend mappából: „npm run dev”

### 3.5. Adatbázis (MongoDB)

A MongoDB egy dokumentum-orientált NoSQL adatbázis, amely JSON-szerű dokumentumokat használ az adatok tárolására. Az alkalmazásomban a MongoDB Atlas-t használtam, amely a MongoDB felhőalapú szolgáltatása. Ez lehetővé teszi az adatbázisok könnyű skálázását, biztonságos tárolását és menedzselését.

#### Adatbázis sémák

Felhasználó:

```
const userSchema = new mongoose.Schema({
  username: {
    type: String,
    required: true,
    unique: true,
  },
  password: {
    type: String,
    required: true,
    minlength: 6,
  },
  roomId: {
    type: mongoose.Schema.Types.ObjectId,
    ref: 'Room',
    default: null,
  },
  status: {
    type: String,
    enum: ['offline', 'online', 'ready', 'playing', 'waiting'],
    default: 'offline',
  },
})
```

20. ábra User séma

Szoba:

```
const roomSchema = new mongoose.Schema({
  name: {
    type: String,
    unique: true,
    required: true,
  },
  owner: {
    type: mongoose.Schema.Types.ObjectId,
    ref: 'User',
    required: true,
  },
  users: {
    type: [mongoose.Schema.Types.ObjectId],
    ref: 'User',
  },
  chat: {
    type: mongoose.Schema.Types.ObjectId,
    ref: 'Chat',
  },
  //game session
  gameId: {
    type: mongoose.Schema.Types.ObjectId,
    ref: 'Game',
    default: null
  },
  hasPassword: {
    type: Boolean,
    default: false
  },
  password: {
    type: String,
    default: null
  }
},{ timestamps: true })
```

21. ábra Room shéma

Üzenethez és a chat-hez

```
import mongoose from "mongoose";

const messageSchema = new mongoose.Schema({
  senderName: {
    type: String,
    required: true
  },
  senderId: {
    type: mongoose.Schema.ObjectId,
    ref: 'User',
    required: true
  },
  receiverId: {
    type: mongoose.Schema.ObjectId,
    ref: 'Room',
    required: true
  },
  message: {
    type: String,
    required: true
  }
},{timestamps: true})

const chatSchema = new mongoose.Schema({
  roomId: {
    type: mongoose.Schema.Types.ObjectId,
    ref: "Room"
  },
  messages: [
    {
      type: mongoose.Schema.Types.ObjectId,
      ref: "Message",
      default: []
    }
  ]
})
```

23. ábra Chat séma

22. ábra Message séma

## Játék és játékos tábla

```
const gameSchema = new mongoose.Schema({
  roomId: {
    type: mongoose.Schema.Types.ObjectId,
    ref: 'Room',
    required: true
  },
  players: {
    type: [mongoose.Schema.Types.ObjectId],
    ref: 'User',
  },
  playerToMove: {
    type: mongoose.Schema.Types.ObjectId,
    ref: 'User',
    default: null
  },
  bag: {
    type: [String],
    enum: ['red', 'black', 'blue', 'azure', 'yellow', 'empty', 'white'],
    default: []
  },
  markets: {
    type: [[String]],
    enum: ['red', 'black', 'blue', 'azure', 'yellow', 'empty', 'white'],
    default: []
  },
  sharedMarket: {
    type: [String],
    enum: ['red', 'black', 'blue', 'azure', 'yellow', 'empty', 'white'],
    default: ['white']
  },
  playerBoards: {
    type: [mongoose.Schema.Types.ObjectId],
    ref: 'PlayerBoard',
    default: []
  },
  gameStatus: {
    type: String,
    enum: ['waiting', 'started', 'playing', 'ended'],
    default: 'waiting'
  }
});
```

24. ábra Game séma

```
const playerBoardSchema = new mongoose.Schema({
  playerId: {
    type: mongoose.Schema.Types.ObjectId,
    ref: 'User',
    required: true
  },
  points: {
    type: Number,
    default: 0
  },
  collectedTiles: {
    type: [[String]],
    enum: ['red', 'black', 'blue', 'azure', 'yellow', 'empty', 'white'],
    default: [
      ['empty'],
      ['empty', 'empty'],
      ['empty', 'empty', 'empty'],
      ['empty', 'empty', 'empty', 'empty'],
      ['empty', 'empty', 'empty', 'empty', 'empty']
    ]
  },
  wallTiles: {
    type: [[String]], // Represents a 5x5 grid of tiles
    enum: ['red', 'black', 'blue', 'azure', 'yellow', 'empty', 'white'],
    default: [
      ['empty', 'empty', 'empty', 'empty', 'empty'],
      ['empty', 'empty', 'empty', 'empty', 'empty'],
      ['empty', 'empty', 'empty', 'empty', 'empty'],
      ['empty', 'empty', 'empty', 'empty', 'empty'],
      ['empty', 'empty', 'empty', 'empty', 'empty']
    ]
  },
  floorTiles: {
    type: [String],
    enum: ['red', 'black', 'blue', 'azure', 'yellow', 'empty', 'white'],
    default: ['empty', 'empty', 'empty', 'empty', 'empty', 'empty', 'empty', 'empty'],
  }
});
```

25. ábra PlayerBoard séma

## 3.6. Socket.IO Integráció

A Socket.IO lehetővé teszi a valós idejű, két irányú kommunikációt a szerver és a kliens között. Az alkalmazásban a Socket.IO-t használtuk a valós idejű funkciók megvalósításához, mint például a chat, valós idejű értesítések és a játék eseményeinek küldésére.

```

import { Server } from 'socket.io';
import http from 'http';
import express from 'express';

import { messageHandler } from './socketHandlers/messageHandler.js';
import { gameHandler } from './socketHandlers/gameHandler.js';

const app = express();

const server = http.createServer(app);
const io = new Server(server, {
  cors: {
    origin: ["http://localhost:3000"],
    methods: ["GET", "POST"],
  }
});

export const getReceiverSocketId = (receiverId) => {
  return userSocketMap[receiverId];
};

const userSocketMap = {}; // {userId: socketId}

io.on('connection', (socket) => {
  console.log("a user connected", socket.id);

  const userId = socket.handshake.query.userId;
  if (userId != "undefined") {
    userSocketMap[userId] = socket.id;
  }

  messageHandler(socket);
  gameHandler(socket);

  socket.on('disconnect', () => {
    delete userSocketMap[userId];
    console.log('user disconnected', socket.id);
  });
});

export {app, io, server}

```

26. ábra socket.js

A socket eseményekről és azok kezeléséről a Backend és Frontend fejezetekben fogunk részletesebben foglalkozni.

### 3.7. Backend (Express.js)

#### Technológia

Az Express.js egy minimalista és rugalmas Node.js webalkalmazás keretrendszer, amely számos hasznos funkciót kínál a web- és mobilalkalmazások fejlesztéséhez.

#### 3.7.1. Felépítés

Az alkalmazás szerveroldali kódja az Express.js keretrendszert használja. Az Express app-ot, miután integráltuk a Socket.IO-val, a szerverünkbe importáljuk a különböző útvonalainkkal (route-ok a http kérésekhez) összekötjük, majd a szerver indításakor a MongoDB adatbázishoz is csatlakozunk a connectToMongoDB függvény segítségével, és a szerver futását a megadott porton indítjuk el.

```

import express from "express"
import dotenv from "dotenv"
import cookieParser from "cookie-parser"
import authRoutes from "../routes/authRoutes.js"
import roomRoutes from "../routes/roomRoutes.js"

import connectToMongoDB from "../db/connectToMongoDB.js";
import { app, server } from "../socket/socket.js"

const PORT = process.env.PORT || 5000

dotenv.config()
app.use(express.json())
app.use(cookieParser())

app.use("/api/auth", authRoutes)
app.use("/api/rooms", roomRoutes)

server.listen(PORT, () => {
  connectToMongoDB()
  console.log(`Server running on port ${PORT}`)
})

```

27. ábra server.js

Az Express alkalmazáshoz hozzáadjuk a szükséges middleware-eket: `express.json()` a JSON kérés/test formátum támogatására és `cookieParser()` a cookie-k kezeléséhez.

### 3.7.2. Szerver-kliens kommunikáció

#### Routes

Authentikációs és szoba műveleti kérések HTTP kérések formájában érkeznek a szerverre. Az „/api/auth” útvonalra érkező kéréseket az `authRoutes` irányítja tovább, míg az „/api/rooms” útvonalat a `roomRoutes` kezeli.

Az autentikációs útvonalakhoz az `authController` biztosít metódusokat a kérés megfelelő kezeléséhez.

```

import express from "express";
import {signup,login,logout, setReady} from "../controllers/authController.js"

const router = express.Router();

router.post("/signup",signup);

router.post("/login",login);

router.post("/logout",logout);

router.post("/ready/:id", setReady)

export default router;

```

28. ábra User útvonalak

A roomRoutes útvonalait a roomController-ben kezeljük.

## SocketHandlers

Az alkalmazás azon részei, amelyek valós idejű kommunikációt igényelnek és vagy biztosan több felhasználóval kell kommunikálniuk Socket.IO kapcsolaton keresztül teszik meg azt. A socketHandler-eket a socket integráció alatt kapcsoljuk az apponkhoz.

gameHandler:

A játék irányításához szükséges eseményeket figyeli. Ezeket az eseményeket a gameController metódusaival kezeli le a szerver.

```
import { setupGame, getGame, takeTiles } from "../../controllers/gameController.js";

export const gameHandler = (socket) => {
  socket.on("SetupGame", async (data) => {
    setupGame(socket, data);
  });

  socket.on("GetGame", async (data) => {
    getGame(socket, data);
  });

  socket.on("TakeTiles", async (data) => {
    takeTiles(socket, data);
  });
}
```

29. ábra socketHandler a játékhoz

messageController:

a messageControllert használja a sendMessage és getMessage események lereagálásához.

### 3.7.3. Kontrollerek

Az alkalmazás logikáját tartalmazzák. A megfelelő kezelőkön érkezett adatokat, eseményeket dolgozzák fel, mentik az adatbázisba és továbbítják a frontend felé.

**authController:**

A felhasználóval és adataival dolgozik. Biztosít signup, login, logout és setReady metódusokat a regisztrációhoz, bejelentkezéshez, kijelentkezéshez és a ready státusz megváltoztatásához.



HTTP kéréseket kezelünk le, a kéréshez tartozó adatokat a req (request)ben kapjuk meg. A válaszként a res(respons)ban küldjük vissza, res.status-ban jelezve a művelet sikerességét.

- signup: kapott értékeket ellenőrzi, ha még nincs ilyen felhasználó, akkor csinál egy újat. A jelszót bcrypt segítségével hash-eljük és úgy tároljuk el.
- login: a kapott adatokkal megkeres egy felhasználót és bcrypt segítségével ellenőrzi a jelszót.
- logout: kiléptetjük a kérést küldő felhasználót. Ha a user szobában van, akkor a szobából is ki kell vegyük, illetve, ha szobában van és megy a játék, akkor a játékból is.

#### **roomController:**

A szobákra hajt végre módosításokat. A kérések HTTP requestek formájában érkeznek, de a válaszokat, amikor több felhasználóhoz is el kell juttatni, a http válasz mellett Socket.IO eseménnyel is visszaküldi a kliens felé. Metódusai:

- getRooms, visszaküldi a jelenleg hosztolt szobákat. getRoom, az útvonal paraméterében kapott :id alapján keres és visszaküld egy szobát.
- createRoom, létrehoz egy szobát és értesíti róla a klienseket.
- leaveRoom, a küldőt, ha jelenleg szobában van, kilépteti, majd ellenőrzi a szoba tagjainak számát. Ha az nulla, akkor törli a szobát (ha a user játékban volt, akkor kilépteti onnan is).
- deleteRoom: clearUsers-el kiléptet mindenkit a szobából majd kitörli azt.
- senetizeRoom, segéd fv, a szobáról csinál egy jelszó mentes másolatot, amit biztonságosan tudunk elküldeni.

#### **messageController:**

Csak socket kommunikáció van.

- sendMessage, egy új üzenetet létrehoz az eseményen keresztül kapott adatokból és a felhasználó szobájának címezve elküldi azt.
- getMessages, az eseményen keresztül megkap egy szoba azonosítót, ami alapján megkeres egy chatet és annak összes üzenetét visszaküldi.

#### **gameController:**

A társasjáték logikáját tartalmazza. Socket kapcsolaton keresztül kommunikál a klienssel.

- `setupGame`, egy `roomId`-t kap, amivel keres egy szobát. Ha talált szobát megnézi, hogy van-e hozzátartozó játék. Ha a játék létezik, visszaküldjük azt a szoba tagjainak. Ha nem létezik még játék, akkor újat kezdünk és ezt az új játékot küldjük el a szoba tagjainak(`NewGame`), majd elkezdünk egy új kört.
- `startNewRound`, csak a controllerből érhető el. feltölti a piacokat és a közös piacot. majd ezt a játékállapotot küldi el a szoba felhasználóinak (`NewRound`)
- `takeTiles`, kap egy csempét(színt), `marketId`-t, `row`-t (sorindex) és a socket handshakeból a `userId`-t. Ezekből meg tudjuk határozni, hogy melyik játékos, melyik piacról, melyik csempét vette el. Az elvett csempékkel feltöltjük a játékos táblájának a `collectedTiles[row]` sorát. Ha szükséges, a `floorTiles`-t is. `TakeTiles` eseményt küldünk, majd megnézzük, hogy vége van-e a körnek.
- `onRoundOver`, csak a controllerből érhető el. Feladata a `PlayerBoard`-okon a `collectedTiles`-ból a `wallTiles`-ba vinni a megfelelő csempétet és ez alapján pontozni a játékosokat. pontozás után megnézzük, hogy vége van-e a játéknak. Ha igen, akkor `GameOver` eseménnyel küldjük el a játékállást a szoba tagjainak. Ha nem `isGameOver`, akkor a `RoundOver` eseménnyel. Ezután (ha nincs vége a játéknak) `startNewRound` függvényhívása.
- `getGame`, egy lekérdezés amely a kapott `roomId` alapján visszaküld egy játék állapotot (vagy null értéket). `GetGame` eseménnyel küldjük vissza.
- `leaveCurrentGame`, `userId` paraméterrel keres egy játékot, amelynek az adott id user a tagja, majd kivesszük a `players` listából. Ha több, mint 1 játékos marad, akkor folytatjuk a játékot, `PlayerLeftGame` eseményt küldünk a játékállásról. Ha 1 vagy 0 játékos maradt a játékban, akkor `GameOver` eseménnyel lépünk ki.

Segéd fv-ek: `newBag`, `newMarkets`, `newPlayerBoards`, `getRandomTileFromBag`, `loadMarkets`, `calculatePoints`, `isGameOver`, `isRoundOver`, `isValidMove`.

`newBag`, `newMarkets`, `newPlayerBoards` felvesz megfelelő mennyiségű üres komponenst a játék állapotba. A `loadMarkets` a `getRandomTileFromBag` segítségével feltölti a piacokat csempékkel. Az `isGameOver`, `isRoundOver` és `isValidMove` a játékmenet közben ellenőrzi a

játék állapotát, lépés helyességét. A calculatePoints az onRoundOver-ben kiszámolja minden csempe falrahelyezése közben, hogy hány pontot ért.

### 3.8. Frontend (React)

#### Technológia

A React egy JavaScript könyvtár felhasználói felületek készítéséhez. A React lehetővé teszi a komponens alapú fejlesztést, ami egyszerűvé és átláthatóvá teszi a kódot.

#### Felépítés

##### Context-ek

Context-ek adatközlésre szolgálnak a ReactDOM fában. Használatával elkerülhetjük a szülő-gyerek elemek közti property alapú adatátadást.

A dolgozatban az App komponens köré vettem fel a kontextusaimat, így azokat és adatait/funkcióit az összes komponensből elérjük.

```
import React from 'react'
import ReactDOM from 'react-dom/client'
import App from './App.jsx'
import './index.css'
import {BrowserRouter} from "react-router-dom"
import { AuthContextProvider } from './context/AuthContext.jsx'
import { RoomContextProvider } from './context/RoomContext.jsx'
import { ChatContextProvider } from './context/ChatContext.jsx'
import { SocketContextProvider } from './context/SocketContext.jsx'
import { GameContextProvider } from './context/GameContext.jsx'

ReactDOM.createRoot(document.getElementById('root')).render(
  <React.StrictMode>
    <BrowserRouter>
      <AuthContextProvider>
        <SocketContextProvider>
          <RoomContextProvider>
            <ChatContextProvider>
              <GameContextProvider>
                <App />
              </GameContextProvider>
            </ChatContextProvider>
          </RoomContextProvider>
        </SocketContextProvider>
      </AuthContextProvider>
    </BrowserRouter>
  </React.StrictMode>,
)
```

30. ábra main.jsx

Az ábrán látható main.jsx a React alkalmazásunk belépési pontja. Az index.html body tag-jében szereplő <div id="root"></div> elemben hozza létre az App.jsx komponensünket. A

main.jsx-ben található ContextProvider-ek felelnek a gyerekelemek felé történő adatátadásért.

### AuthContext:

A jelenleg bejelentkezett felhasználót tárolja el egy useState hookban és a localStorage-ben. Biztosít egy useAuthContext funkciót, amivel megtudjuk hívni a kontextust és a visszatérési értéke a provider melynek value paraméterében tovább adjuk az authUser változót és a setAuthUser setter metódust.

```
import { createContext, useState, useContext, useEffect } from "react";

export const AuthContext = createContext();

export const useAuthContext = () => {
  return useContext(AuthContext)
}

export const AuthContextProvider = ({children}) => {
  const [authUser, setAuthUser] = useState(JSON.parse(localStorage.getItem("authUser")) || null)

  useEffect(() => {
    localStorage.setItem("authUser", JSON.stringify(authUser));
  }, [authUser]);

  return <AuthContext.Provider value={{authUser, setAuthUser}}>
    {children}
  </AuthContext.Provider>
}
```

31. ábra AuthContext

### SocketContext:

a socket.io-clientből importált io segítségével létesít socket kapcsolatot a szerverrel. Majd ehhez a socket-hez biztosít hozzáférést az App-ban.

```
useEffect(() => {
  if(authUser){
    const socket = io(import.meta.env.VITE_APP_SERVER_URL, {
      query: {
        userId: authUser._id,
      },
    });
    setSocket(socket);
    return () => {socket.close()};
  }else{
    if(socket){
      socket.close();
      setSocket(null);
    }
  }
},[authUser]);
```

32. ábra SocketContext részlet

### **RoomContext:**

A jelenleg aktív szobákat tároljuk benne. Biztosít egy rooms változót és egy setRooms setter metódust. `const [rooms, setRooms] = useState([]);`

### **ChatContext:**

Ennek a kontextusnak (Provider-nek) a használatával elérhetővé tesszük a messages állapot változót és a setMessages setter metódusát.

### **GameContext:**

A játékállapotot tároljuk benne egy gameState állapot változóban és biztosít számunkra egy setGameState setter metódust.

## **Saját hookok**

### **useSignup:**

Regisztrációhoz használjuk. Biztosít számunkra egy signup függvényt, ami egy http kérést küld a szervernek. Sikeres válasznál beállítjuk az authContext authUser-ét.

```

const useSignup = () => {
  const [loading, setLoading] = useState(false)
  const {setAuthUser} = useAuthContext()

  const signup = async ({username, password, confirmPassword}) => {
    const success = handleInputErrors({username, password, confirmPassword})
    if(!success) return;

    setLoading(true)
    try {
      const res = await fetch("/api/auth/signup",{
        method: "POST",
        headers: {"Content-Type": "application/json"},
        body: JSON.stringify({username, password, confirmPassword})
      })

      const data = await res.json()
      if(data.error){
        throw new Error(data.error)
      }

      localStorage.setItem("authUser", JSON.stringify(data))
      setAuthUser(data)
    } catch (error) {
      toast.error(error.message)
    } finally{
      setLoading(false)
    }
  }

  return{loading, signup}
}
export default useSignup

```

33. ábra useSignup hook

### useLogin:

Bejelentkezéshez használjuk. Biztosít login metódust, ami http kérést küld a szervernek és a válasz alapján beállítja az authUser-t.

### useLogout:

Biztosít egy logout metódust, ami egy http kérést küld a szervernek és beállítja az authUser-t null értékre, majd törli a localStorage-ből.

### useJoinRoom:

A következő metódusokat tartalmazza: joinRoom, leaveRoom, getRooms. Ezekkel http kéréseket küldünk a szervernek a szobák kezelésével kapcsolatban. joinRoom-mal csatlakozunk egy meglévő szobához. leaveRoom-mal kilépünk a jelenlegi szobából. A getRooms-szal lekérdezhethetjük az összes aktív szobát. Ha sikeres értékkel tér vissza egy request, akkor frissíti az authUser-t és a rooms-ot (roomContext-ből) is.

## useListenRoom:

A szobákhoz tartozó socket eventeket figyeljük rajta. Ezek az eventek a következők. newRoom, hozzáadja az eseménnyel érkező új szobát a rooms értékéhez. deleteRoom, kitörli az eventtel érkező szobát a rooms-ból. updateRoom, egy rooms-beli szobát frissít. getRooms az egész rooms értéket felülírja az eventből kapott új szobákkal.

```
const useListenRooms = () => {
  const { socket } = useSocketContext();
  const { setRooms, rooms } = useRoomContext();
  const { authUser, setAuthUser } = useAuthContext();
  const { setGameState } = useGameContext();

  useEffect(() => {
    socket?.on("newRoom", (room) => {
      setRooms([room, ...rooms]);
    });
    socket?.on("deleteRoom", (room) => {
      setRooms(rooms.filter((existingRoom) => existingRoom._id !== room._id));
      if (authUser.roomId === room._id) {
        setAuthUser({ ...authUser, roomId: null });
        setGameState(null);
      }
    });
    socket?.on("updateRoom", (room) => {
      setRooms(rooms.map((existingRoom) => {
        if (existingRoom._id === room._id) {
          return room;
        } else {
          return existingRoom;
        }
      }));
    });
    socket?.on("getRooms", allRooms => {
      setRooms(allRooms);
    });
    return () => {
      socket?.off("newRoom");
      socket?.off("deleteRoom");
      socket?.off("updateRoom");
      socket?.off("getRooms");
    };
  }, [socket, rooms, setRooms]);
};
export default useListenRooms;
```

34. ábra useListenRooms

## useGame:

Socket eseményeket küld a szervernek.

```

const useGame = () => {
  const { socket } = useSocketContext();

  const setupGame = async (roomId) => {
    socket.emit("SetupGame", { roomId });
  };

  const takeTiles = async (tile, marketId, row) => {
    socket.emit("TakeTiles", { tile, marketId, row });
  };

  const getGame = async (roomId) => {
    socket.emit("GetGame", { roomId });
  };

  return { setupGame, takeTiles, getGame };
}

```

35. ábra useGame

setupGame, takeTiles, getGame metódusokat biztosít a számunkra. setupGame-mel megpróbálunk egy új játékot kezdeni. A takeTiles metódussal irányítjuk a játékot. A getGame lekérdezi a jelenlegi játékállást.

#### **useListenGame:**

A szerver felől érkező, a játékhoz kapcsolódó socket eseményeket kezeli. Ezekkel az eseményekkel frissítjük a gameState megfelelő adattagjait, illetve bizonyos esetekben meganimáljuk a játék változásait. Hogy biztosítsuk, a megfelelő sorrendű animációt és követhető legyen a játék változása egy sorba tesszük a beérkező eseményeket és egyesével kezeljük le őket.



```

const processEvent = () => {
  if (eventQueue.length > 0) {
    const { type, data } = eventQueue[0];
    switch (type) { ...
    }
    setTimeout(() => {
      setEventQueue((prevQueue) => prevQueue.slice(1));
    }, 500);
  }
};

useEffect(() => {
  processEvent();
}, [eventQueue]);

useEffect(() => {
  const handleEvent = (type, data) => {
    console.log(`Event received: ${type}`, data);
    setEventQueue((prevQueue) => [...prevQueue, { type, data }]);
  };

  socket?.on("NewGame", (game) => handleEvent("NewGame", game));
  socket?.on("GetGame", (game) => handleEvent("GetGame", game));
  socket?.on("UpdateGame", (game) => handleEvent("UpdateGame", game));
  socket?.on("NewRound", (game) => handleEvent("NewRound", game));
  socket?.on("GameOver", (game) => handleEvent("GameOver", game));
  socket?.on("RoundOver", (game) => handleEvent("RoundOver", game));
  socket?.on("TakeTiles", (data) => handleEvent("TakeTiles", data));
  socket?.on("PlayerLeftGame", (data) => handleEvent("PlayerLeftGame", data));

```

36. ábra useListenGame eventQueue

Ezek az események a backend/cotrollers/gameController-éből érkeznek.

### useListenMessages:

A chat funkcionalitáshoz szükséges. Új üzeneteket fogad, melyek a newMessage socket eseménnyel érkeznek. Amikor szobát váltunk vagy frissül a socket (socketContextből) egy socket eseménnyel, a getMesseg-es-el lekérdezzük a szobához tartozó jelenlegi üzeneteket. Ugyan ezen, a getMessages néven kapjuk vissza a szoba összes eddigi üzenetét. Továbbá ez a hook biztosít egy sendMessage metódust új üzenet küldésére.

```

const useListenMessages = () => {
  const { socket } = useSocketContext();
  const { setMessages, messages } = useChatContext();
  const { authUser } = useAuthContext();

  useEffect(() => {
    if(authUser.roomId){
      socket?.emit("getMessages", {roomId: authUser.roomId});
    }
  }, [socket, authUser.roomId]);

  useEffect(() => {
    socket?.on("newMessage", (message) => {
      setMessages([...messages, message]);
    });

    socket?.on("getMessages", (messages) => {
      setMessages(messages);
    });

    return () => {
      socket?.off("newMessage");
      socket?.off("getMessages");
    };
  }, [socket, messages, setMessages, authUser.roomId]);

  const sendMessage = (message) => {
    console.log("sendMessage: ", message);
    if(!message) return;
    if(!authUser.roomId) return;
    socket?.emit("sendMessage", { receiverId: authUser.roomId, message});
  };

  return {sendMessage, messages};
}

```

37. ábra useListenMessages

## App

A React alkalmazás React Router DOM-ot használ a kliens oldali útvonalkezeléshez, lehetővé téve az egyoldalas alkalmazás (SPA) élményt.

```

function App() {
  const { authUser } = useAuthContext();
  useListenErrors();
  useRedirect();

  return (
    <div className='h-screen flex flex-col'>
      <div className='flex-grow h-full overflow-hidden p-4 pb-0'>
        <Routes>
          <Route path="/" element={authUser && authUser.roomId ? <Navigate to={`/${session.roomId}` } /> :
            (authUser ? <Home /> : <Navigate to="/login" />) />
          </Route>
          <Route path="/login" element={authUser ? <Navigate to="/" /> : <Login />} />
          <Route path="/signup" element={authUser ? <Navigate to="/" /> : <Signup />} />
          <Route path={`/${session.roomId}`} element={authUser ? <Session /> : <Navigate to="/" />} />
        </Routes>
        <Toaster />
      </div>
      <div>
        <Navbar/>
      </div>
    </div>
  );
}

```

38. ábra App

## **További frontend komponenseink**

### **Signup oldal**

Egy űrlapot ábrázol mellyel megadható a felhasználó neve, jelszava és a jelszava még egyszer. Ezekkel az adatokkal indítja meg a regisztrációs folyamatot. Ha sikeres átirányít a Home oldalra.

### **Login oldal**

Egy űrlapot ábrázol felhasználónévvel és jelszóval. Ezekkel az adatokkal megpróbál bejelentkezni, majd a Home oldalra irányít.

### **Home page**

A RoomForm és HomeSideBar-t jeleníti meg.

Feltölti a roomsContext-et a useJoinRoom::getRooms metódussal és a HomeSideBar-nak paraméterként odaadja a rooms-ot.

### **RoomForm**

Egy űrlap szoba készítéshez. Az új szobára automatikusan meghívja a useJoinRoom::joinRoom metódusát.

### **HomeSideBar:**

Megjelenít egy keresési inputot és a jelenlegi szobákat kártya formában. A kapott rooms propból minden szobára csinál egy RoomCard-ot. A keresési input leszűri a RoomCard-ok listáját név alapján.

### **RoomCard:**

Propertynek kap egy szobát és kártya formában jeleníti meg. Ha a szobának van jelszava, akkor ad a kártyához egy jelszó beviteli mezőt. A csatlakozás gomb onClick eseményéhez hozzárendeli a useJoinRoom::joinRoom metódusát.

### **Session oldal:**

Tartalmazza a ChatBox komponenst, gombokat a játékos státusz váltásához és a szoba elhagyásához. Illetve megjeleníti a játékteret, ami a jelenlegi játék állapot és isGameOver state

változó függvényében a következő lehet: StatusBoard, GameOverPanel vagy maga a játék tábla. A játéktábla PlayerBoardCard-okból, Market-ekből és SharedMarketből áll.

A szoba elhagyását a useJoinRoom::leaveRoom metódusa végzi a „leave button” onClick eseményén. A játékos státusz váltását a handleToggleReady végzi.

```
const handleToggleReady = async () => {
  const url = `/api/auth/ready/${authUser._id}`;

  let newStatus = playerReady ? "waiting" : "ready";
  const res = await fetch(url, {
    method: "POST",
    headers: {
      "Content-Type": "application/json",
    },
    body: JSON.stringify({ status: newStatus }),
  });

  const data = await res.json();
  console.log("handleReady data: ", data);
  if (res.ok) {
    setPlayerReady((prevState) => !prevState);
  } else {
    console.error("Error toggling ready status: ", data.error);
  }
};
```

39. ábra handleToggleReady

Ez a függvény módosítja a Session komponens PlayerReady state változóját.

Az alábbi useEffect-tel figyeljük, hogy el tudjuk-e indítani a játékot:

```
useEffect(() => {
  console.log("canStartGame: ", canStartGame());
  if (canStartGame()) {
    handleSetupGame();
  }
}, [room?.users, playerReady]);
```

40. ábra start game useEffect

a canStartGame igazzal tér vissza, ha megfelelő számú felhasználó van a szobában és mindenkinek „ready” státusza van. a handleSetupGame a useGame::setupGame segítségével indítja a játékot.

A StatusBoard-ot csak akkor mutatjuk, ha még nem indult el a játék és a gameOver state változó hamis.

```
{!gameState && !gameOver && (
  <StatusBoard />
```

A játéktábla elemeit a gameState-ből (GameContext) töltjük fel, illetve a játék irányításához függvényeket adunk át property-ként a játék komponenseknek. A Market és SharedMarket az handleTileClicked metódust kapják meg, amivel a selectedTile és selectedMarket state változóknak adnak értéket.

```
const handleTileClick = (tile, marketId) => {
  console.log(`Tile ${tile} clicked in market ${marketId}`);
  setSelectedTile(tile);
  setSelectedMarket(marketId);
  if (selectedRow !== null) {
    handleTakeTiles(tile, marketId, selectedRow);
  }
};
```

41. ábra csempe kiválasztása

```
{gameState.markets.map((market, index) => (
  <div className="inline-block m-1" key={index}>
    <Market
      marketId={index}
      tiles={market}
      onTileClick={handleTileClick}
      selectedTile={selectedTile}
      selectedMarket={selectedMarket}
    />
  </div>
))}
```

42. ábra csempe választás átadása

A PlayerBoardCard megkapja a handleRowClick metódust, mellyel a selectedRow state változó kap értéket.

Miután ez a három változó kapott értéket meghívjuk a handleTakeTile metódust, ami a useGame::takeTiles függvénnyel elküldi socket-en keresztül a szerver felé a játék akciót.

```
const handleTakeTiles = async (tile, market, row) => {
  if (tile !== null && market !== null && row !== null) {
    await takeTiles(tile, market, row);
    setSelectedTile(null);
    setSelectedMarket(null);
    setSelectedRow(null);
  }
};
```

43. ábra takeTiles

A játék a Session.jsx oldalon addig tart, amíg nem érkezik GameOver socket esemény a szerver felől, ekkor a gameOver értéket igazra váltjuk, ezzel feltételesen rendereljük a GameOverPanel-t.

```

{gameOver && (
  <div ref={ref} => {
    if (ref) {
      ref.style.opacity = '0';
      setTimeout(() => {
        ref.style.opacity = '1';
      }, 0);
    }
  }} className="h-full w-full flex justify-center items-c
  <GameOverPanel playerBoards={finalPlayedBoards} />
</div>
)}

```

44. ábra GameOverPanel

### StatusBoard:

Property-ként megkapja az adott szoba users (felhasználói) tömbjét és táblázatos formában mutatja a játékosok nevét és státuszát.

### GameOverPanel:

Property-ként megkapja a játék játékos tábláinak tömbjét, a játékosnév és szerzett pont adatokból csinált táblázatot. Megjelenít egy "Accept" gombot, melynek az onClick eseménye meghívja a useJoinRoom::leaveRoom metódusát

### Tile

Property-ként kap egy tile-t a játék állapotból, egy onClick eseményt (Session-ből kapta a Market és továbbadja a Tile-nak), isSelected (property jelzi, ha ki van jelölve a Tile) és egy id-t, ami fontos lesz az animációk miatt.

```

const Tile = ({ tile, onClick, isSelected, id }) => (
  <div
    id={id}
    style={{
      border: isSelected ? '3px solid #000' : '1px solid #ddd',
      backgroundImage: tile ? `url(${tileImages[tile]})` : 'none',
      backgroundSize: 'cover',
      cursor: tile !== 'empty' ? 'pointer' : 'default',
    }}
    className={`w-8 h-8 rounded-md cursor-pointer mx-1 my-1 ${
      tile ? `bg-${tile}` : 'bg-gray-300'
    }`}
    onClick={tile !== 'empty' ? onClick : null}
  />
);

```

45. ábra Tile

### Market:

A piac csempéket (Tile) tart. Ezeket a gameState-től a tiles property-én keresztül kapja meg.

onTileClick, selectedTile, selectedMarket property-eit továbbküldi a gyerek(Tile)eknek

### SharedMarket:

Hasaló a Markethez, itt a marketId fix -1

### PlayerBoardCard:

propertynek kap egy playerBoardot és egy onClick metódust. A playerBoarddal feltölti adatokkal a kártyát és a collectedTiles soraihoz hozzárendeli az onClicket. Ez szükséges lesz a csempe mozgatásához (Session.jsxben handleTakeTile)

A kártya tartalmazza a játékos nevét, pontszámát, collectedTiles-, wallTiles- és floorTiles tömbjét.

### ChatBox:

Egy chat ablakot ábrázol. Két komponense a Messages és a MessageInput. Ezen felül mutatja a szoba nevét.

### Messages:

A useListenMessages::messages értékével készít listát Message komponensekből. A legutóbbi üzenetet a React useRef hook segítségével eltároljuk és egy useEffect hook-kal legörgetünk hozzá. Így mindig látni fogjuk a legfrissebb üzenetet.

```
const Messages = () => {
  const { messages, loading } = useListenMessages();
  useListenMessages();
  const lastMessageRef = useRef();

  useEffect(() => {
    setTimeout(() => {
      lastMessageRef.current?.scrollIntoView({ behavior: "smooth" });
    }, 100);
    console.log("Updated messages state:", messages);
  }, [messages]);

  return(
    <div className='px-4 flex-1 min-h-60 max-h-60 overflow-y-auto no-scrollbar'>
      {!loading &&
        messages.length > 0 &&
        messages.map((message) => (
          <div key={message._id} ref={lastMessageRef} className="break-all">
            <Message message={message} />
          </div>
        ))}
    </div>
  )
}
```

46. ábra Messages

### MessageInput:

Egy form, amely egy input mezőből és egy küldés gombból áll. A submit eseményt a gombbal és az enter billentyű lenyomásával is kiválthatjuk. Egy useState változóban message tároljuk az aktuális üzenetet és a useListenMessages::sendMessage metódusával küldjük el.

### Message:

Property-nek kapott message-ből készít egy megjeleníthető üzenetet.

```
return (
  <div className={`message ${authUser._id === message.senderId ? "sent"
: "received"}`} >
    <p><i>{message.senderName}</i>: {message.message}</p>
  </div>
);
```

### Navbar:

Egy navigációs sáv, ami tartalmazza a kijelentkezéshez szükséges gombot, egy súgó gombot és megjeleníti a bejelentkezett felhasználó nevét. A Navbar tartalmazza a HelpPanel komponenst is, amelyet egy showHelpPanel useState változó értéke alapján mutatunk a felhasználók felé. A showHelpPanel értékét a súgó gomb onClick eseményével változtatjuk.

### HelpPanel:

Súgóként használjuk, információt biztosít az alkalmazás használatáról HowToUse komponens és a játék szabályairól GameRules komponens. Hogy melyik komponenst kell megjeleníteni azt a [selectedTab, setSelectedTab] = useState változóban tároljuk. selectedTab értékét a HelpPanel „Game Rules” és „How to use” gombjaival tudjuk beállítani. A súgó tartalmaz még egy gombot az ablak bezárásához.

### HowToUse, GameRules:

Statikus komponensek, egy JSX állománnyal térnek vissza.

### Animációk

Az AnimateChanges.js fájl biztosít számunkra egyszerű animációkat a játékhoz. Ezeket a useListenGame-ben használjuk, amikor kapunk egy új játék állapotot. Összehasonlítjuk a



mostani és a következő gameState-et és az alapján végezzük el az animációkat a következő metódusokkal:

- animateTakeTiles, csempék mozgatása a TakeTiles esemény alapján Market -> PB (PlayerBoard)
- animateRoundOver, csempék mozgatása RoundOver esemény alapján PB.collectedTiles -> PB.WallTiles

Használunk egy handleTileMove segéd metódust, ami kap egy toElementId-t és egy fromElementId-t. DOM-on megkeresi ezeket és a két pont között egy ideiglenes csempe mozgását animálja meg.

```
const handleTileMove = (data) => {
  try {
    console.log("MoveTile", data);
    const { fromElementId, toElementId } = data;

    console.log("Still MOVING")

    const fromElement = document.getElementById(fromElementId);
    const toElement = document.getElementById(toElementId);

    if (fromElement && toElement) {
      const fromRect = fromElement.getBoundingClientRect();
      const toRect = toElement.getBoundingClientRect();

      // Create a temporary element for the animation
      const tempTile = fromElement.cloneNode(true);
      tempTile.style.position = 'absolute';
      tempTile.style.top = `${fromRect.top}px`;
      tempTile.style.left = `${fromRect.left}px`;
      tempTile.style.width = `${fromRect.width}px`;
      tempTile.style.height = `${fromRect.height}px`;
      document.body.appendChild(tempTile);

      // Animate the tile movement
      tempTile.animate([
        { transform: `translate(${toRect.left - fromRect.left}px, ${toRect.top - fromRect.top}px)` }
      ], {
        duration: 500,
        easing: 'ease-in-out'
      }).onfinish = () => {
        document.body.removeChild(tempTile);
      };
    }
  } catch (error) {
    console.error(error);
  }
};
```

47. ábra Csempe mozgatás animáció

- `animatePlayerLeft`, `PlayerLeftGame` eseménynél kiszürkíti a kilépett játékos tábláját, ezzel jelezve, hogy a játékos elhagyta a játszmát. Ehhez az átmenethez egy CSS stílust adunk a játékos táblájához

```
leftPlayerBoardElement.classList.add('desaturate');
```

```
@keyframes desaturate {
  from {
    filter: saturate(100%);
  }
  to {
    filter: saturate(0%);
  }
}

.desaturate {
  animation: desaturate 0.5s forwards;
  pointer-events: none;
  opacity: 0.5;
}
```

48. ábra kiszürkítés css

## Tesztelés

A teszteléshez egy új adatbázist hozok létre. Ehhez elegendő a `MONGO_URI` környezeti változóban tárolt connection stringet átírni. Ha megváltoztatjuk a DB nevét, akkor a MongoDB Atlas egy új Collection-t hoz létre az új névvel.

A teszteléshez egy feketedoboz megközelítést alkalmazok, ahol a webalkalmazás funkcióit fogom kézzel tesztelni, közben ellenőrizve azok helyes működését. A teszteseteket az alkalmazás funkciói alapján csoportosítottam, leírtam, hogy az adott funkciót miként használtam és mit tapasztaltam.

## Regisztráció

- Nem töltünk ki minden mezőt: el se küldjük a http request-et, hibát kapunk. Toaster error-ral van kezelve.
- Kitöltünk minden mezőt, de nem azonos jelszót adunk meg: status 400 Bad request, toaster erroral jelezve (Passwords don't match).
- Jelszavak egyeznek, de túl rövid: status 400 Bad request, toaster error(Password must be at least 6 long).

- Helyes bejelentkezés: beléptünk a fő oldalra. User adatai megjelentek az adatbázisban, authUser eltárolva a localStorage-ben.
- Újra regisztrálás azonos névvel: status 400 Bad request, toaster error(username already exists).

#### **Bejelentkezés:**

- Nem töltünk ki minden mezőt: el se küldjük a http request-et, hibát kapunk. Toaster error-ral van kezelve.
- Helytelen (nem létező username) névvel kitöltve: status 400 Bad request, toaster error(Invalid credentials).
- Helyes username, rossz jelszó: status 400 Bad request, toaster error(Invalid credentials).
- Helyes adatok megadása: beléptünk a főoldalra. authUser eltárolva a localStorage-ben.

#### **Kijelentkezés:**

- Kijelentkezés a webfelületről: helyesen kijelentkeztünk, localStorage authUser-e null, átirányított minket a Login page-re.
- Nem vagyunk bejelentkezve: ReactDOM-ban nincs is felvéve a logout elem.
- Szobába vagyunk, játék nélkül: sikeres kijelentkezés, localStorage authUser-e null, minket átirányított a Login page-re.
- Szobában vagyunk, fut a játék: sikeres kijelentkezés, localStorage authUser-e null, minket átirányított a Login page-re.

#### **Fő oldal / Szobák:**

- Névnélküli szoba létrehozása: status 400 Bad request, toaster error (couldnt host room)
- Szoba hozztolása csak név: szoba felvételre került, felhasználó automatikusan csatlakozott, átnavigált a szoba nézetre.
- Szoba hozztolása egy meglévő szoba nevével: status 400 Bad request, toaster error (couldnt host room)
- Meglévő várokozó szobához csatlakozni, ahol < 4 felhasználó van: a felhasználó sikeresen csatlakozott, átnavigált a szoba nézetre.

- Meglévő szobához csatlakozni, ahol 4 felhasználó van: nem lehet csatlakozni, nincs is rá lehetőség.
- Teli vagy játékot már megkezdett szobához csatlakozni: nincs lehetőségünk.

#### **Chat:**

- Üzenet küldés: a szoba összes felhasználójánál megjelenik.
- Meglévő szobához csatlakozva: ha volt beszélgetés, akkor az töltődik be. Ha nem volt, akkor üres.
- Más szobához csatlakozva: előző beszélgetésünk helyett, az új szoba beszélgetését látjuk.

#### **Szoba elhagyás:**

- Nem utolsóként hagyjuk el: visszakerülünk a fő oldalra, a szoba még a listában.
- Utolsóként hagyjuk el: visszakerülünk a fő oldalra, szoba kitörlődik a listából.
- Elhagyás utáni újra csatlakozás: kiszámítható módon a fő oldal esetei szerint.

#### **Játék:**

- Ready-zés, mindenki ready: A játék elindul, szoba állapota „game in progress”-re vált. Ready button eltűnik.
- Ready-zés, nem mindenki ready: kiszámíthatóan csak akkor indul el, ha egy pillanatban minden felhasználó ready.
- Nem a saját körünkben lépünk: nem történik semmi, toaster error(Not your turn)
- Saját körünkben nem szabályos lépés: nem történik semmi, toaster error(Invalid move)
- Saját körünkben szabályos lépés: Csempék játékszabály szerint kerülnek elhelyezésre, mindenkinél frissül a játékállapot. Új játékos következik.
- A kiválasztott csempénket az előző játékos kihúzza, majd a sorunkra kattintunk: nem történik semmi, továbbra is a mi körünk van. toaster error(No tiles to take)
- Egy felhasználó játék közben kilép: játék megy tovább, a kilépett játékos táblája kiszürkül, sose fog sorra kerülni. Következő körtől a piacok száma újra csökken.
- Egy felhasználó saját köre közben kilép: kiszürkül a táblája, a következő játékos kerül sorra. A játék zökkenőmentesen folytatódik.

- Egy felhasználó kilépése után egyedül maradunk játékban: azonnali Game Over, nem tudjuk folytatni a játékot. Játék vége panel megjelenítése.
- A játékszabályok szerinti játék vége: Játék vége panel megjelenítése.
- Játék vége elfogadása: szoba elhagyása gombbal egyenértékű.

A tesztelés célja a felhasználók számára egy kiszámítható, megbízható és hibamentes weboldal biztosítása. A manuális tesztelés során minden lehetséges felhasználói interakciót kipróbálunk, hogy az esetleges hibákat és hiányosságokat azonosítsuk. A cél egy olyan felhasználói élmény biztosítása, amely minden elvárásnak megfelel, és zavartalan felhasználói élményt biztosít.

A program jelenleg minden(fenti) tesztnek eleget tesz. A tesztelés alatt egy hibát találtam, a „A kiválasztott csempénket az előző játékos kihúzza”. Ezt azonnal javítottam, a játék alatt nem tudunk már nem létező csempét kihúzni.

## 4. Összefoglalás és további fejlesztési lehetőségek

### 4.1. Összefoglalás

Ez az alkalmazás, melyet a hozzám hasonlóan társasjáték-kedvelő célközönség számára fejlesztettem, egy online platformot biztosít, ahol barátok, családtagok, ismerősök vagy akár teljesen idegenek közösen élvezhetik a társasjátékot, függetlenül a fizikai távolságtól. Az alkalmazás tervezése és fejlesztése során kiemelten fontos szempont volt, hogy a felhasználói élmény egyszerű és intuitív legyen. Ennek érdekében a felhasználói interfészt úgy alakítottam ki, hogy az magától értetődő legyen, és bárki könnyen tudja használni, akár technikai háttérismeret nélkül is.

Az alkalmazás tartalmazza a játékok indítását, csatlakozást a különböző játék szobákhoz, és a játék közbeni kommunikációt, ami lehetővé teszi a valós idejű interakciót a játékosok között.

### 4.2. További fejlesztési lehetőségek

Továbbfejlesztéshez az első ötletem az lenne, hogy a jelenlegi 1 társasjátékot implementáló weboldalból egy társasjáték hub-ra fejlesztenénk. Az oldal szerkezetét csak részben kéne módosítani. A szoba készítésnél lehetőséget adnánk társasjáték választásra és az így létrejövő szoba felhasználóinak száma a játékéhoz azonosulna. A jelenlegi Session page-ből absztraktálnánk a játékot, helyette különböző játék komponenseket készítenénk. Minden egyes új játékhoz továbbá szükségünk lesz:

- Egyedi hookokra, hogy tudjunk a szerverrel kommunikálni
- Adatbázis sémára a játékhoz
- Socket kezelőre
- új játék kontrollerre

## 5. Irodalomjegyzék, hivatkozások

- [1] Azul: <https://planbgames.com/next-move/games/azul-next-move-games-michael-kiesling-strategy-abstract-board-game-winner-spiel-des-jahres-game-of-the-year-cannes-portuguese-tiles-royal-palace-of-evora-5>

(A játék hivatalos kiadója)

- [2] Node.Js: <https://nodejs.org/en>
- [3] ExpressJS: <https://expressjs.com/>
- [4] MongoDB: <https://www.mongodb.com/>
- [5] MongoDB Atlas: <https://www.mongodb.com/products/platform/atlas-database>
- [6] React: <https://react.dev/>
- [7] Tailwind CSS: <https://tailwindcss.com/>
- [8] DaisyUI: <https://daisyui.com/>
- [9] Socket.IO: <https://socket.io/>