Online Quarto

Az applikáció programozási specifikációja

Tartalom jegyzék:

1. Az applikáció célja:

Az applikáció célja egy reszponzív webes felületen keresztül bemutatni egy szerver és a hozzá kapcsolódó kliensek közötti kommunikációt egy web applikáció formályában, ami magában foglal egy instant chat alkalmazást és egy egyszerű táblás játékot, mindezt Node.js szerver alapú futtatási környezetben megoldva és az adatokat MySQL relációs adatbázis kezelő rendszerben tárolva. Ezen kívül mind az applikáció chat és játék része erősen támaszkodik a Socket.io web csatlakozóra (socket) ami felelős a kliensek és a szerver közti kommunikációért és a köztük folyó adat forgalomért.

2. Felhasznált technológiák

2.1 Fejlesztési környezet:

A fejlesztési környezet alapvetően Microsoft Windows 10 és Manjaro Linux operációs rendszereken történt. Ezen operációs rendszerek alatt lett tesztelve több különböző böngészővel.

A tesztelt böngészők:

- Google Chrome
- Mozilla Firefox
- Microsoft Edge

A felhasznált fejlesztői eszköz a Microsoft Visual Studio Code nyílt forrású szerkesztő a következő kiegészítőkkel:

- Live server
- Rest Client

A szerverek:

Az applikációnak alapvetően csak a MySql szerverre van szüksége a működéshez, de kényelmi okokból a phpMyAdmin adatbázis kezelői felület eléréséhez az Apache http szervert is felhasználtam fejlesztés közben, Windows platformon ezt a XAMP csomag és Linux platformon a LAMP csomag használatával tettem.

2.2 Node.js:

Az applikáció alapvetően a **Node.js** JavaScript alapú és szerver oldali futtatási környezet felhasználásával készült. Alapvetően szerver oldali leképzést használok (SSR: server side rendering) evvel kiváltva a frontend teljesen külön való fejlesztését és ennek köszönhetően pár előnyös tulajdonságot elérve:

- A szerveroldali leképzés lehetővé teszi az oldalak gyorsabb betöltését, javítva ezzel a felhasználói élményt.

- Szerveroldali megjelenítéskor a keresőmotorok könnyen indexelhetik és feltérképezhetik a tartalmat, mivel a tartalom még az oldal betöltése előtt leképezhető, ami ideális SEO számára
- A weboldalak megfelelően vannak indexelve, mert a böngészők a gyorsabb betöltési idővel rendelkező weboldalakat részesítik előnyben.
- A szerveroldali megjelenítés segít a weboldalak hatékony betöltésében a lassú internetkapcsolattal rendelkező vagy elavult eszközökkel rendelkező felhasználók számára.

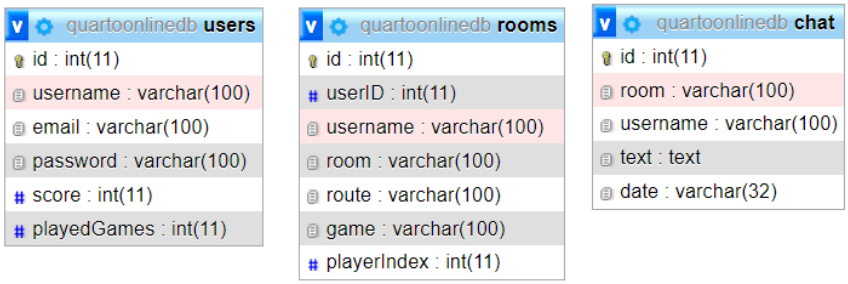
Ettől függetlenül a dokumentációban mégis a hagyományos módon (frontend – backend) fogok utalni az applikáció részeire mivel viselkedés szempontjából két részre lehet választani, első sorban a Socket.IO web csatlakozó miatt.

2.3 Socket.IO

A **Socket.IO** egy olyan könyvtár, amely alacsony késleltetésű, két irányú és eseményalapú kommunikációt tesz lehetővé az ügyfél és a szerver között. A WebSocket protokollra épül, és további garanciákat nyújt, mint például a HTTP hosszú lekérdezés vagy az automatikus újra csatlakozás.
Minden az oldalra bejelentkezett felhasználó egyrészt egy session-be kerül másrészt a felhasználó és a szerver között ilyenkor kiépül egy web socket kapcsolat, amin keresztül történik egyrészt a weboldal chat részének a kommunikációja másrészt a két játékos ezen a csatlakozón keresztül kapcsolódik a játékhoz és az abban történt események ezen a csatlakozón keresztül lesznek szállítva.

2.4 MySql

Az **adatbázis** és az adatok tárolása **MySql** relációs adatbázis segítségével van megoldva. Minden a chat ablakokban történt esemény, és minden a játékkal kapcsolatos eredmény vagy a felhasználók szerveren való közlekedése adatbázisban van eltárolva. Gyakorlatilag sikerült le minimalizálnom és három táblával megoldani az egész applikáció kezelését:



A **users** tábla tartalmazza a regisztrált felhasználók adatait:
- id: a felhasználó egyedi azonosítója
- username: a felhasználó neve, amivel be tud jelentkezni

- email: a felhasználó e-mail fiókja
- password: a felhasználó jelszava
- score: a felhasználó játék pontszáma
- playedGames: hány játékot játszott a felhasználó

A **rooms** tábla tartalmazza, hogy az aktuális felhasználó épp melyik szobában van:
- id: a tábla indexe
- userID: a session-ben tárolt és szállított, de a users táblából kinyert egyedi felhasználói azonosító id
- username: a felhasználó neve
- room: az aktuális szoba neve
- route: az aktuális végpont, amin tartózkodik a kliens
- game: ha épp játékban van az aktuális játék szoba neve
- playerIndex: az aktuális játékban az éppen soron következő játékos indexe

A **chat** tábla tartalmazza az aktuális szobák beszélgetéseit.
- id: a tábla indexe
- room: az aktuális szoba neve
- username: a kliens neve
- text: maga a beszélgetés üzenete
- date: mikor történt az elküldött üzenet

3. Szerver oldal (“backend”):

3.1 server.js:

A server.js az alap szerver indító forrás álománya, különböző a Node.js futtatási környezethez tartozó applikációs interfészek - keretrendszerek és modulok inicializálásával kezdődik:

express: Az ExpressJS a Node.js minimális, valamint rugalmas webes alkalmazási keretrendszerének tekinthető, amely robusztus funkciókat kínál a web és a mobil alkalmazások használatához. Az ExpressJS-t nyílt forráskódú keretnek is tekintik, amelyet a Node.js alapítvány fejlesztett ki és tart fenn.
Minimális felületet biztosít az alkalmazások készítéséhez. Az ExpressJS ezen kívül különböző eszközöket ad nekünk, amelyekre szükség van az alkalmazás felépítéséhez.

express-session: A HTTP állapot nélküli; ahhoz, hogy egy kérést bármilyen más kérelemhez társíthasson, valamilyen módon szükség van a felhasználói adatok tárolására a HTTP kérések között. Erre nyújt megoldást az express-session modul.

http: A http “ügynök” felelős a webes kapcsolat fennmaradásának és a HTTP-kliensek újra felhasználásának kezeléséért. Sorban tartja a függőben lévő kéréseket egy adott gazdagéphez és ’port’-hoz, és mindegyikhez egyetlen ’socket’ kapcsolatot használ mindaddig, amíg a sor ki

nem ürül, ekkor a ’socket’ vagy megsemmisül, vagy egy készletbe kerül, ahol megtartják, hogy újra felhasználhassák a kérésekhez. ugyanarra a gazdagépre és portra.

ejs: Az EJS egy egyszerű sablonnyelv, amely lehetővé teszi HTML-jelölések generálását egyszerű JavaScript használatával.

morgan: HTTP kérésnaplózó köztes szoftver a node.js számára

os: Ez a modul számos olyan funkciót kínál, amelyek segítségével információkat kérhet le az alapul szolgáló operációs rendszerről és arról a számítógépről, amelyen a program fut, és interakcióba léphet vele.

fs: Az fs modul számos nagyon hasznos funkciót kínál a fájlrendszer eléréséhez és az azzal való interakcióhoz.

moment: A moment modul a dátumok és időpontok értelmezésére, érvényesítésére, manipulálására és megjelenítésére szolgál JavaScriptben.

db: a mysql modul segítségével létrehozzuk az adatbázist kezelő modellt

initDB: a mysql adatbázis inicializálása és felkészítése a munkára.

router: Az útválasztás annak meghatározására vonatkozik, hogy egy alkalmazás hogyan válaszol egy adott végponthoz intézett ügyfélkérésre, amely egy URI (vagy elérési út) és egy adott HTTP-kérés metódusa (GET, POST és így tovább). Ennek viselkedését itt inicializáljuk.

socket.io: A Socket.IO egy olyan könyvtár, amely alacsony késleltetésű, két irányú és eseményalapú kommunikációt tesz lehetővé az ügyfél és a szerver között.

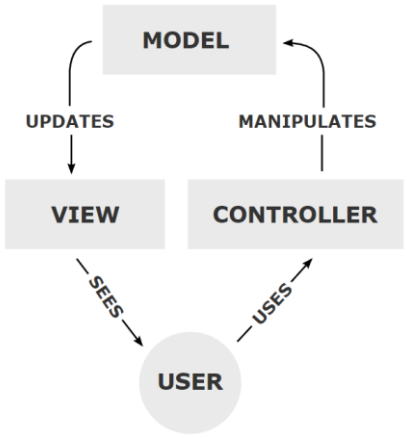
ansi: egy egyszerű objektum, amiben tárolva vannak különböző ’ANSI escape’ szekvenciák. Ezt a szerver konzolján történő kiíratások formázására használom.

- TWO_HOURS: változó beállítása, a cookie-k az ebben a változóban beállított idő intervallum után semmisülnek meg.
- inicializáljuk az express modult: app = express();
- inicializáljuk a munkamenet köztes szoftvert (sessionMiddleware)
    - **secret**: egy véletlenszerű egyedi karakterlánc-kulcs, amelyet a munkamenet hitelesítésére használnak. Környezeti változóban van tárolva, és nem teheti közzé a nyilvánosság számára. A kulcs általában hosszú és véletlenszerűen generálódik éles környezetben.
    - **resave**: lehetővé teszi a munkamenet visszatárolását a munkamenet-tárolóba, még akkor is, ha a munkamenet soha nem módosult a kérés során. Ez versenyhelyzetet eredményezhet, ha egy kliens két párhuzamos kérést küld a szervernek. Így az első

kérés szekciójában végrehajtott módosítás felülírható, amikor a második kérés véget ér.
    - **saveUninitialized**: ez lehetővé teszi bármely inicializálatlan munkamenet elküldését az áruházba. Amikor egy munkamenetet létrehoznak, de nem módosítanak, azt inicializálatlannak nevezzük.
    - **Cookie**: maxAge: ez beállítja a cookie lejárati idejét. A böngésző a beállított időtartam letelte után törli a cookie-t. sameSite: azt jelzi, hogy a cookie „ugyanaz a webhely” cookie e.
- hozzáadjuk az applikációhoz a sessionMiddleware-t (app.use(sessionMiddleware);)
- urlencoded: a bejövő http kérések url kódolásának elemzése middleware
- beállítjuk a publikus mappa elérési útvonalát (app.use(express.static('./assets/public'));)
- beállítjuk és inicializáljuk az EJS sablon nyelv motorját (app.set('views', './API/view/'); app.set('view engine','ejs');)
- elinditjuk a morgan modult, ami a server terminal kimenetén logoltatja a http kéréseket
- beállítjuk az applikációhoz a gyökér elérési útvonalat a routerban (app.use('/',router);)
- inicializájuk és elindítjuk a http szervert
- elindítjuk a socket.io modult az ./API/tools/socketIO

3.2 Az API:

Az applikációs felület az úgynevezett MVC szoftvertervezési mintára alapul. Ez egy olyan minta, amelyet általában olyan felhasználói felületek fejlesztésére használnak, amelyek a kapcsolódó programlogikát három, egymással összefüggő elemre osztják, a Model, a View és a Control elemekre.



Ennek a mintának először a Control elemét mutatom be.

A Control az applikációs végpontok viselkedését irányítja, pl.: különböző HTTP kérésekre hova szállítsa el a felhasználót az oldalon vagy hova küldjön adatokat. Ezeket a router (útvonal választó) fogja össze a **router.js** (./API/router.js) file-ban.

A router.js állományban található szintaktika a következő képen alakul:
[router objektum amiben tárolva vannak a kérések nevei].[http kérés neve]([az objektum amiben benne van a http kéréshez kapcsolt funkció],)
Pl.: a **router.get('/', auth.GET_root);** egy HTTP GET művelettel elviszi a klienst a '/' útvonalra azaz a szerver gyökerébe. A **router.post('/register', auth.POST_register);** egy HTTP POST művelettel elküldi a felhasználó regisztrációs adatait a /register végpontra.

A router.js tartalmaz még két funkciót:
redirectLogin(): visszavezérli a klienst a login oldalra ha az adott session-ban nincs tárolva userID. Ez azt a célt szolgálja hogyha az adott session-ben van userID tárolva akkor automatikusan hitelesítve lesz a kliens és a login adatok beírása nélkül beengedi az oldalra.
redirectLanding(): ha az aktuális session-ben van userID tárolva akkor elvisz a session-ben tárolt útvonalra. Ez például arra jó, hogy ha egy kliens kijelentkezik azt oldalról de még benne van ugyanabban a session-ben akkor kényelmesen vissza rakja a klienst a web oldalon legutoljára járt útvonalra.

3.3 A kontrolok (./API/control/):

Két fajta kontrollt készítettem el. Az egyik a szerverre való bejelentkezés és hitelesítéssel kapcsolatos (./API/control/auth.js), a másik (./API/control/landing.js) magán a weboldalon való közlekedéssel és adat mozgatással kapcsolatos. Mivel az egész applikáció szerver oldali leképzést használ ezért a két kontrol állományban található kontrolok nagy része az EJS sablon nyelv használatával elkészített DOM-okat jeleníti meg a végpontokon.

./API/control/auth.js

- GET_root: elvisz a szerver gyökerébe és EJS-el megjeleníti a gyökér végpontont található HTML dokumentumot.
- GET_login: elvisz a login oldalra és megjeleníti a HTML tartalmat.
- POST_login: a login oldalon a bejelentkezési űrlapon megadott értékek alapján lekérdezi az adatbázist, hogy regisztrált e a felhasználó, ha igen be enged az oldalra.
- GET_register: elvisz a regisztrációs oldalra.
- POST_register: a regisztrációs oldalon megadott adatokat itt elemzi ki és regisztrálja a klienst utána pedig belépteti az oldalra.
- POST_logout: mivel a felhasználó itt lép ki az oldalról, ezért először törli az adatbázisból, hogy melyik szobában volt aztán törli a session-t és a sütit. Végül visszavezérel a '/' gyökér útvonalra.

Funkciók:

- sendError(msg,res,route): az EJS sablon nyelv által a HTML dokumentumba beágyazott **errorMsg** változóba beírja a funkció paramétereként megadott hiba üzenetet ha a bejelentkezési/regisztrálási adatok hibásak.
- db_reggister(name, email, passwd1,req,res): mySql adatbázisban tárolja az újonnan regisztrált felhasználót.

- db_login(username, password,req,res): a mySql adatbázisban megnézi hogy létezik e a felhasználó, ha igen belépteti a klienst.

./API/control/landing.js:
- GET_game: elviszi a klienst a /game végpontra.
- POST_game: a /game végponton beállítja a sessionban a socket.io szobát és a játék szobát.
- GET_lobby: elbisz a /lobby végpontra.
- POST_lobby: törli a session-ben a játékszobát, pl., ha vége van egy játéknak.
- GET_highscore: elvisz a pont táblázat végpontra (/highscore)
- GET_help: elvisz a segítség oldalra (/help)
- GET_about: elvisz a rólam oldalra (/about)

Funkciók:
- set_userPath(req,res,path): beállítja a session-ben a kliens elérési útvonalait (room és route) és ezt tárolja adatbázisban.
- set_userGamePath(req,res,path): beállítja a játék útvonalat, ha nincs játék akkor útvonalnak az alap (/lobby) útvonalat állítja be és tárolja adatbázisban.
- getUserInfo(req): beállítja az a sessionokban hordozott felhasználói adatokat.

3.4 A Model (./API/model/):
Itt alapvetően az adatbázis modellt álltom be, mivel más modellre jelen esetben nincs szükség. Az adatbázis MySql alapú.

./API/model/model-mysql.js:
db: a db objektum tárolja a mysql kapcsolódási adatokat, ezt az objektumot hívom meg ha bármilyen adatbázis műveletet akarok kezdeményezni.
db.getConnection: csatlakozok az adatbázishoz és a konzolra ansi szekvenciák segítségével kiíratom a folyamat eredményét.

./API/model/model-mysql-init.js:

Init,init.getConnection,initDB és init.query. inicializálom az adatbázist, elkészítem az adatbázist és a táblákat, ha még nem léteznek és a felhasználó táblát feltöltöm pár előre definiált felhasználói adattal.

3.5 A View (./API/view/):
Ebben a könyvtárban találhatóak ez EJS modul által felhasznált HTML sablon fileok.

- footer: az oldal alap lábléce

- header: az oldal alap fejrésze
- landing: a landing.js control-hoz kapcsolódó HTML EJS sablonok
    - about.ejs: az about oldal HTML sablonja
    - game.ejs: a játék oldal
    - help.ejs: a segítség oldal
    - highscore.ejs: a pont táblázat
    - lobby.ejs: a fő lobby oldal, ahova alapvetően bejelentkezés után megérkezünk

- login: az auth.js control-hoz kapcsolódó HTML EJS sablonok
    - login.ejs: a bejelentkezési oldal
    - register.ejs: a regisztrációs oldal
    - root.ejs: az oldal gyökér végpontja ide érkezik meg mindenki, aki az oldal url-jét beírja a böngészőbe
- widgets: itt kisebb HTML EJS sablonok találhatók, amiket beágyazok az oldallakra mint rész komponensek
    - chat.ejs: az oldalon található chat komponens
    - footer.ejs: minden egyes a weboldalon megatalálható HTTP GET művelettel elérhető végpontnak van egy külön láb léce.
    - game.ejs: a játék HTML EJS sablonja
    - gamelist.ejs: a /lobby végponton található játék lista, ami lista szerűen jelzi, ha valakik kreáltak játékot
    - header.ejs: minden egyes a weboldalon megatalálható HTTP GET művelettel elérhető végpontnak van egy külön fej léce
    - login-logo.ejs: az auth.js-ben meghatározott végpontokon megjelenő Quarto logó komponens
    - logo.ejs: a bejelentkezés után a fejléc komponensben található logó
    - modal.ejs: egy “modal” komponens, ez akkor jelenik meg ha felugró ablakot akarok megjeleníteni, pl.: ki akarsz lépni a játékból, vagy nyertél
    - webGL.ejs: egy a Three.js keretrendszert használó komponens, ez jeleníti meg a háttérben három dimenzióban hullámzó tengert.

Mivel ezek a sablonok többnyire rendkívül rövidek és egyszerű HTML CSS leíró nyelveket használnak nem részletezném őket, kivéve egy - két sablont. Fontos megjegyezni, hogy mindegyik fő sablon (nem a widget-ek) elindítanak egy **socket.io** klienst, tehát az oldalon közlekedve bárhol elérhető a **socket.io** web socket kliens.

Fő sablonok:

./API/view/game.ejs: ez elindítja a **/js/quarto/landing/game.js** modult, amiben található az amőba játék és egyéb kliens oldali funkciók.

./API/view/highscore: egy egyszerű FOR ciklussal HTML táblázatba kiíratom a GET_highscore control által a results objektumban szállított adatbázisból kinyert pont táblázatot.

./API/view/lobby.ejs: ez elindítja a **/js/quarto/landing/lobby.js** modult, amiben található a /lobby végponton működő widgetek egyes kódjai (játék lista, chat ablak stb), ha egy játékos épp játékban van, vagy csak kreált egy játékot a if (userInfo.game == 'null') vizsgálat alapján eltüntetem a játék listát. Ezt azért teszem hogy ha valaki épp játszik vagy kreált egy játékot nem tudjon egyszerre sok játékot csinálni vagy átlépkedni más játékokba.

Widgetek, azaz kisebb komponensek:

./API/view/widgets/chat.ejs

```
<% if(userInfo.route == 'game'){ %>
        <%= userInfo.game %>
<% } else{ %>
        <%= userInfo.room %>
<% } %>
```

Evvel a node.js beágyazott kóddal beállítom, hogy a chat ablak input mezőjének bal oldalán található tag kijelezze az aktuális socket.io szoba nevét.

./API/view/widgets/footer.ejs

```
<% if(userInfo.route == 'lobby' && userInfo.game == 'null'){ %>
```

A **create** game gomb megjelenítése a **footer**-en attól függ, hogy ha kliens a lobby végponton van és nincs játékban.

```
<% if(userInfo.game != 'null' && userInfo.route != 'game'){ %>
```

Ha a kliens játékban van, de épp nem a játék oldalán van, akkor megjeleníti a vissza játékba gombot footer-en.

```
<% if(userInfo.route == 'game'){ %>
```

Ha a kliens a játék végponton van jelenítse meg a **leave** gombot, amivel ki lehet lépni a játékból vissza a lobby szobába és evvel el is veszti a játékot, egyenlő a játék feladásával.

A footer widget tartalmaz még egy rövid java scriptes részt:

```
document.querySelector(`.button_logout`).addEventListener('click', (event)=>{ logout(); });
if(userInfo.route == 'game') {
        document.querySelector(`.button_leave`).addEventListener('click', (event)=>{ leave(); });
}
```

Egér kattintás eseményt adok a fent említett két gombhoz, a **leave** gombhoz csak akkor, ha ez útvonalam a játék végponton van. Mindkettő kap egy hasonló funkciót (**logout**() és **leave**()). Mindkét funkció kreál egy HTML DOM objektumot ami egy kicsi “modal” ablak. Ebben megkérdezzük a klienst, hogy tényleg ki akar e lépni játékból vagy ki akar jelentkezni. Ezek után egy JQuery-s kóddal végrehajtunk egy **ajax**-os POST műveletet, amivel kijelentkezik a játkos vagy csak kilép a játékból vissza a /lobby végpontra. Az **ajax** művelet végén ki “emitel” azaz egy socket.io üzenetet küld az aktuális játékszobába, hogy a játékos elhagyta a játékot. A sockte.io a kapott üzenet alapján kilépteti a játékban maradt felhasználót a szobából vissza a /lobby végpontra, de ennek a részleteit majd később tárgyalom.

Pl.:

```
$("#leave-form").submit(function(e) {
        e.preventDefault();
        $.ajax({
           url: "/lobby",
           type: "POST",
           data: '',
           success: function(data){
              socket.emit('leaveFromGame',userInfo.userID)
              location.href = '/lobby';
           }
        });
     });
```

./API/view/widgets/game.ejs:

```
<div id="gameTable">
  <% for(let i=0; i<225; i++) { %>
    <div class="cell" id="cell<%= i %>"></div>
  <% } %>
</div>
```

Evvel a beágyazott kóddal legenerálom az amőba játék tábláját és mindegyik cella meg kapja a saját id-jét.

./API/view/widgets/webGL.ejs:

A Three.js api-t használó háromdimenziós óceán háttér komponense. Alapvetően a Three.js API a következő sorrendben épít fel egy három dimenziós környezetet:
Létrehozunk egy kamerát, ennek beállítjuk a pozícióját és a látószögét.

```
camera = new THREE.PerspectiveCamera(60, mapDimensions.width/mapDimensions.height, 1, 20000);
```

Utána beállítjuk a színhelyet. Ez lesz az a háromdimenziós tér amiben az objektumokat elhelyezzük:

```
scene = new THREE.Scene();
```

A színhelynek (scene) beállítjuk a színét, hátterét, adunk neki “ködöt” és hozzáadunk egy fényforrást mert enélkül nem látnánk semmit.
Ezek után beálltjuk a leképzőt (renderer). A leképző felelős azért, hogy hogyan és milyen méretben/formában jelenítse meg a három dimenziós tartalmat.

```
myRenderer();
```

Ezek után a leképzőt hozzáadjuk egy HTML DOM objektumhoz, fontos hozzátenni hogy ez nem úgy működik mintha egy DOM elemet beleraknék egy szülő DOM-ba. Sajnos a Three.js canvas DOM-ja alapvetően nem követi a HTML DOM struktúráját, ezt külön le kell kezelni.

```
canvas.appendChild( renderer.domElement );
```

Ezek után meg kreáljuk a háromdimenziós tartalmat.

```
ocean();
```

A háromdimenziós tartalom három egységből épül fel:

- **geometria**: gyakorlatilag az a háromdimenziós tér, amiben tárolni fogjuk a 3D-s modellünk koordintáit, úgy kell elképzelni mint egy tároló edényt
- **matéria**: ez lehet egy textúra (bármilyen raszteres kép), vagy shader (a videókártya által generált "élő" textúra), a matériát úgy kell elképzelni mintha a bőre lenne a 3D-s modellnek

- háló, azaz angolul **mesh**, ez a három dimenziós koordináta pontok hálója ami gyakorlatilag felépíti a három dimenziós modellt.

```
    geometry = new THREE.PlaneGeometry( 20000, 20000, worldWidth - 1, worldDepth - 1 );
        geometry.rotateX( - Math.PI / 2 );
    const position = geometry.attributes.position;
    position.usage = THREE.DynamicDrawUsage;
    for ( let i = 0; i < position.count; i ++ ) {
      const y = 35 * Math.sin( i / 2 );
      position.setY( i, y );
    }
    const texture = new THREE.TextureLoader().load( '/media/textures/water.jpg' );
    texture.wrapS = texture.wrapT = THREE.RepeatWrapping;
    texture.repeat.set( 5, 5 );
    material = new THREE.MeshBasicMaterial( { color: 0x0044ff, map: texture } );
    mesh = new THREE.Mesh( geometry, material );
```

Ezek után az elkészült háromdimenziós modellünket hozzáadjuk a színhelyhez:

```
scene.add( mesh );
```

Ezuek után animáljuk a leképzőt, amivel elindítjuk a leképzését a három dimenziós tartalomnak.

3.6 A Tools (./API/tools/):

A tools könyvtárban találhatóak azok a kód részletek, amik valamilyen módon kapcsolódnak egyes komponensekhez

./API/tools/amoba.js:

Az amoba.js-ben találhatóak azok a funkciók, amik közvetlenül az amőba játékhoz kapcsolódnak. Két funkció van:

A **checkFive**() funkció:

```
checkFive(row, col, user,session)
```

A **checkfive**() funkció vizsgálja meg hogy vízszintesen - függőlegesen vagy átlósan van e öt db. egymás után álló egyszínű pont lerakva, ha igen a játékos megnyerte a játékot. A tábla egy 15x15 mátrix, amit egy 2 dimenziós tömbben tárolok, Maga a funkció 4 részből áll, a funkciót meghívó paraméterek alapján mind a 4 rész először megnézi hogy hogy az adott irány vizsgálata belefér e még a tábla dimenziójába, azaz nem lóg e túl a táblán, ezek után az adott irányban elkezdi vizsgálni hogy a **user** változóban tárolt szám egymás utám megtalálható ötször amit egy számlálóval tárol (counter) ha a számláló egyenlő lesz öttel

Akkor eltárolja a talált sort egy **amoba**[] nevű tömbben és a **win** nevű **boolean** változót átállítja igazra amit visszatérési értékként visszaküld a funkció. Minden játék egy közös games nevű

objektumban van eltárolva a szerver memóriájában, azon belül játékonként külön - külön rész objektumokban. A struktúra a következő képpen néz ki:

- gamename: a játék neve ami a játékot létrehozott felhasználó adatbázis táblájának id elsődleges kulcsa + kötőjel + a felhasználó nevéből áll össze, így biztosítva hogy nem lehet ugyan olyan nevű játék.
- player1: tömb: játékos neve és id-je
- player2: tömb: játékos neve és id-je
- users: töm: kliens id, kliens neve, játék szoba neve
- full: jelzi, ha tele van szoba, egy szobában maximum két db. játékos lehet
- currPlayer: az aktuális játékos, aki épp lép
- gameState: a játék állapota, **true** ha lehet játszani, **false** ha nem (pl. vége van a játéknak, vagy még várunk egy bejövő játékosra)
- table: tömb, maga a játék tábla modellje tömbben tárolva

```
game = {
    gamename: `${session.userID}-${session.username}`,
    player1: [session.username,session.userID],
    player2: ['',-1],
    users: [],
    full: 0,
    currPlayer: 0,
    gameState: false,
    table: [
    [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
    [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
    [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
    [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
    [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
    [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
    [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
    [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
    [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
    [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
    [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
    [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
    [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
    [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
    [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0] ]}
```

A **proccesWin**() funkció:

```
proccesWin(userIndex,session,type)
```

Ez a funkció két esetben fut le. Ha egy játékos megnyerte a játékot, vagy ez egyik játékos kilépet a játékból annak befejezése nélkül, tehát mindig akkor, ha véget ér egy játék. Ha játékszoba még nincs tele és a játék készítője kilép akkor az adott kliensnek beállítja hogy már nincs játékban és ezt beállítja az adatbázisban is. Ha a játék szoba tele van (2 játékos) akkor attól

függően, hogy kilépés történt e vagy nyerés díjazza a nyertest, ha úgy nyert hogy valaki kilépet akkor csak 5 pontot kap, ha szabályosan megnyerte akkor 20 pontot kap. A kilépés vagy nyerést a funkció meghívásakor a type paraméter jelzi, 0 ha szabályos nyerés és 1 ha kilépés, a **userIndex** paraméter az tömb indexe a fent bemutatott game objektumban található **users** tömbben, a session paraméter pedig az aktuális kliens session-jét adja át, többek között ebben utazik a game objektum is a kliensek felé. A funkció ezen kívül a végén mindent tárol adatbázisban.

./API/tools/ansi.js:
Ez a file csak egy egyszerű objektumot tartalmaz amiben **ansi escape** kód szekvenciák vannak, Ezt arra használom hogy a szerver konzolon történő kiírásai szebb formában történjenek meg.

./API/tools/rooms.js:
Ez a file a Socket.io szoba szerver oldali kezelésével és ahhoz kapcsolódó felhasználók kezelésével foglalkozik. Alapvetően a fent említett **games** objektumon belül a game objektumban tárol mindent:

```
game = {
	gamename: `${session.userID}-${session.username}`,
	player1: [session.username,session.userID],
	player2: ['',-1],
	users: [],
	full: 0,
	currPlayer: 0,
	gameState: false,
	table: [
	[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
	[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
	[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
	[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
	[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
	[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
	[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
	[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
	[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
	[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
	[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
	[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
	[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
	[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
	[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0] ]}
```

- **joinRoomUser(userID,name,room,socketID,firstTime)** funkció a nem játékos szobába való belépéssel foglalkozik, egyrészt tárolja adatbázisban hogy az adott kliens melyik szobába

lép be másrészt a visszatérési értéke egy objektum amiben tárolva van a játékos id-ja, neve, socket id-ja és hogy először csatlakozik e a szobához.

- **playerJoinGame(session)**: ez a funkció az adott játékhoz tartozó game objektumon belül tárolt users objektumban tárolja az aktuálisan a játékszobába belépett játékost. Visszatérési értéke ez az objektum.
- **getCurrentPlayer(id,session)**: kikeresi a a games[‘játék szoba neve’].users-ben tárolt kliensek közül az aktuális játékost.
- **gePlayerIndex(id,session)**: kikeresi a game objektumon belül tárolt **users** objektumban az aktuális játékos index-ét.
- **playerLeaveGame(id,session)**: ha a játékos elhagyja az aktuális szobát akkor a game.users-ből törli a játékost.
- **getGamePlayers(room,session)**: kikeresi a játékosokat az aktuális játék szobában.
- **roomHistory(room,io)**: az adatbázis chat táblájában tárolt beszélgetéseket olvassa be és egy socket.io emit parancsal kiküldi az aktuális szoba chat ablakába.
- **formatTime()**: az aktuális időt adja meg visszatérési értékként **string** formában, két formázott string egy hosszabb és egy rövidebb formája az idő kijelzésének.
- **formatMessage(username, text)**: a chat ablakban megjelenő üzenetet formázza meg, visszatérési értéke egy objektum aminek a felépítése: [felhasználó neve],[az aktuális üzenet],[a formázott idő]

./API/tools/socketIO.js: a Socket.io fő kezelési funkciói találhatóak itt

- **io.sockets.on('connection', (socket)**: mikor a kliens csatlakozik az adott sockethez a session változóban eltároljuk az aktuális http kérés session-jét, ezen kívül a socket id-jét is.
- **socket.on('joinToRoom', ():** mikor a játékos csatlakozik egy sima szobához:
    - kiírja az aktuális szoba chat ablakába az adatbázisban tárolt beszélgetések előzményeit
    - a user változóban tárolja az aktuális felhasználó adatait (egyedi azonosító id, felhasználó neve, az aktuális szoba neve, az aktuális socket id azonosító és hogy először csatlakozott az aktuális szobához)
    - a socket io val ezek után beléptetjük a felhasználót a szobába
    - a /lobby végponton található játékok listáját frissíti kliens oldalon (emit.(updateLobby))
    - elküldünk egy privát rendszer üzenetet a chat ablakba, amivel üdvözöljük az aktuális klienst
    - és elküldünk egy publikus üzenetet, amivel jelezzük a többi felhasználónak, hogy a kliens csatlakozott a szobához
- **socket.on('joinToGame', ()**: mikor a játékos csatlakozik egy játék szobához
    - kikeressük az aktuális játék game objektumát
    - megnézzük, hogy tele van e már a játék szoba
    - amig nincs tele tároljuk a belépett kliensek adatait az aktuális game objektumban
    - amikor tele lesz a szoba töröljük a szoba bejegyzését a /lobby végponton megjelenített játék listából, ergo többen már nem tudnak csatlakozni a játékhoz (maximum 2 játékos)

    - frissítjük a /lobby végponton található játék listát
    - véletlen szám generátorral legeneráljuk, hogy ki fogja kezdeni a játékot (1 vagy 2)
    - beállítjuk a véletlen szám alapján a game objektumban található **currPlayer** mezőt
    - a game objektumban található **gameState** mező **boolean** értékét igazra állítjuk, ez felelős azért hogy a játék mezőn lehet e kattintani, azaz lehet e játszani vagy nem
    - beléptetjük a klienst a játék szobába
    - az aktuális szobában jelezzük, hogy elindult a játék
    - kiírjuk az aktuális szoba chat ablakába az adatbázisban tárolt beszélgetések előzményeit
    - ismételten beléptetjük a klienst, ez ugye abban az esetben fontos, ha játék közben a játékos átvált egy másik szobára
    - elküldjük a **users** objektumban bejegyzett kliens indexét (games[játék neve].users) a kliensnek (socket.emit('UserIndex', gePlayerIndex(session.userID,session)+1);)
    - frissítjük az aktuális játékszobát kliens oldalon (updateGameRoom)
    - frissítjük az adatbázis **rooms** táblájában a kliensekhez tartozó sor szoba és játék oszlopait
    - üdvözöljük a játékost a szobában
    - a szoba többi kliensének chat üzenetben jelezzük, hogy a kliens belépet a játékba
- **socket.on('logoutFromGame', (id)**: ha a játékos játék közben elhagyja a szervert (logout)
    - megszerezzük a játékos indexét
    - a játék státuszát beállítjuk **false**-ra azaz nem lehet klikkelni a játék táblán
    - elküldunk egy üzenetet a szoba többi kliensének, hogy a játékos elhagyta a szobát
    - a rooms.js ben található proccesWin() funkcióval kiértékeljük a nyertest (aki bent maradt a szobában kap 5 pontot)
    - töröljük a kilépett játékost a users objektumból
    - elküldünk a kliens oldalnak egy ‘gameAborted’ üzenetet, ami a kliens oldalon visszalépteti a bent maradt játékost a /lobby végpontra és evvel együt kilépteti ezen klienst az aktuális játékból, a szoba ilyenkor bezáródik és megsemmisül
- **socket.on('leaveFromGame', (id)**: majdnem ugyanaz mint a fenti **'logoutFromGame'** avval a különséggel hogy a kilépett játékos és a bent maradt játékos visszakerül a /lobby végpontra
- **socket.on('message', (msg)**: elküldünk egy formázott üzenetet az aktuális szoba chat ablakába
- **socket.on('createGame', ()**: mikor a játékos a /lobby végponton kreál egy játékot
    - elkészítjük az aktuális játék game objektumát, amiben a játékhoz szükséges adatokat kezeljük (játékosok, játék státusza, játék tábla stb.)
    - eltároljuk az első játékos (aki kreálta a játékot) adatait ebben az objektumban
    - jelezzük a kliens oldal felé a /lobby végponton, hogy elkészült a játék ahol a kliens oldal megjeleníti a játék listában a játék nevét
- **socket.on('putCell', (id)**: ez a kliens oldal által aktiválódik mikor egy játékos az amőba játék táblán kirak egy pontot
    - megkeressük az aktuális játék indexét, az aktuális klienst, aki épp kattintott a játék táblán és eltároljuk az aktuális játék game objektumában

- a paraméterként kapott id alapján kiszámoljuk melyik sor és oszlopra klikkelt a kliens és a játék táblában tároljuk ezen sor és oszlopban a játékos id-jét (1 vagy 2)
- visszaküldjük a kliens oldalra az üzenetet, hogy a táblán hova rajzoljuk be az a pontot 'drawCell'
- megvizsgáljuk hogy történt egy nyerés a **checkFive**() funkcióval
- ha nyert valaki a játék státuszát átrakjuk **false**-ra (nem lehet kattintani a táblán)
- töröljük a játékot
- küldünk egy üzenetet a kliens oldal felé, hogy nyert valaki
- a kliens oldalon majd ezt feldolgozzuk, eltároljuk adatbázisban és a játékosokat visszaléptetjük a /lobby végpontra

4. Kliens oldal ("frontend"):

A kliens oldali rész kódjai és egyéb fájlok az ./assets/public mappában találhatóak. A **public** mappa nyilvános bármilyen a szerverre felcsatlakozó kliens számára elérhető. Három részre van osztva:
A **css** könyvtár, ami tartalmazza pár DOM objektum stílus formázását, a **js** könyvtár, amiben a kliens oldalon futtatott JavaScript kódok találhatóak és a **media** könyvtárat, amiben lényegében csak képek vannak.

A stílus formázásokat különösebben nem részletezem mivel egyszerűek és értelmezésük adja magát:

- login.css: a weboldal bejelentkezési oldalához tartozó stílus formázások
- wrapper.css: egy a fő szülő formázási stílus, ez öleli körbe és tartalmazza többit
- widgets: egyes kisebb DOM komponensek stílus formázása, amit célszerű volt külön rakni és nem közvetlenül az view-ben található EJS sablon állományokba mivel ezeket sok más helyen is felhasználom
    - buttons.css: a weboldalon megjelenő gombok stílusa
    - gamelist.css: a /lobby végponton található játék lista stílus formázása
    - scrollbars.css: a weboldalon található görgető sávok formázása, egyellőre csak Google Chrome leképző motorját használó böngészők támogatják (Blink engine), például a Miscrosoft Edge is.

4.1 A kliens oldali javascript kódbázis:

./assets/public/js/quarto/landing/game.js

A játék kliens kódja. Itt található az összes olyan kód részlet, ami játékkal kapcsolatos.

```
$( document ).ready( INIT() );
```
JQuery funkció, mikor az aktuális oldal DOM-ja teljesen betöltődött elindítja az INIT() funkciót.
```
initChat();
```
Inicializáljuk a chat felületet. Ennek részleteit majd a chat widget tárgyalásánál részletezem.
```
player1 = document.querySelector('#player1');
player2 = document.querySelector('#player2');
```
Beállítom a JavaScript számára a két játékos nevét tartalmazó DOM elemet.
```
 for(let i=0; i<225; i++) {
      document.querySelector(`#cell${i}`).addEventListener('click', (event)=>{
          setPos(i);
          });
}
```
A #cell[0-255] DOM elemeknek adok egy egér kattintás esemény figyelőt.
```
socket.emit(`joinToGame`);
```
Elküldök a szervernek egy üzenetet, hogy csatlakoznék a szobához. Ezt a szerver feldolgozza és beenged a szobába.
```
socket.on(`joinedToGame`, (msg)=>{
      if (msg !== "") {
         outputMessage(msg,chat_display);
      }
   });
```
Ha a szerver beengedett a szobába válaszol egy **‘joinedToGame’** üzenettel és a chat ablakban kapunk egy rendszer üdvözlő üzenetet.
```
socket.on('updateGameRoom', (game)=>{
      updateGameRoom(game);
   })
```
Frissítjük a játék szobát, azaz ha például a táblán már vannak kirakva pontok és éppen visszacsatlakozok a szobához vagy újra töltöm a böngészőben az adott oldalt, akkor a szoba tartalmát visszatöltöm az updateGameRoom() funkcióval.
```
socket.on('UserIndex', (index)=>{
      userindex = index;
   })
```
Ezek után a szerver beállítja azaktuális **userindex**-et, ez fogja jelezni hogy az aktuális játékos hányas index számmal rendelkezik (1 vagy 2)
```
   socket.on('drawCell', (id, userNr)=>{
      let currentCell = document.getElementById('cell'+id);
      currentCell.classList.add('takeP'+userNr);
      if (userNr == 1) {
         currentPlayer = 2;
      } else {
         currentPlayer = 1;
      }
      displayCurrentPlayer();
    });
```

Ez a szervertől kapott üzenet akkor jön, ha a másik játékos lépést hajtott végre a táblán és a szerver ezt elküldi és kirajzolja ezen a kliensen. Ezek után váltok a másik játékosra, amit a currentPlayer változoban tárolok.

```
socket.on('gameStarted', (rnd,game)=>{. . . .
```

Ha elindult a játék a szervertől kapok egy üzenetet, hogy megtörtént ez az esemény, az **currentPlayer** változóban megkapom a szerveren véletlenszerűen meghatározott kezdő játékost (1 vagy 2).
A **userindex** változót beállítom a jelenlegi játékosra, azaz a szervertől kapott véletlen számra. A játék státusát átkapcsolom játszhatóra. A displayCurrentPlayer() funkcióval kijelzem a DOM-ban hogy ki az aktuális játékos és a játék státusz mezejében kijelzem hogy elindult a játék.

```
socket.on('win', (winner,state)=>{. . .
```

Ez az üzenet akkor jön a szervertől ha valaki megnyerte a játékot, státusz mezőben kiíratom hogy nyerés történt és a renderwin() funkcióval megjelenítem a 'modal' ablakot ami ugyanezt kijelzi egy ok gombbal egyetemben amire kattintva visszatér a játékos a /lobby végpontra.

```
socket.on('gameAborted', ()=>{. . . .
```

Ezt az üzenetet a szerver akkor küldi, ha valaki kilépett a játékból. Ez egy **Ajaxos** művelet, ami egy POST kérelmet küld a /lobby végpont felé ami törli a session-ben a játékszoba bejegyzést. Ezek után átirányít a /lobby végpontra.

A funkciók:

```
setPos(id)
```

Ez a funkció akkor fut le, ha egy játkos rákattint a táblán lévő cellák egyikére. Ha a játék státusza nem játszható (pl. vége van a játéknak vagy még várunk egy másik játékos csatlakozására) akkor visszatér anélkül hogy csinálna bármi mást (if (gameState == false)), ha az aktuális kliens (**userindex**) lép éppen (currentPlayer == userindex) akkor elküldjük a szervernek hogy léptünk és másik játékos kliensére a szerver küldje el a fent tárgyalt 'drawcell' üzenetet ami azon a kliensen is kirajzolja az aktuális pontot a táblán. Ha mégse ez a kliens lép kiírja, hogy a másik játékos van soron.

```
renderStatus(msg,timeout)
```

A játék státusz mezejében kiírja az aktuális játékhoz kapcsolódó üzenetet (ki lép, várunk a egy játékosra stb..)

```
updateGameRoom(game)
```

Frissíti a játékszoba állapotát, a játékos neveket (#player,#player2), ha még nem csatlakozott másik játékos a státuszba kiírja hogy várunk még egyjátékosra. A szerveren tárolt és itt paraméterként megkapott game objektumból beállítja az aktuális játékost, aki lép, a játék státuszát (lehet e játszani vagy nem) és a game objektum táblájából frissíti a kliens játék tábláját.

```
displayCurrentPlayer()
```

Kijelzi a játék státusz mezejében hogy éppen ki lép, és az aktuális játékos nevét jelző DOM elemnek add egy **css** animációt.

```
renderWin(winner)
```

Ez a funkció akkor indul el, ha valaki nyert. Kreál egy 'modal' DOM elemet amin kijelzi hogy ki nyert és az alján lévő OK gombhoz hozzárendel egy **Ajax** POST műveletet ami törli az aktuális

kliens session-ben tárol játékszoba nevét (req.session.game) azaz így már véglegesen nincs játékban a kliens és a végén átirányítja a klienst /lobby végpontra.

./assets/public/js/quarto/landing/lobby.js

A weboldalon a /lobby végpont egyfajta gyűjtő szerepet kap, a frissen bejelentkezett kliensek ide csatlakoznak be először vagy ha vége van egy játéknak akkor ide irányít vissza a szerver. A /lobby felépítése két részre osztható, a játék listára, ami fent vagy bal oldalt helyezkedik el a képernyő mérettől függően és a chat ablakra.

```
$( document ).ready( INIT() )
```

Hasonlóan a game.js-ben látottakhoz itt is van egy **JQuery**-s init funkció ami elindítja benne lévő kódot ha az oldal teljesen betöltődött. Ugyanúgy, mint a game.js-ben először inicializáljuk a chat ablakot.

```
  $(".create-game-form").submit(function(e) {
    e.preventDefault();
    $.ajax({
      url: "/game",
      type: "POST",
      data: {'gameroom': `${userInfo.userID}-${userInfo.name}`},
      success: function(data){
        socket.emit('createGame');
        location.href = '/game';
      }
    });
  });
```

Ezek után **JQuery**-vel a .**create-game-form** osztályú DOM elemhez hozzáadok egy **Ajaxos** POST műveletet ami egyrészt elküldi a /game végpontra az aktuális felhasználó adatait (userID, name),ha ez sikeres volt elküld egy üzenetet a szervernek hogy játékot akarunk kreálni és átirányit a /game végpontra.
Ez persze csak akkor történik meg ha kliens rá kattint erre gombra.

```
socket.emit(`joinToRoom`);
```

Csatlakozunk a szobához.

```
  socket.on(`joinedToRoom`, (msg)=>{
    if (msg !== "") {
      outputMessage(msg,chat_display);
    }
  });
```

Ha sikeres volt a csatlakozás a szervertől megkapjuk a választ ez megtörtént és egy üdvözlő szöveget kiíratunk a chat ablakba.

```
socket.on('gameCreated', (gamesList)=>{
    updateGameList(gamesList,gameListDOM,socket);
  });
```

Ez az üzenet akkor jön a szervertől, ha valaki csinált egy játékot, ilyenkor frissíti a játék listát.

4.2 A kliens oldali widgetek azaz kisebb komponensek kódja.

./assets/public/js/quarto/widgets/chat.js

A chat komponens mind a /lobby és a /game végponton is fel van használva. Ez kód hozzátartozik a chat komponenshez és ellátja a szükséges alap funkciókat.

```
export const chat_display = document.querySelector('.chat-display');
const chatInputBox = document.querySelector('.chat-input-box');
const chatInputButton = document.querySelector('.chat-input-button');
```

Deklaráljuk a **chat** komponens DOM elemeit.

```
initChat()
```

Az **initChat**() funkció inicializálja a **chat** komponenst.

```
  chatInputButton.addEventListener('click', (event)=>{
     sendMessage(chatInputBox,socket); });
  chatInputBox.addEventListener("keyup", (event)=> {
     if (event.key === 'Enter') { sendMessage(chatInputBox,socket);} });
```

A chat komponensben található chatInputBox és chatInputButton hozzáadunk két eseményfigyelőt, az egyik az enter lenyomását figyeli a másik, hogy egérrel kattintottunk e a gombon. Mindkettő kiírja azt az üzenetet a chat ablakba amit beírtunk a chatInputBox-ba.

```
socket.on('message', (msg)=>{
```

A szervertől kapott üzenet ami akkor kapunk ha valamelyik másik kliens üzenetet írt, ezt az **outputMessage**() funkcióval jelenítjük meg chat ablakban.

```
socket.on('chat-history', (data)=>{
```

Mikor a szerver elküldi a chat előzményeket, amit adatbázisban tárolunk

```
socket.on(`updateLobby`, (games)=>{
```

Frissíti a /lobby végpont egyes tartalmát, ezt berakhattam volna a lobby.js-be mivel hozzá tartozik, de mivel a ugyan az a chat komponens van felhasználva a játék szobákban is ezért egyszerűbb és tisztább érzés.

A funkciók:

```
outputMessage(msg)
```

Kreál egy **html** DOM elemet, amiben formázva van tárolva a chat üzenet és ezt hozzáadja minden üzenet küldéskor a chat ablakhoz. Azaz megjeleníti az üzenetet a kliens oldalon.

```
sendMessage(DOMelement,socket)
```

Elküldi az üzenetet a szervernek, ami majd továbbítja a többi kliensnek.

```
displayChatHistory(data)
```

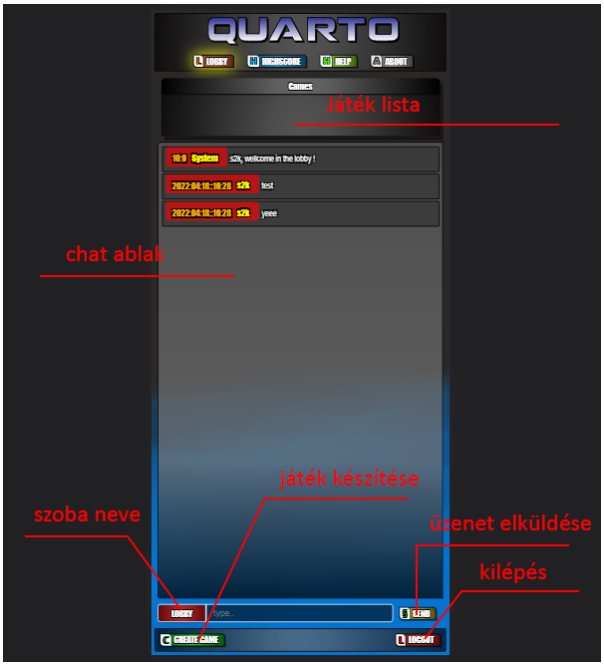
Szintén kreál egy **html** DOM elemet. Ez fogja tartalmazni a szervertől kapott chat előzményeket és hozzáadja a chat ablakhoz.

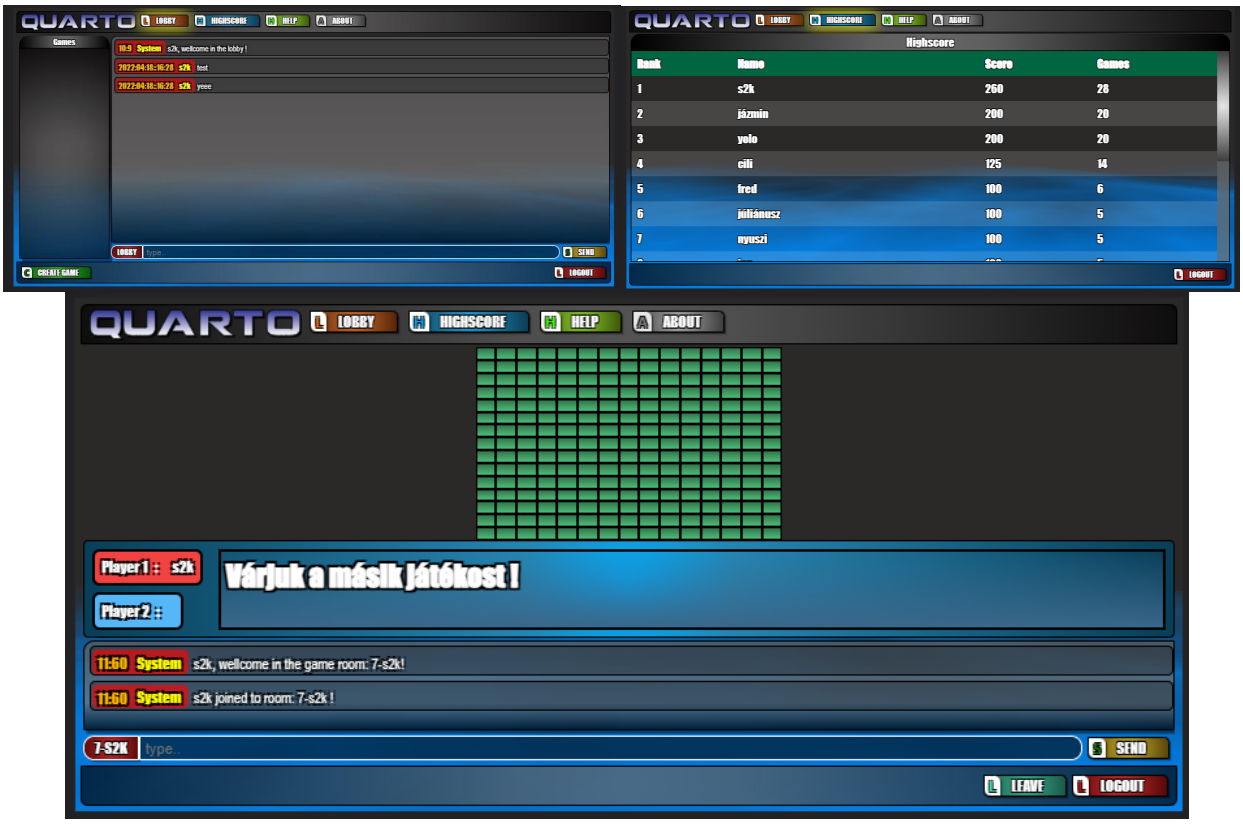
./assets/public/js/quarto/widgets/gamelist.js
A játék lista komponens, ami tartalmazza a játkosok által kreált játékok listáját.

A funkciók:
updateGameList(gamesList,DOMelement)
A játék lista kliens oldali frissítése. A paraméterként a szervertől kapott gamesList tömb alapján frissíti ezt a komponenst. Ha a játékos épp játékban van (if(userInfo.game != 'null')) akkor nem frissíti az előzőekben felsorolt okok miatt nem is látja a játékos a listát (ne lépkedhessen át másik játékokba játék közben stb.). Ezek után elkészíti a DOM elemet, gyakorlatilag egy mini HTML űrlapot amihez **Ajax** művelettel hozzáadunk egy HTTP POST kérelmet ami elküldi a gombon kattintott játékos adatait a /game végpontra, ekkor a játékos session-jében a POST művelet által be lesz állítva hogy melyik játék szobába jelentkezett és ezután átirányítja erre /game végpontra játékost, bejelentkezteti session.game-ben a POST kérelem alapján beállított szobába és ott azonnal elkezdődik a játék mivel aki innét megy be a játékba az már a második játékos.

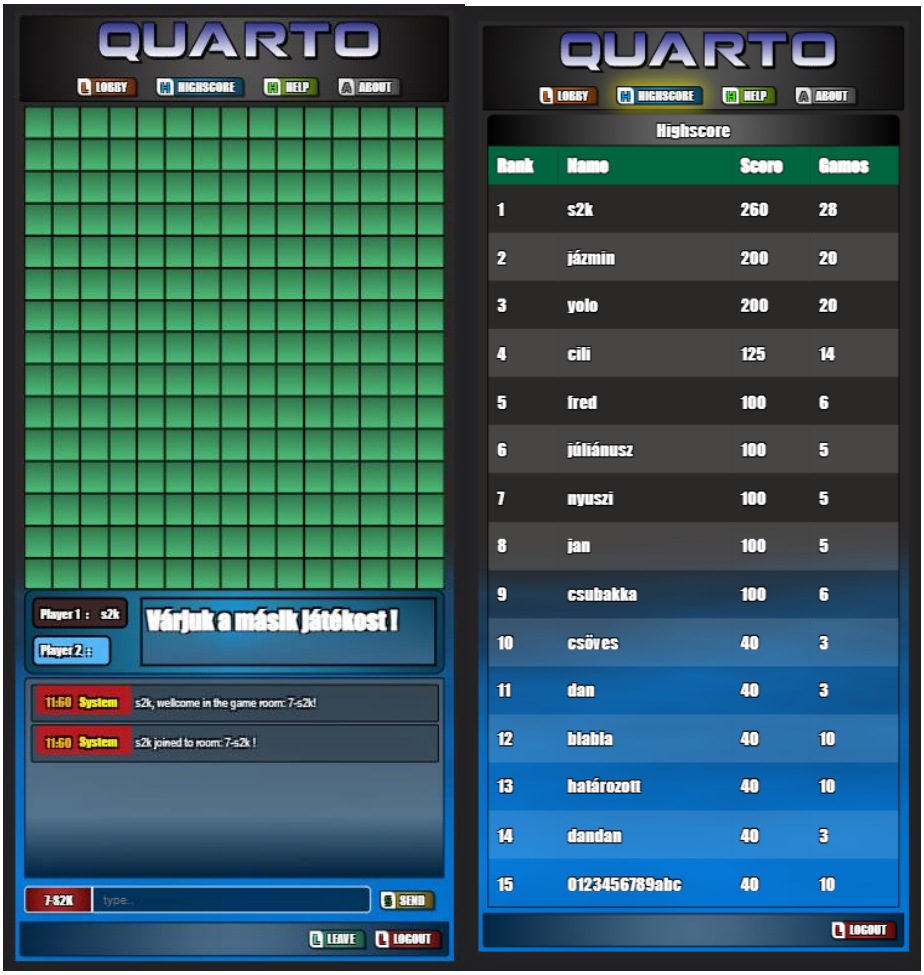
5. A layout azaz a weboldal elrendezése

A layout alapvetően reszponzív viszont elsősorban az álló mobil nézetet preferálja, ezen mutat a legjobban a játék tábla jellegzetessége miatt.




QUARTO
LOBBY
HIGHSCORE
HELP
ABOUT
Games
10:9 System s2k, wellcome in the lobby !
2022:04:16:16:28 s2k test
2022:04:16:16:28 s2k yeee
LOBBY type..
SEND
CREATE GAME
LOGOUT
Highscore
Rank
Name
Score
Games
1 s2k 260 28
2 jázmin 200 20
3 yolo 200 20
4 cili 125 14
5 fred 100 6
6 jolánusz 100 5
7 nyuszi 100 5
LOGOUT

QUARTO
LOBBY
HIGHSCORE
HELP
ABOUT
Player 1 : s2k
Player 2 :
Várjuk a másik játékost !
11:60 System s2k, wellcome in the game room: 7-s2k!
11:60 System s2k joined to room: 7-s2k !
7-S2K type..
SEND
LEAVE
LOGOUT



6. Az amőba játék

Az amőba kétszemélyes absztrakt stratégiai táblás játék, a **gomoku** változata. A győzelemhez (legalább) öt bábunkat kell egy vonalba lerakni. A standard **gomokutól** eltérően (amelyben az 5-nél hosszabb sor nem nyer) az amőbában különleges virtusnak számít a hosszabb (legalább 6 bábut tartalmazó) győztes sor kirakása.
Az amőba elméletileg a táblás játékok közé tartozik, de a valóságban jellemzően egy füzet kockás papírján, ceruzával, tollal, gyerekek szokták játszani. A tábla mérete nem kötött, a játék a füzetlap széléig, a még szabad területen folyik.
A játék a nevét onnan kapta, hogy a győztesé az a „megtiszteltetés”, hogy a végül létrejött alakzatot (szögletes vagy görbe vonallal) szorosan körbe rajzolja. Az így létrejött vonal egy állábakat növesztett amőbára hasonlít.

Játékmenet

A játék nem sokban különbözik a gomokutól, csak nincs rögzítve a tábla mérete, és így nem is kerülhet sorra a bábuk tologatásával játszott második játékszakasz.

A két játékos felváltva tesz egy-egy bábut a táblára. A játék célja, hogy vízszintes, függőleges vagy átlós irányban megszakítás nélkül öt saját bábut sikerüljön letenni. Az ellenfél ezt a kialakulni látszó vonal végére tett bábukkal próbálja megakadályozni.

A papíron játszott, legjellemzőbb változatban az egyik játékos X, a másik O jelekkel helyettesíti a bábukat, a játék OX elnevezése innen ered, bár azt a Tic-tac-toe-ra is használják. A kétféle jelet szokták kétféle színnel is jelölni, ahogy az a fényképen is látható.