**Online Quarto**

**Az applikáció programozási specifikációja**

Az applikáció alpavetően a **Node.js** javascript alapú és szerver oldali futtatási környezet felhasználásával készült. Alapvetően szerver oldali renderinget használok (SSR) evvel kiváltva a frontend teljsen külön való fejlesztését és ennek köszönhetően pár elönyös tulajdonságot elérve:

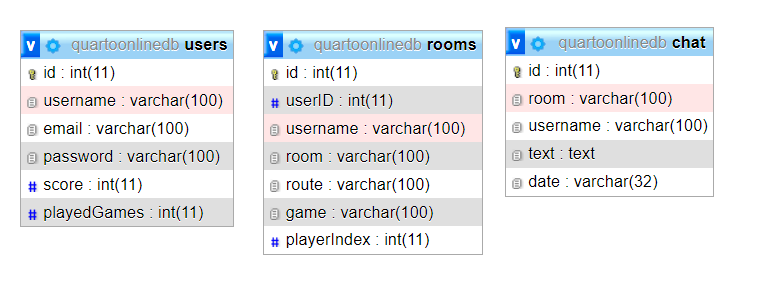
* A szerveroldali leképzés lehetővé teszi az oldalak gyorsabb betöltését, javítva ezzel a felhasználói élményt.
* Szerveroldali megjelenítéskor a keresőmotorok könnyen indexelhetik és feltérképezhetik a tartalmat, mivel a tartalom még az oldal betöltése előtt renderelhető, ami ideális SEO számára
* A 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 websocket miatt.

A **Socket.IO** egy olyan könyvtár, amely alacsony késleltetésű, kétirányú 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 újracsatlakozás.

Minden az oldalra bejelentkezett felhasználó egyrészt egy session-be kerül másrészt a felhasználó és a szerver között ilyenkor kiépül egy websocket kapcsolat amin keresztül történik egyrészt a weboldal chat részének a kommunikációja másrészt a két játékos ezen a socket-en keresztül kapcsolódik a játékhoz és az abban történt események ezen a socketen keresztül lesznek szállítva.

Az **adatbázis** és az adatok tárolása **MySql** segítégével van megoldva. Minden a chat abalakokban történt esemény, és minden a játékkal kapcsolatos eredmény vagy a a felhasználók szerveren való közlekedése adatbázisban van eltárolva. Gyakorlatilag három táblával sikerült megoldanom 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 szálított de a users táblábol 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

**Szerver oldal (“backend”):**

**server.js**:

A server.js az alap szerver indító forrás fileja, különböző a nodejs 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 NodeJS alapítvány fejlesztett ki és tart fenn.

Minimális felületet biztosít az alkalmazások készítéséhez. Az ExpressJS ezen kívül eszközöket ad nekünk, amelyekre szükség van az alkalmazás felépítéséhez. Az ExpressJS szintén rugalmas, mivel léteznek különféle modulok, amelyeket npm-en bocsátanak rendelkezésre, és amelyek közvetlenül be vannak dugva, azaz az Express.

**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 újrafelhasználásának kezeléséért. Sorban tartja a függőben lévő kéréseket egy adott gazdagéphez és porthoz, 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 modelt

**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, kétirányú 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.

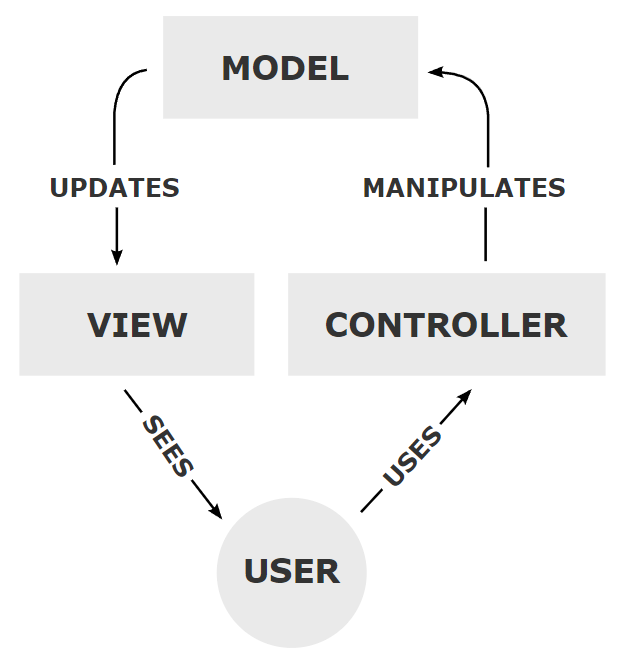
* TWO\_HOURS: változó beállítása, a cookie-k az ebben a változóban beállított idő intervallum után semmisülnek meg.
* inicializáljuk az express modult: app = express();
* inicializáljuk a munkamenet köztes szoftvert (sessionMiddleware)
  + **secret**: egy véletlenszerű egyedi karakterlánc-kulcs, amelyet 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.
* hozzáadjujk az applikációhoz a sessionMiddleware-t (app.use(sessionMiddleware);)
* urlencoded: a bejövő http kérések url kódólásának elemzése middleware
* beállítjuk a publikus mappa elérési útvonalát (app.use(express.static('./assets/public'));)
* beállítjuk és inicializáljuk az ejs template 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

**Az API**:

Az aplikációs interface 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 aplikációs végpontok viselkedését iránítja, pl.: különböző HTTP kérésekre hova szállítsa el a felhasználót az oldalon vagy hova küldjön adatokat. Ezeket a router (útvonal választó) fogja őssze a **router.js** (./API/router.js) file-ban.

A router.js fileban található szintatika a következő képpen 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 müvelettel elviszi a klienst a ‘/’ útvonallra azaz a szerver gyökerébe. A **router.post('/register', auth.POST\_register);** egy HTTP POST müvelettel elküldi a felhasználó regisztrációs adatit 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 kényelmessen vissza rakja a klienst a siten legutoljára járt útvonalra.

**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 aplikáció szerver oldali renderinget használ ezért a 2 kontrol fileban található kontrolok nagy része az EJS template 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 oldara és megjeleníti a html tartalmat.
* POST\_login: a login oldalon a bejelentkezési form-ban 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 adtbázisból hogy melyik szobában volt aztán törli a session-t és a cookiet. Végül visszavezérel a ‘/’ gyökér útvonalra.

Funkciók:

* sendError(msg,res,route): az EJS template nyelv által a html dokumnetumban beágyazot **errorMsg** változóba beírja a funkció paramétereként megadott hibaüzenetett ha a bejelentkezési/regisztrálási adatok hibásak.
* db\_reggister(name,email,passwd1,req,res): mySql adatbázisban tárolja az újjonan 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 klients 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.

**A Model (./API/model/):**

Itt alapvetően az adatbázis model-t álltom be, mivel más modelre jelen esetben nincs szükség. Az adatbázis MySql alapú.

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

db: a db objetum 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 defibniált felhasználói adattal.

**A View (./API/view/):**

Ebben a könyvtárban találhatóak ez ejs modul által felhasznált html template 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 utánm megérkezünk
* login: az auth.js control-hoz kapcsolódó html ejs templatek
  + login.ejs: a bejelentkezési oldal
  + register.ejs: a regisztrációs oldal
  + root.ejs: az oldal gyökér végponja ide érkezik meg mindenki aki az oldal ala url-jét beírja a böngészőbe
* widgets: itt kisebb ejs html sablonok találhatók amiket beágyazok az oldallakra mint rész komponensek
  + chat.ejs: az oldalon található chat komponens
  + footer.ejs: minen egyes a weboldalon megatalálható HTTP GET művelettel elérhető végpontnak van egy külön láb léce.
  + game.ejs: a játék html sablonja
  + gamelist.ejs: a /lobby végponton található játék lista, ami lista szerűen jelzi ha valakik kreáltak játékot
  + header.ejs: minen 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 bejelenkezés után a fejléc komponesben 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 frameworkot használó komponens, ez jeleníti meg a háttérben három dimenzióban hullámzó tengert.

Mivel ezek a sablonok tönnyire rendkívűl rövidek és egyszerű HTML CSS leíró nyleveket 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** websocket 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 kíratom 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. Ezut 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 mezejének 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 objetumot 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 kijelenkezik a játkos vagy csak kilép a játékból vissza a /lobby végpontra. Az ajax művelet végén ki “emitel” egy socket.io üzenetet 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 három dimenziós oceán háttér komponense. Alapvetően a Three.js api a következő sorrendben épít fel egy három dimenziós környezetet:

Létrehozunk egy kamerát, ennek beállítjuk a pozicióját és a látószögét.

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

Utána beállítjuk a színhelyet. Ez lesz az a három dimenziós 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 mntha egy DOM elemet beleraknék egy szülő DOM-ba. Sajnos a Three.js canvas DOM-ja alapvetően nem követi a HTML DOM strukturáját, ezt külön le kell kezelni.

canvas.appendChild( renderer.domElement );

Ezek után meg kreáljuk a három dimenzios tartalmat.

ocean();

A három dimenziós tartalom három egységből épül fel:

* **geometria**: gyakorlatilag az a három dimenziós 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 modelnek
* háló azaz angolul **mesh**, ez a három dimenziós koórdinta pontok hálója ami gyakorlatilag felépíti a három dimenziós modelt.

        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 három dimenziós modellunket hozzáadjuk a színhelyhez:

scene.add( mesh );

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

**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 vízszintessen - függőlegessen vagy átlósan van e 5 db egymás után álló egyszínű pont lerakva, ha igen a játékos megnyerte a játékot. A tábla egy 15x15 mátrix amit egy 2 dimenziós tömben tárolok, Maga a funkció 4 részből áll, a funkciót meghívó paraméterek alapján mind a 4 rész először megnézi hogy hogy az adott irány viszgálata 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 tömben és a win nevű boolean változót átállítja igaz-ra amit viszsatérési értékként visszaküld a funkció. Minden játék egy nagy games nevű objetumban 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 primary kulcsa + kötöjel + a felhasználó nevéből áll őssze, 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 max. 2 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 modelje tömben 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] ]}

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 kliens nek 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 dijjaza 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 meghivásakkor 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 (többnyire logolás ) 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 mndent:  
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 elöször 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 obektumon 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 visszatéréi értékként string formában, két formázott string egy hosszabb és egy rövidebb formálya az idő kijelzésének.
* **formatMessage(username, text)**: a chat ablakban megjelenő üzenetet formázza meg, visszatérési értéke egy objektum aminke 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 socket-hez 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 elöször 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) )
  + elküldunk egy privát rendszer üzenetet a chat ablakba amivel üdvözöljuk 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 csatlakkozik eghy 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 obketumban
  + amikor tele lesz a szoba töröljuk a szoba bejegyzését a /lobby végponton megjelenített játék listából, ergó 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 állitjuk, ez felelős azért hogy a játék mezőn lehet e kattintani, azaz lehet e játszani vagy nem
  + beléptetjuk 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);)
  + frisítjük az aktuális játékszobát kliens oldalon (updateGameRoom)
  + frissítjük az adatbázis rooms táblájában a klienshezhez 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)
  + töröljuk a kilépett játékost a users objektumból
  + elküldünk a kliens oldalnak egy ‘gameAborted’ üzenetett ami a kliens oldalon visszalépteti a bent matadt játékost a /lobby végpontra és evvel eggyüt kilépteti ezen klienst az aktuális játékból, a szoba ilyenkor bezáródik és megsemmisül
* **socket.on('leaveFromGame', (id)**: madjnem ugyanaz mint a fenti **'logoutFromGame'** avval a különséggel hogy a kilépett játékos és a bentmaradt játékos visszakerül a /lobby végpontra
* **socket.on('message', (msg)**: elküldünk egy formázott üzenetett 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 aktíválódik mikor egy játékos az amőba játék táblán kirak egy pontot
  + megkeressük az aktuális játék indexét, az aktuális klienst aki épp clickelt a játék táblán és eltároljuk az aktuális játék game objektumában
  + a paraméterként kapott id alapján kiszámoljuk melyik so 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 adatzbázisban és a játékosokat vissza léptetjük a /lobby végpontra

**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 oldaon futtatott javascript kódok találhatóak és a **media** könytárat 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 template fájlokba 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, geyellőre csak Google Chrome leképző motorját használó böngészők támogatják (Blink engine), például a Miscrosoft Edge is.

**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');

Beállítiom 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 hogycsatlakoznék 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 ésa chat ablakban kapunk egy rendszer üdvözlő üzenetet.

socket.on('updateGameRoom', (game)=>{

        updateGameRoom(game);

    })

Frisítjük a játék szobát, azaz ha például a táblán már vannak kirakva pontok és éppen visszacsatlakozok a szobához vagy ujratöltöm a böngészöben az adott oldalt, akkor a szoba tartalmát visszatöltöm az updateGameRoom() funkcióval.

socket.on('UserIndex', (index)=>{

        userindex = index;

    })

Ezek után a szever beállítja azaktuális userindex-et, ez fogja jelezni hogy az aktuális játékos hanyas 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 véletlenszerüen meghatározott kezdő játékost (1 vagy 2).

A userindex változót beállítom a jelenlegi játékosra azaz aszervertől kapott véletlen számra. A játék státusát átkapcsolom játszahatóra. 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() fukcióval megjelenítem a ‘modal’ ablakot ami ugyanezt kijelzi egy ok gombal egyetemben amire kattinva visszatér a játékos a /lobby végpontra.

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

Ezt az üzenetet a szerver akkor küldi ha valaki kilépett a játékból. Ez egy ajaxos művbelet 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 eggyikére. Ha a játék státusza nem játszaható (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) akor 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 aktulis pontot a táblán. Ha mégse ez a kliens lép kiírja hogy a msáik játékos van soron.

 renderStatus(msg,timeout)

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

updateGameRoom(game)

Frisítí a játékszoba állapotát, ajátékos neveket (#player,#player2), hamég nem csatlakozott mésik játékos a státuszba kirírja hogy várunk még egyjátékosra. A szerveren tárolt és itt paraméterként megakapott game objektumból beállítja az aktuális játékost aki lép, a játék státuszát (jlehet e játszani vagy nem) és a game objektum táblájából frisíti a kliens játék tábláját.

displayCurrentPlayer()

Kijelzi a játék státusz mezejébe hogy éppen ki lép, és az aktuális játékos nevét jelző DOM elem-nek add egy css animációt.

renderWin(winner)

js:

quarto

landing:

game.js

socket io:

socket.emit(`joinToGame`);

socket.on("connect", () => {});

socket.on("disconnect", () => {});

socket.on(`joinedToGame`, (msg)=>

socket.on('updateGameRoom', (game)=>

socket.on('UserIndex', (index)=>

socket.on('drawCell', (id, userNr)=>

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

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

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

functions:

INIT()

initChat()

setPos(id)

renderStatus(msg,timeout)

updateGameRoom(game)

displayCurrentPlayer()

renderWin(winner)

lobby.js:

socket io:

socket.emit(`joinToRoom`);

socket.on("connect", () => {});

socket.on("disconnect", () => {});

socket.on(`joinedToRoom`, (msg)=>

socket.on('gameCreated', (gamesList)=>

functions:

INIT()

initChat()

widgets:

chat.js:

socket io:

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

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

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

functions:

chatInputButton.addEventListener('click', (event)=>

chatInputBox.addEventListener("keyup", (event)=>

outputMessage(msg)

sendMessage(DOMelement,socket)

displayChatHistory(data)

gamelist.js

functions:

rndColor()

updateGameList(gamesList,DOMelement)

threejs: