**Eötvös Loránd Tudományegyetem**

**Informatikai Kar**

**Informatikatudományi Intézet**

**Média- és Oktatásinfromatika Tanszék**

Online többjátékos társasjáték megvalósítása, Azul

Szerző: Témavezető:

Gyepes Gergő Korom Szilárd

Programtervező informatikus BSc. Doktorandusz

**Budapest, 2023**

Ide kerül a hivatalos témabejelentő lap.

Tartalomjegyzék

[1. Bevezetés 1](#_Toc87521353)

[2. Felhasználói dokumentáció 2](#_Toc87521354)

[3. Fejlesztői dokumentáció 3](#_Toc87521355)

[4. Összefoglalás és további fejlesztési lehetőségek 4](#_Toc87521356)

[5. Irodalomjegyzék 5](#_Toc87521357)

[6. Melléklet 6](#_Toc87521358)

# Bevezetés

A dolgozatomban egy teljes webalkalmazás implementációja volt a célom. Minden építőelemét magamtól megtervezni, implementálni és egy teljes egész összeépíteni.

A szakdolgozatom célja egy olyan könnyen hozzáférhető, letisztult és egyszerűen kezelhető játék készítése volt, amit

# Felhasználói dokumentáció

A dolgozat egy böngészőben játszható társasjáték megvalósítását tűzte 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 követit. 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.

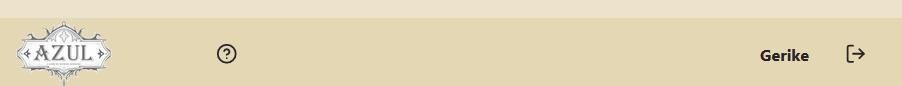
Az webalkalmazás felépítése.

Az alkalmazás egy úgy nevezett SPA (single page application), de a könnyen érthetőség kedvéért, az egy böngésző oldalon, egy időben megjelenő felületet oldalnak 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.

2.1 Alsó 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.



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.

A képen szöveg, képernyőkép, Betűtípus, szám látható

Automatikusan generált leírás

2.2 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 képen szöveg, képernyőkép, Betűtípus, Téglalap látható

Automatikusan generált leírás

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

A képen szöveg, képernyőkép, Betűtípus, tervezés látható

Automatikusan generált leírás

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

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

2.3 Fő oldal

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

2.3.1 Új szoba felvétele

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

A képen szöveg, képernyőkép, Betűtípus, szám látható

Automatikusan generált leírás

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

A képen szöveg, képernyőkép, Betűtípus, Téglalap látható

Automatikusan generált leírás

2.3.2 Meglévő szobák

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

A képen szöveg, képernyőkép, tervezés látható

Automatikusan generált leírás

Az itt megjelenő kártyák a szobákról tartalmaznak 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.

2.4 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 játékostársainknak, hogy készen állunk a játék indítására. A gombok alatt található egy chat ablak, amellyel valós időben tudunk kommunikálni a szoba tagjaival.

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

A képen szöveg, képernyőkép, Téglalap, diagram látható

Automatikusan generált leírás

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.

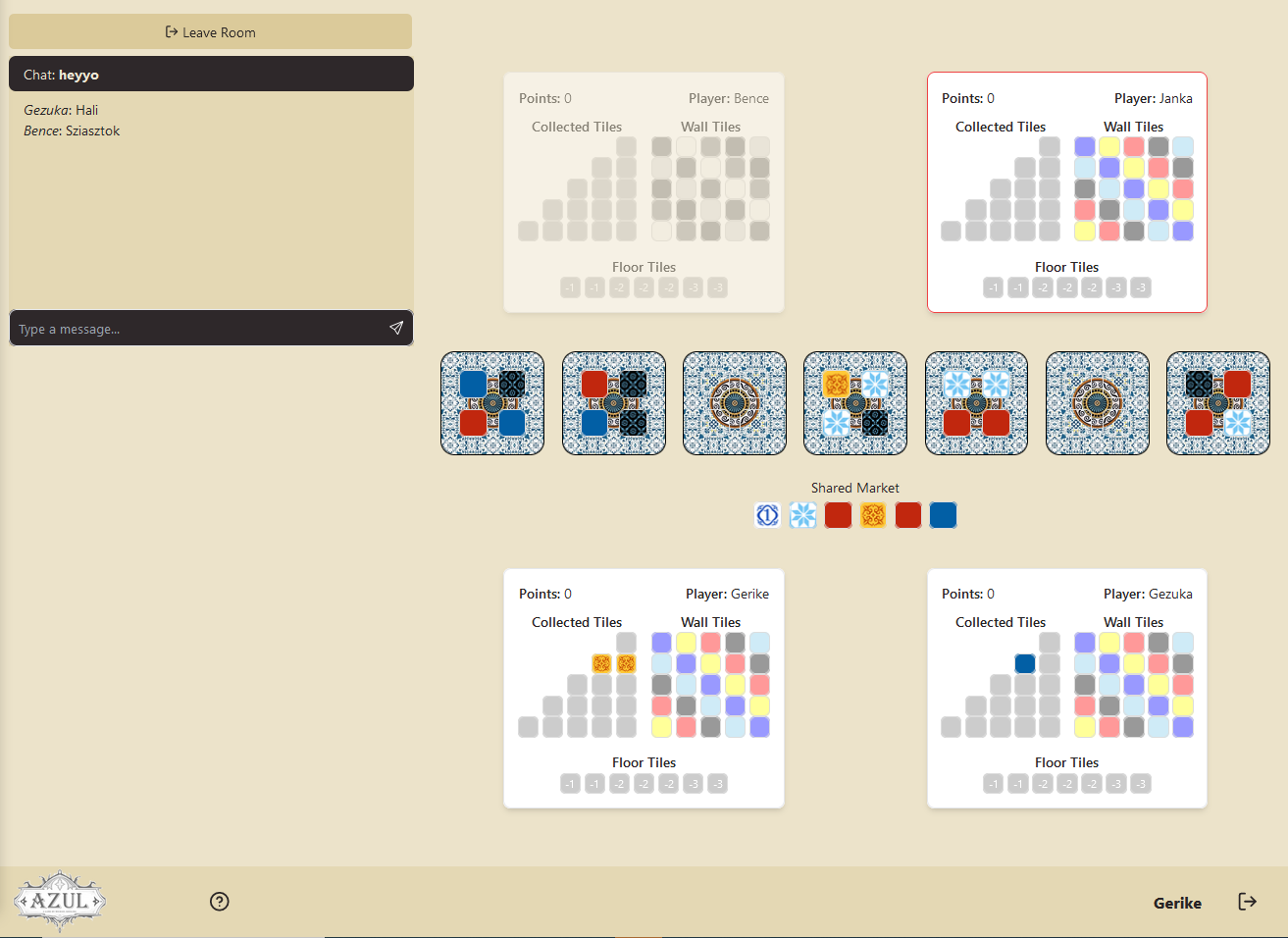
A játéktéren a státusz panelt leváltja az aktuális játéktábla.

A képen szöveg, képernyőkép, diagram, Téglalap látható

Automatikusan generált leírás

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 köre van és zölddel, ha a mi következünk.

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.



Ha egy játékos kilépése után nincs megfelelő mennyiségű játékos a szobában, véget ér a játék.

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

A képen szöveg, képernyőkép, Betűtípus, sor látható

Automatikusan generált leírás

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

A képen szöveg, képernyőkép, Számítógépes ikon, szoftver látható

Automatikusan generált leírás

A játék menet és egyéb funkciók nem változnak.

# Fejlesztői dokumentáció

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

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',

    },

})

const User = mongoose.model("User", userSchema)

Szobához:

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 })

const Room = mongoose.model('Room', roomSchema)

Üzenetekhez és a chat-hez

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 Message = mongoose.model("Message", messageSchema)

const chatSchema = new mongoose.Schema({

    roomId: {

        type: mongoose.Schema.Types.ObjectId,

        ref: "Room"

    },

    messages:[

        {

            type: mongoose.Schema.Types.ObjectId,

            ref: "Message",

            default:[]

        }

    ]

})

const Chat = mongoose.model('Chat', chatSchema)

Játékhoz (és játékos táblához)  
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'

    }

});

const Game = mongoose.model('Game', gameSchema);

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'],

    }

  });

const PlayerBoard = mongoose.model('PlayerBoard', playerBoardSchema);

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.

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/authRouts.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}`)

})

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

Kommunikáció és kérések

Routes

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

Az authentikációs 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;

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. Ezekhez az eseményeket a gameController metódusaival kezeli.

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);

    });

}

messageController:

a messageControllert használja a sendMessage és getMessage események lereagálásához.

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 válaszunkat 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 bycript segítségével hash-eljük és úgy tároljuk el.
* login: a kapott adatokkal megkeres egy felhasználót és bycript 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 szobákon 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 játkban 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, cask a controllerből érhető el. feltölti a piacokat és a közös piacot. majd ezt a jétékállapotot küldi el a szoba felhasználóinak (NewRound)

takeTiles, kap egy csempét(színt), marketId-t, row-t (sorindex) és a socket handshakeből a userId-t. Ezekből meg tudjuk határozni, hogy melyik játékos, melyik piacról, melyik csempéket vette el. Az elvett csempékkel feltöltjük a játékos táblájának a collectedTiles[row] sorát. Ha szükséges, a floorTiles-t is.TakeTiles eseményt küldönk, 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 csempéket 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, akkkor 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 értket). 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 maradz, akkor folytatjuk a játékot, PlayerLeftGame eseményt küldünk a játékállásról. Ha 1 vagy 0 játékos maradt a játéban, akkor GameOver eseménnyel lépünk ki.

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

newBag, newMarkets,newPlayerBoards felvesz megflelő 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.

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 segítségével prop-okat tudunk eljuttatni a szülő és összes gyerek eleme között. A context adatait az összes további gyerek elemben elérjük, segítik az adatok továbbítását.

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

main.jsx

ReactDOM.createRoot(document.getElementById('root')).render(

  <React.StrictMode>

    <BrowserRouter>

      <AuthContextProvider>

        <SocketContextProvider>

          <RoomContextProvider>

            <ChatContextProvider>

              <GameContextProvider>

                <App />

              </GameContextProvider>

            </ChatContextProvider>

          </RoomContextProvider>

        </SocketContextProvider>

      </AuthContextProvider>

    </BrowserRouter>

  </React.StrictMode>,

)

A képen 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.

AuthContex:

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>

}

SocketContext:

a socket.io-client ből 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]);

RoomContext:

A jelenleg aktív szobákat tároljuk benne. Biztosít egy rooms változót és egy setRooms setter metódust.

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 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

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 authUsert 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 lekérdezhetjük az összes akítv 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 eventtel 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]);

};

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 };

}

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) {

        case ...

      }

      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));

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

useListenMessages:  
A chat funkcionalitásához szükséges. Új üzeneteket fogad, melyek a newMessage socket eseménnyel érkeznek. Amikor szobát váltunk vagy frissül a *socket* (socketContextből) egy socket eseménnyel, a getMesseges-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.

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) => {

        if(!message) return;

        if(!authUser.roomId) return;

        socket?.emit("sendMessage", { receiverId: authUser.roomId, message});

    };

    return {sendMessage, messages};

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 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>

  );

}

További frontend komponenseink

Signup oldal

Egy űrlapot ábrázol mellyel megadható a felhasználó neve, jelszava és a jelaszava 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();

    if (res.ok) {

      setPlayerReady((prevState) => !prevState);

    } else {

      console.error("Error toggling ready status: ", data.error);

    }

  };

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]);

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-k 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) => {

    setSelectedTile(tile);

    setSelectedMarket(marketId);

    if (selectedRow !== null) {

      handleTakeTiles(tile, marketId, selectedRow);

    }

  };

{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>

                    ))}

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 fügvénnyel 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);

    }

  };

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-center transition-opacity duration-500 ease-in-out">

              <GameOverPanel playerBoards={finalPlayedBoards} />

            </div>

          )}

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 játékostábláinak tömbjét, a játékosnév és szerzett pont adatokból csináltáblázatot. 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 (prop, 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}

  />

);

Market:

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

onTileClick , selectedTile, selectedMarket property-eit továnnküldi a gyerek(Tile)einek

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 florrTiles 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>

    )

}

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);

    }

};

* 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;

}

Tesztelés

Az API végpontjaimat Postman-nel teszteltem.

# Összefoglalás és további fejlesztési lehetőségek

# Irodalomjegyzék

# Melléklet