Záródolgozat

Kecskeméti SZC
Kandó Kálmán
Technikum

Skribbl.IO Clone

Tóth Csongor Gábor 13/a

Tartalom

Milyen területen írtam a záródolgozatomat?

Programozással már hamarabb megismerkedtem, már közel 5 éve. Sok programot írtam már, sajnos az esetek jelentős részében nem a saját akaratom és érdeklődésem miatt, hanem beadandó iskolai feladat miatt, de arról nincs kétség, hogy ne élveztem volna egyiket sem. Mindig volt 1 – 2 feladatrész, aminél hosszabb ideig kellett gondolkodnom milyen jó megoldások léteznek a problémára, és ez az, ami megfogott az egészben, a problémák megoldása és azok továbbfejlesztése. Ezzel szemben a webfejlesztést csupán alig 2 éve ismerem, és ezen belül is kevesebb mint 1 éve kezdtem el komolyabban foglalkozni vele. Elsőre a webfejlesztés nem érdekelt, mert az elején csak HTML és CSS volt, amivel, habár szép és érdekes dolgokat tudtam már akkor létrehozni, nem éreztem azt, amit a programozás tud nyújtani. Ezután ismerkedtem meg a weboldalak újabb, szinte legfontosabb elemét alkotó részével, a JavaScript-tel. Viszont sokáig ez sem keltette fel az érdeklődésemet. Már tudtam, hogy a weboldalról bevitt adatokkal tudok dolgozni, tudok a felhasználóval kommunikálni, de itt is csak a felszínt kapargattam, ami nem kötött le. De miután jobban el kezdtem beleásni magamat a témába, rájöttem mennyi potencia és lehetőség van egy webes projekt elkészítésében. Ezek után a webes téma választása eléggé meggyőzőnek és ígéretesnek tűnt, de ez még nem volt elég ahhoz, hogy meghozzam a végső döntésemet.

Web és Mi?

Ahogy már mondtam, egy webes dolgozat választása jó döntésnek tűnt, de önmagában nem tudta felülmúlni a játék iránti érdeklésemet. Biztos sokszor hallottuk már a ’És te mégis miért érdeklődsz annyira az informatika vagy esetleg a programozás felé?’ kérdésre azt a választ, hogy ’Sokat játszok játékokkal, így jó öltetnek tűnt.’ vagy hogy ’Szeretnék videójátékokat csinálni, és esetleg ebből jelentős pénzre szert tenni majd a jövőben.’. Nos az én gyerekkori nézeteim egy részét a játékkészítés fedte le amikor valamit ezt kérdést tette fel nekem. De szerintem ez majdnem minden leendő programozó, szoftverfejlesztő álma volt, hogy egyszer játék készítő/fejlesztő legyen, néhánynak biztosan sikerült is. Sokáig ez volt a fejembe, hogyha megszerzem ezt a szakmát, akkor én valamilyen játék projektben szeretnék részt venni. Ez a vágy azonban a jelenlegi projekt kiválasztásában nem segített. Mivel az egyik téma a régi, a másik pedig az új érdeklődési körömet fedte le, és emellett azt hitem a kettő terület nagyon messze fekszik egymástól, sokat vacilláltam mégis melyiket válasszam. Nem sok idő elmultával feltettem magamnak a kérdést, mégis miért kéne választanom a kettő közül csak az egyiket? Egy csomó játék található meg az interneten, s már akkor is volt ezekből amikor még el sem elkezdtem érdeklődést mutatni a számítástechnika és az informatika iránt. Manapság pedig már olyan játékokat tudnak az emberek létrehozni az interneten, amik többjátékosak, mind online és offline, emellett meglepően részletes grafikával rendelkeznek. Mért ne lehetne egy olyan weboldalt készíteni, ami önmagában egy játék? Így jutottam arra a döntésre, hogy nem választok, mindkettő ki fog venni valamennyi részt a záródolgozatból, s készítek egy olyan projektet, ami nem csak egy weboldal, de egy játék is egyben, így mindkettő területből tudok tovább fejlődni.

Milyen programokat, architektúrákat, keretrendszereket használtam?

Visual Studio Code

A Visual Studio Code egy világszerte ismert és használt program, amit a Microsoft fejlesztett és adott ki 2015-ben. A szoftver ingyenesen megszerezhető a weboldalukon, emellett több rendszer is támogatja, így akár Windowsra, macOS-re, vagy Linuxra is telepíthető. A program nagyon hamar megnyerte a szoftverfejlesztők tetszését, hisz a program meglehetősen sok személyre szabható elemmel rendelkezik. Alapvetően az alkalmazás támogatja a JS, HTML, CSS és PHP kiterjesztésű fájlok olvasását és formázását, de néhány kiegészítő letöltésével több 10 nyelvhez kaphatunk még támogatást, esetleg kódjavaslatokat is. Az ilyen bővítményekkel a program képes kiértékelni a kódunkat még mielőtt lefuttatnánk, a hibáinkat piros vagy egyéb színnel aláhúzva jelzi a hiba forrását, még képes javaslatot is tenni hogyan lehetne a problémát orvosolni. Az ilyen kiegészítők sokat tudnak segíteni, hogy a kód olvasható és átlátható legyen, ezen felül segíti a még tanulók fejlődését a kód megértésében és a hibák kijavításában is.

Emellett az alkalmazás alap funkciója a verziókezelés lehetősége, ami mind kisebb mind nagyobb alkalmazások esetében jól tud jönni. Miután összekötöttük az adott mappát egy verziókezelést támogató szolgáltatással, a program képes felismerni, ha ez megtörtént és bármely változás, amely az adott mappán belül történik felkínálásra kerül az alkalmazásban a felhasználó számára. Alapesetben a program csak a GIT verzió kezelőt ismeri, de ez itt is bővítményekkel kiegészíthető. Maga az alkalmazás kinézete is számos részben megváltoztatható. Lehetőség van a színek változtatására, az alkalmazáson belüli elemek áthelyezésére, még néhány ikon is megváltoztatható. Összességében az alkalmazás nagyon sok személyre szabási lehetőséget az a felhasználó számára, de változtatások nélkül is egy jól felépített, gyors, hatékony és szép alkalmazást kapunk, így nem csoda, hogy a program ilyen sok embernek nyerte el tetszését.

REST architektúra

Egy REST típusú architektúra kliensekből és szerverekből áll. A kliensek kéréseket indítanak a szerverek felé; a szerverek kéréseket dolgoznak fel és a megfelelő választ küldik vissza. A kérések és a válaszok erőforrás-reprezentációk szállítása köré épülnek. Az erőforrás lényegében bármilyen koherens és értelmesen címezhető koncepció lehet. Egy erőforrás-reprezentáció általában egy dokumentum, mely rögzíti az erőforrás jelenlegi vagy kívánt állapotát.

Bármely adott pillanatban egy kliens vagy állapotok közötti átmenetben van, vagy "nyugalmi" állapotban. A nyugalmi állapotban lévő kliens képes interakcióra a felhasználójával, de nem hoz létre terhelést és nem fogyaszt tárolót a szervereken vagy a hálózaton.

Ha a kliens készen áll az átmenetre egy új állapotba, akkor elkezdi küldeni a kéréseit a szerverekhez. Míg legalább egy olyan kérés van, amelyre nem érkezett válasz, a kliens átmeneti állapotban marad. Egyes erőforrás-reprezentációk hivatkozásokat tartalmaznak további erőforrásokra, amelyeket a kliens felhasználhat új állapotba történő átmenetkor.

A REST eredetileg a HTTP keretein belül lett leírva, de nem korlátozódik erre a protokollra. Egy "RESTful" architektúra más alkalmazási rétegbéli protokollra is épülhet, amennyiben az már rendelkezik értelmes erőforrás-reprezentáció átvitelhez szükséges gazdag és egységes szókinccsel. A "RESTful" alkalmazások maximálisan kihasználják a választott hálózati protokoll már létező, jól kialakított interfészeit és egyéb beépített képességeit, és minimalizálják új alkalmazás-specifikus jellemzők bevezetését.

Egy REST alkalmazás a HTTP protokoll meglévő tulajdonságait használja, és így lehetővé teszi a proxyknak és az átjáróknak, hogy együttműködjenek az alkalmazással (például gyorsítótárazás vagy biztonsági funkciók formájában).

Egy REST architektúrának a következő 4 megszorításnak kell többé, kevésbé megfelelnie, miközben az egyes komponensek implementációit nem korlátozza:

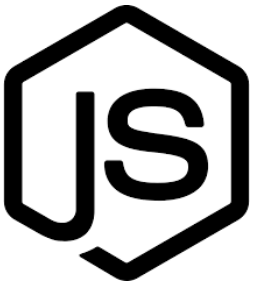
Kliens-szerver architektúra: A kliensek el vannak különítve a szerverektől egy egységes interfész által. Az érdekeltségek ilyen nemű szétválasztása azt jelenti, például, hogy a kliensek nem foglalkoznak adattárolással, ami a szerver belső ügye marad, és így a kliens kód hordozhatósága megnő.

Állapotmentesség: Az állapotmentesség egy olyan kommunikációs protokoll, amiben a kérést fogadó szerver nem tárol el adatot a kliensről. A kliens-szerver kommunikáció állapotmentes az által, hogy minden egyes kérés bármelyik klienstől tartalmazza az összes szükséges információt a kérés kiszolgálásához, és minden állapotot a kliens tárol.

Réteges felépítés: Egy kliens általában nem tudja megmondani, hogy közvetlen csatlakozott-e a végpont szerverhez, vagy közvetítő segítségével. A közvetítő szerverek megnövelhetik a rendszer skálázhatóságát terheléseloszlással és megosztott gyorsítótárak használatával.

Egységes interfész: Az egységes kliens-szerver interfész alapvető a „RESTful” rendszerek tervezéséhez. Egyszerűsíti és elválasztja az architektúrát. Ezáltal lehetővé teszi, hogy egymástól függetlenül fejlődjenek az egyes részek.

Node.JS



A Node.js egy többplatformos, nyílt forráskódú programozási környezet, amely Windows, Linux, Unix, macOS és egyéb rendszereken is futtatható. Az alkalmazás egy háttérben futó JavaScript futtatókörnyezet és keretrendszer, és az ide beérkező JavaScript-kódot webböngészőn kívül hajtja végre. A Node.js lehetővé teszi a fejlesztők számára a JavaScript használatát parancssori eszközök írásához és szerveroldali szkriptek készítéséhez. A JavaScript szerveren való futtatásának képességét gyakran használják dinamikus weboldaltartalom generálására, mielőtt az oldalt elküldené a felhasználó webböngészőjének. A program képes egyetlen programozási nyelv köré gyűjteni a webalkalmazások fejlesztését, szemben azzal, hogy a szerver- és kliensoldali programozáshoz különböző nyelveket használ.

A Node.js lehetővé teszi webszerverek és hálózati eszközök létrehozását JavaScript és a különféle alapvető funkciókat kezelő „modulok” használatával. Modulokat biztosítanak a fájlrendszer kezelésére, hálózatkezeléshez (ilyen a DNS vagy a HTTP/HTTPS), bináris adatokhoz, kriptográfiai funkciókhoz, adatfolyamok és egyéb alapvető funkciókat. A Node.js moduljai olyan API-t használnak, amely csökkenti a kiszolgálóalkalmazások írásának bonyolultságát, így akár kezdőként is hamar meg lehet érteni a különböző folyamatok működését.

A Node.js-t elsősorban hálózati programok, például webszerverek készítésére használják. A legjelentősebb különbség a Node.js és a PHP között az, hogy a PHP-ban a legtöbb függvény a befejezésig blokkol (a parancsok csak az előző parancs befejezése után fut le), míg a Node.js függvények nem blokkolnak (a parancsok párhuzamosan képesek futni, ezzel felgyorsítva a szerveroldali folyamatokat).

Ahogy már említettem, a Node.js rendelkezik úgynevezett modulokkal, melyek előre megírt függvények/osztályok. Hasonlóan, mint egy JavaScript könyvtárhoz, az adott modul letöltése után be tudjuk emelni a projektünkbe és ott szabadon használhatjuk. Ebbe a projektbe is használatra került néhány ilyen modul, melyek megkönnyítették a feladatom egy részét.

Node.js: Socket.io

A Socket.io egy a Node.js keretrendszerben használt modul, ami a webes WebSocketekre épül. Ezen modul beemelésével képesek vagyunk a szerver és a felhasználó közti kapcsolatot teremteni és fenntartani.

Maga a WebSocket egy ma már elterjedt technika, ami kétirányú, duplex kommunikációs csatornák kiépítését teszi lehetővé egyetlen TCP protokollon keresztül. Kifejlesztésének fő motivációja volt, hogy a webböngészőben futó alkalmazás képes legyen a szerverrel való kétirányú kommunikációra az úgynevezett Comet barkácsmegoldásai nélkül, bizonyos esetekben a szükségtelen fejlécforgalom akár 500:1-1000:1 arányú, a késleltetés 3:1 arányú csökkentésével. A WebSocket előtt nem volt lehetőség komplexebb, a szerverrel valós idejű kommunikációt igénylő webes alkalmazás, pl. csevegő alkalmazás, játékok, levelezőkliensek a HTTP-kapcsolatot nem rendeltetésszerűen használó alkalmazások megvalósítására.

A Socket.io nem sokban különbözik a websocket technológiától funkció és sebesség téren. Amiben eltér az az ezen modul által készített programok beemelése már projektekbe, hisz Socket.io-t használva, mi nem írunk közvetlen websocket kódot, mi csupán használjuk a már előre megírt függvényeket, amelyek különböző feltételeket biztosítanak, és egy adott verzió frissítés után automatikusan frissülnek. Ezzel szemben websocket esetén egy újabb verzió kiadása esetén nagyobb az esély a kód újraírására, míg Socket.io esetén ilyen miatt ritkán kell félnünk. Emellett a modul ígér egyéb biztonsági lehetőségeket, ilyen a visszaesési lehetőség, https kapcsolat romlás esetén automatikusan képes http kapcsolatra váltani, másrészt a szerver és a felhasználó közti kapcsolatot próbálja addig fenntartani amíg a kapcsolat meg nem szakad, vagy azt be nem fejezik, ami erőforráskímélő.

Node.js: Mongoose/MongoDB

A MongoDB egy mind online és offline használható adatbázis kezelő alkalmazás. Meg van benne minden alapvető funkció, amit egy adatbázis kezelő alkalmazásnak el kell látnia. Képes új adatot belehelyezni az adatbázisba, képes meglévő adaton módosítást végrehajtani és képes adatot törölni. Viszont ez az adatbázis nem a szokásoshoz hasonló adatokat képes tárolni, egy nem megszokott adatszerkezetben. Sok adatbázis kezelő alkalmazás saját nyelvet vagy adatszerkezetet hoz létre (SQL, XQuery, OQL), ezzel szemben a MongoDB JSON fájlformátumot használ, ami főképp webes területen terjedt el, de más programozási nyelvben is használható.

A MongoDB adatbázis szerkezete és szabályai is eltérnek az átlag adatbázisokéhoz képest, hisz közel sem annyira kötöttek, ezen kívül pedig egyedi adattípusokat is tárolhatunk. Kötetlenség alatt az adatbázisnak előre létrehozott adatszerkezetéhez való kötöttséget értem, persze itt sem lehet egy szám típusú mező helyére egy szöveget beírni. De például tömb esetében, ha nem határoztuk meg milyen típusú elemet helyezhetünk bele, akkor a program nem fog belekötni mit teszünk bele. Ebben viszont hátrány is rejlik, mert ilyenkor a program hibázási esélyt ad a felhasználó, de a program kezébe is. Egy nem pontosan létrehozott adatszerkezet esetében nem kívánt adat is belekerülhet a többi adat közé, ami nem helyes hibakezelés esetén össze is omlaszthatja a programot. Ezzel szemben egy SQL adatbázis esetén a szerkezet sokkal kötöttebb, kevesebb adattípussal rendelkezhetünk, de azok megbízhatóak. A legkisebb hiba esetén is jelez az adatbázis a hibáról, így azt időben észre tudja venni a fejlesztő és ki tudja javítani a hibát.

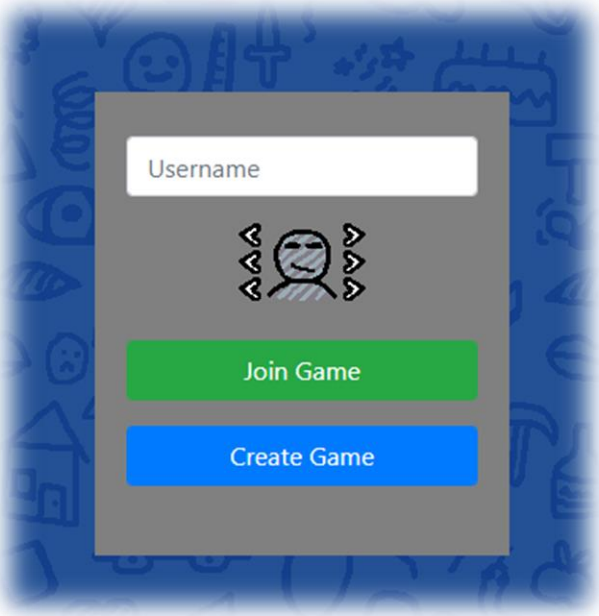
Amint már említettem, ezen fajta adatbázis szerkezete nem hasonlít a hagyományos adatbázis szerkezethez, ugyanis itt nincsenek táblák, helyette úgynevezett sémát tudunk készíteni, mely kulcs érték párokat tartalmaz. Ez adja az adatbázis kötetlenségét, hisz egy séma az nem egy pontosan követendő mintának minősül, hanem csupán egy várt eredményt megközelítő képként viselkedik. A végső eredményben lehet eltérés, de az általában nem tér el sokban a várt eredménytől. Nagy előnye még az adatbázisnak a skálázhatósága, hisz ezt horizontális skálázásra tervezték. Ez annyit takar, hogy jobb a meglévő alkatrészek jobbra való kicserélése helyett, új, akár ugyanolyan teljesítményű szervergépet helyezünk a rendszerbe, a bejövő kéréseket pedig megosszák egymás között. Egy erősebb processzor és sok ram képes annyiba kerülni, hogy abból tudjunk venni egy ugyanolyan számítógépet, mint amilyennel most rendelkezünk, akár többet is. Így összegészében ez a fajta adatbázis képes meglepően jó teljesítményt nyújtani, könnyen megérthető és skálázható, és ezen tulajdonságok igen fontosak egy adatbázis esetében.

A játék

Ahogy már az egyik előző szövegrészben olvasható volt, a záródolgozatokat webes környezetben szerettem volna megírni, emellett szerettem volna azt is, hogy egy játék legyen egybe. Akkor viszont még nem tudtam, hogy pontosan milyen játékot szeretnék készíteni. Azt

tudtam, hogy az elmúlt néhány évben egyre nagyobb szerepet kaptak az internetes játékok mind a fiatalabb, mind az idősebb korosztály részében. Nem is csoda, hisz most már képesek vagyunk olyan grafikával rendelkező játékokat készíteni, melyek néhány évvel ezelőtt elképzelhetetlenek lettek volna. De nem csak az ilyen játékok terjedtem el, hanem a .io játékok is. talán az egyik legkorábbi és egyben talán a legismertebb internetes játék az agar.io, de ezen kívül még egy csomó .io játék létezik, ilyen a már egy ideje ismert diep.io, hole.io vagy a slither.io. Egy ilyen játék viszont jobban felkeltette az érdeklődésemet, ez pedig a skribbl.io volt. A többi játékhoz képest, ahol egymás ellen harcolnak a felhasználók, itt az embereknek a szókincsüket és a rajzolási tudásokat kell használniuk és azt használva kell nyerni. Nagyon megtetszett ez a koncepció, és mivel megfeleltek a záró projektem feltételeinek, így arra jutottam, hogy a záródolgozatomat ennek a játéknak a mintájára csinálom.

A játék alapesetben böngésző és platform független, tehát különböző operáció rendszerrel rendelkező számítógépen, telefonon, tableten is játszható, viszont a tudásom hiányossága miatt a tableten és telefonon történő játékot nem tudtam biztosítani. Ezzel szemben asztali számítógépek esetében megpróbáltam a lehető legtöbbet megtenni, hogy a felhasználó ugyanúgy élvezze a játékot. Viszont térjünk is át arra, hogy mit látunk játék közben.



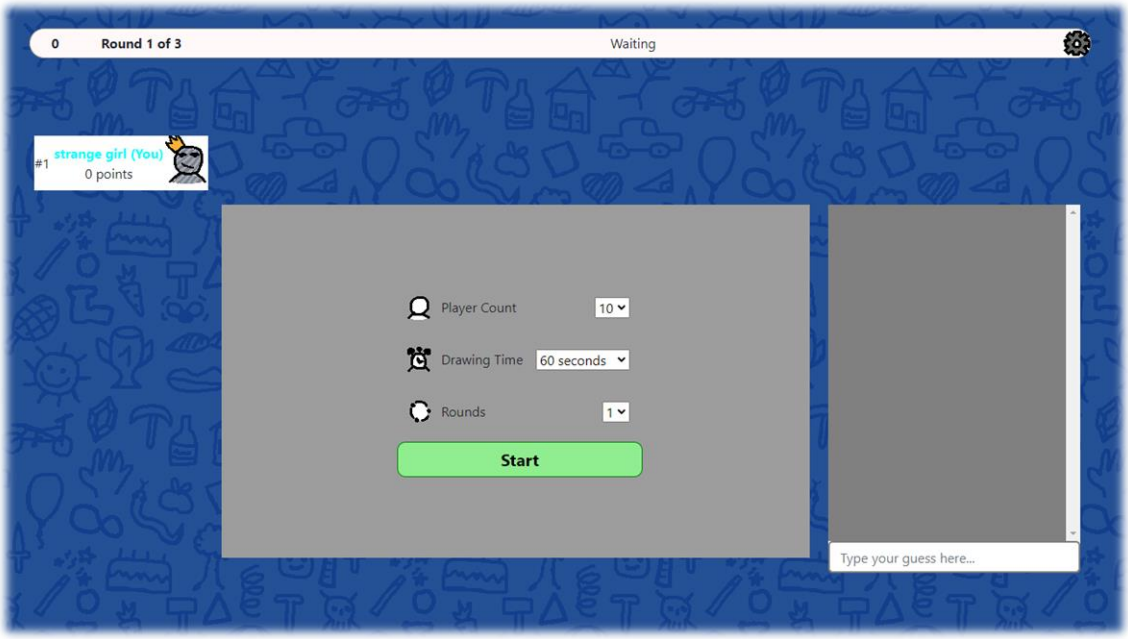
A weboldalra csatlakozáskor egy várószoba vár minket, fentről lefelé haladba a fontosabb elemeket említve, akkor elsőnek a felhasználónevünket adhatjuk meg, viszont, ha ezt üresen hagyjuk indításkor a program generál nekünk saját nevet.

Lehetőségünk van a mi saját kis karakterünk megváltoztatására is. A jobb és baloldali nyilak segítségével megváltoztathatjuk a karakter szemét, száját és testét is, amivel igen viccesen kinéző karaktereket hozhatunk létre. Találhatunk még két gombot melyekkel csatlakozni tudunk egy véletlen szerű szobához, vagy létre tudjuk hozni a saját szobánkat, ahol módosítani tudunk egy ilyen szoba alapbeállításain.

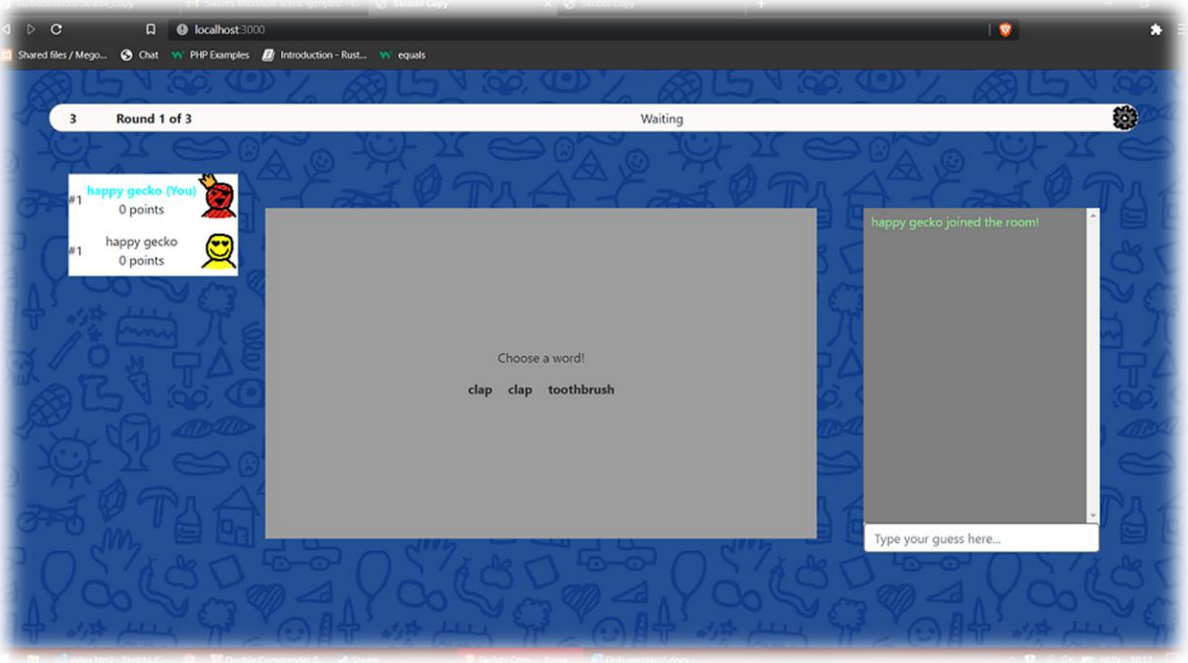
Ha a 'Join Game' gombra kattintunk, akkor amint említettem a szerver egy olyan szobát próbál találni számunkra, amiben van elég hely számunkra a biztonságos csatlakozáshoz. Ha

nem talál a szerver ilyen szobát, akkor generál számunkra egyet, ahol várnunk kell a többi játékos csatlakozására.

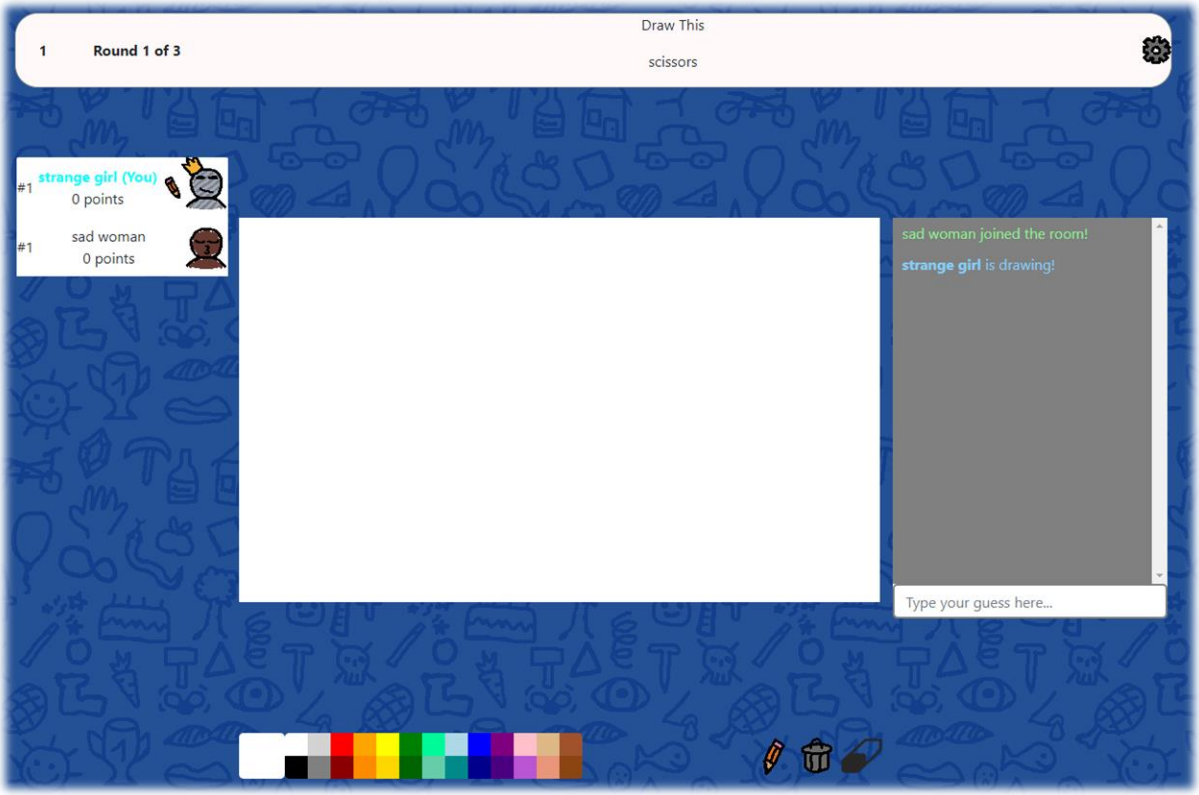
Ha pedig a ’Create Room’ gombra kattintunk, akkor a szerver biztosít számunkra egy új szobát, ahol be tudunk állítani néhány alapvető dolgot a szobával kapcsolatba. Ilyen a maximum csatlakozható játékos, amely minimum 2 és maximum 10 játékos lehet. A rajzolásra szánt idő, hisz itt minden körben minden játékosnak esélye van egy szó lerajzolására. Minél többen találják ki ezt a szót, annál több pontot kap a rajzoló, de ha senki sem találja ki, akkor neki sem jár a pont. Egy ilyen rajzolási idő minimum 15, és maximum 120 másodperc. Ezután a körök számát is megadhatjuk, itt legalább 1, és maximum 5 kört játszhatnak a játékosok. Ha minden beállítással elégedettek vagyunk, elindíthatjuk a játékot a ’Start’ gombbal.



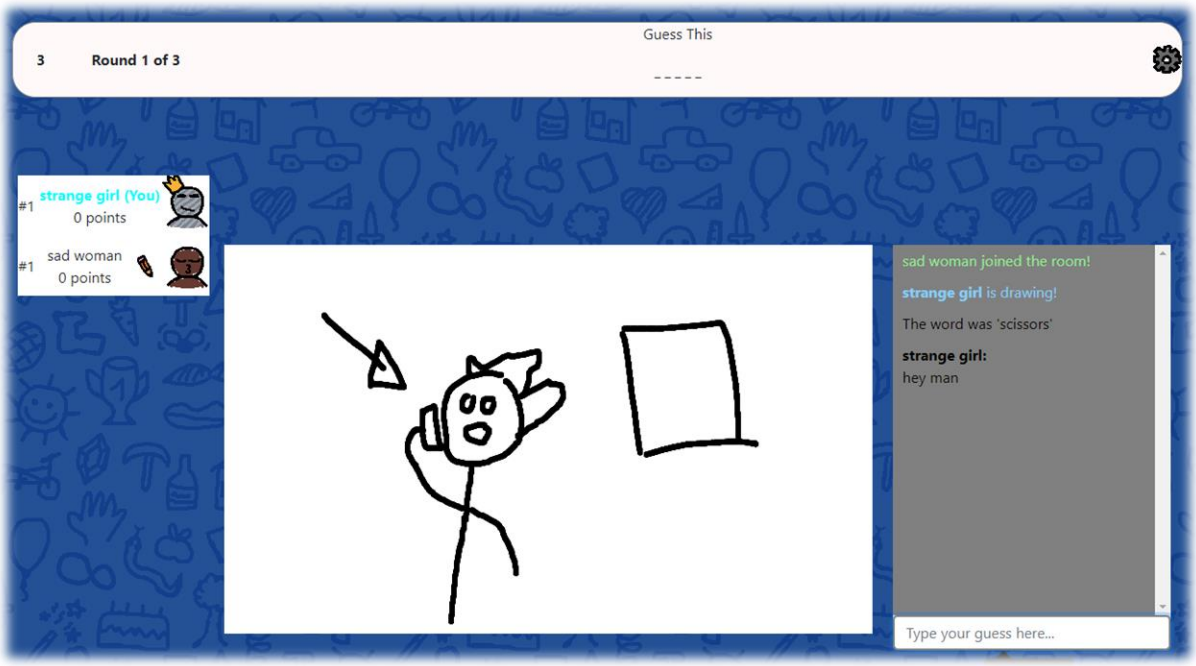
Ha elindítottuk a játékot, akkor a szerver eldönti, hogy a játékosnak még várakoznia kell, mert csak egyedül van a szobába, vagy a játék indulhat, mert a játékosok száma elérte a minimumot. A játék indításakor a játék tudatja velünk éppen melyik kör folyik, emellett az első rajzoló lehetőséget kap arra, hogy válasszon 3 szó közül melyiket szeretné lerajzolni. A játékos persze nem időzhet örökké, csupán 15 másodperce van kiválasztani a szót.



Ha a játékos kiválasztotta a szót, akkor megkezdődik a rajzolás, ellenkező esetben, az idő lejárta esetén a szerver véletlen választ a rajzoló számára. A rajzoló megkapja a palettát, amivel rajzolni tud, néhány alapszínt tud használni, kap egy ecsetet, egy radírt, és egy szemetest, amivel törölni tudja a teljes eddigi rajzát. Emellett persze nem kell a rajzolónak emlékeznie milyen szót választott, hisz ahol a többi játékos azt tudja meg, milyen hosszúságú az adott szó, ott a rajzolónak megjelenik a lerajzolandó szó.



Amíg a rajzoló a rajzolással van elfoglalva, addig a többi játékos tud figyelni, vagy tudja használni a csevegő csatornát, ahol tud más játékosokkal beszélni, vagy ha tudja mi a kitalálandó szó, ide tudja beírni. Ha helyes szót adott, meg a játék tudatja vele és a többi játékossal is hogy ő kitalálta azt, ezután viszont a menet végéig csak azok látják az ő üzenetét, akik szintén kitalálták a megfejtendő szót, ez alól kivételt tesz a rajzoló, aki mindenkinek az üzenetét látja. Ebből adódóan a rajzoló sem képes olyan üzenetet írni, amit lát olyan játékos is, aki még nem találta ki a szót.



Minden menetbe van rá esély, hogy a rajzoló nehéz szavakat kap, és nem tudja pontosan lerajzolni a kiválasztott szót. Ilyenkor a szerver segít a többi játékosnak. Az adott menet végéig a kitalálandó szó betűinek felét ismerteti a játékosokkal, melyeket folyamatosan pontos időközönként ismertet velük. Ha mindenki kitalálta a szót, vagy ha letelt az idő, akkor az adott menet véget ér, és megjelenik mindenki számára mi lett volna a helyes szó. Ezen kívül láthatjuk, hogy abban a menetben ki, mennyi pontot szerzett magának.

Ezt követően a menet folytatódik, és az előbb leírt folyamat ismétlődik az adott kör végéig. Ha elérjük a kör végét, akkor annyival egészül ki a menetek közti várakozás, hogy a játék ismét ismerteti melyik a jelenlegi kör.

Ha elérünk az utolsó kör végére, a játék még egyszer megmutatja abban a körben ki mennyi pontos szerzett, majd kiértékeli minden egyes játékos pontját és rangsorba állítja őket az alapján. A játék megjeleníti a már ranglistába sorolt felhasználókat, megjelenik a helyezésük, a nevük és a karakterük. Ezt követően a játék véget ér, de mindenki bent marad a szobába, és az adott szoba vezére újra elindíthatja a játékot, így nem kell mindig új szobát létre hozni minden alkalommal amikor játszani szeretnénk.

A játék felépítése

Szerver oldal

Az egész játék szívét és motorjának jelentős részét a mögöttes szerver teszi ki. A szerver felelős szinte majdnem minden fontos dologért, ilyenek maga a szobához való csatlakozás, vagy egy szoba létrehozása, a felhasználók kilépésének és belépésének a kezelése, az üzenetek továbbítása és még sorolhatnám.

user.js/room.js bemutatása (MongoDB sémák)

Ezen két JavaScript fájl szintén az előzőhöz hasonlóan csak a szerver oldalon található meg. Nem rendelkeznek nagy mérettel, de kulcsfontosságú szerepet töltenek be, hisz ezek adják az adatbázis sémáját. A MongoDB sémáinak egyik egyedi tulajdonsága, hogy a legtöbb adattípusnál lehetőség van további beállításokra. A legtöbb kulcsnak tudunk alapértéket állítani, szám esetében tudunk minimum és maximum értékeket megszabni. Szöveg esetében megtudjuk mondani az adatbázisnak, hogy a bevitt szó előtt és után lévő üres karaktereket automatikus távolítsa el. Fennáll a lehetőség arra is, hogy a fejlesztő maga írjon egy sajt beállítást. Ezzel tud a fejlesztő olyan ellenőrzéseket tenni, amit alapesetben nem lehetne, ilyen például, ha egy szöveg hosszát ellenőrizzük, vagy ha valamilyen indokból szeretnénk tudni, hogy egy szám páros e vagy sem. Viszont térjünk vissza a két JavaScript fájlra.

A kevésbé használtabb, de fontos fájl az a user.js, ugyanis ez adja minden felhasználó adatának az alapját. A séma csumán a legfontosabb adatokat tartalmazza, fontos volt, hogy felesleges esetleg nem szükséges információt ne tegyünk a szerkezetbe, hisz ebből az adatszerkezetből lesz a legtöbb az adatbázisba. A séma a következő fontosabb kulcsokat tartalmazza:

```
const mongoose = require("mongoose");

const userSchema = new mongoose.Schema({
    username : {
        type : String,
        trim : true,
        required : true,
    },
    socketid : {
        type : String,
        required : true,
    },
    isPartyLeader : {
        type : Boolean,
        default : false,
    },
    guessedIt : {
        type: Boolean,
        default: false,
    },
    guessedIndex : {
        type : Number,
        default : -1
    },
```

username: Ez tartalmazza a felhasználó nevét. Ez a kulcs tartalmaz néhány beállítást is, ezek a trim, és a required, melyek igaz/hamis változók. A trim a szöveg elején és végén levágja az üres szöveget, ilyen a space vagy egyéb speciális karakter. A required beállítás pedig kötelezővé teszi ennek a mezőnek a kitöltését.

socketid**:** Minden felhasználó csatlakozáskor kap egy azonosítót, ez a mező azt az azonosítót kell, hogy tartalmazza. A kulcs típusa szöveges, és a kulcs megadása kötelező.

isPartyLeade**r:** Egy szobának rendelkeznie kell egy vezérrel, aki képes elindítani az újabb játékot, miután az előző véget ért. Abban az esetben, ha a jelenlegi vezér kilép, ezt a kulcs fog megváltozni egy másik felhasználónál. A kulcs típusa logikai, tehát vagy igaz, vagy hamis értéket vehet fel. Itt találkozunk egy új beállítással, ami az alapérték adás, mely ebben az esetben hamis.

guessedIt: Ha egy ember kitalálja az adott körben lerajzolt szót, arra valahogy emlékeznünk kell, és ez a változó pont ezt teszi. A típusa logikai, és az alapértelmezett értéke hamis.

guessedIndex: Ez a változó arra szolgál, hogy tudjuk az adott menetben a felhasználók milyen sorrendbe találták ki a szót, hisz ez alapján tudjuk megmondani, ki mennyi pontot kap. Típusa logikai, alapértelmezett értéke pedig -1.

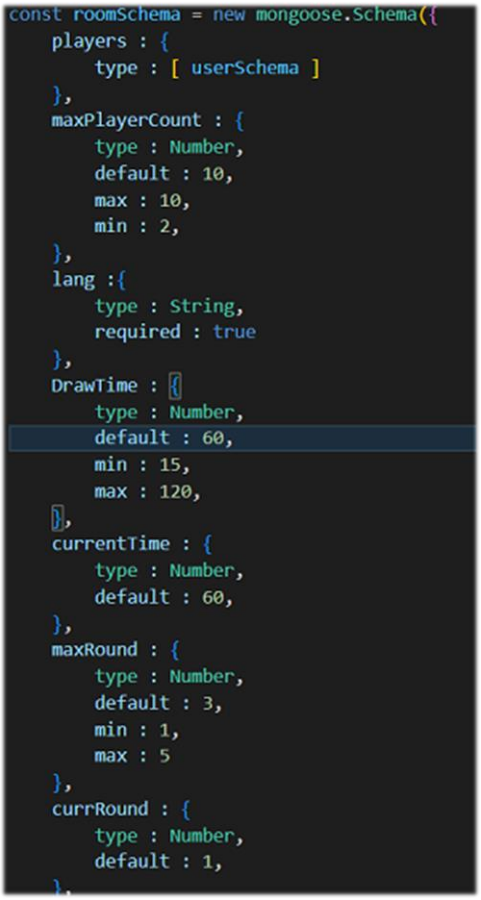
isDrawing: A változó azt mutatja, hogy az adott felhasználó rajzol ebben a körben, vagy sem. A kulcs típusa logikai, alapértéke pedig hamis.

points: Ez a kulcs csupán az adott játékban elért pontot menti el, a típusa szám érték, az alapértelmezett értéke pedig egyértelműen 0.

body_index, eye_index, mouth_index: Ez a három kulcs mind hasonló szerkezettel rendelkezik, csupán a feladatuk tér el egy keveset. Ezen kulcsok a mi saját létrehozott karakterünk adatait menti el, sorrendbe a karakter teste, a szeme, majd a szája. Mind a három típusa szám érték, és az alapértékük 0.

```
    isDrawing : {
        type : Boolean,
        default : false
    },
    points : {
        type : Number,
        default : 0,
    },
    body_index : {
        type : Number,
        default : 0
    },
    eye_index : {
        type : Number,
        default : 0
    },
    mouth_index : {
        type : Number,
        default : 0
    }
});
const usermodel = mongoose.model('User',userSchema);
module.exports = {usermodel,userSchema};
```

Az ennél kicsit fontosabb fájl, az a room.js, ugyanis ezt tartalmazza a szobákhoz szükséges sémát. A sémában kicsikét több adat és beállítás szerepel az előző sémához képest, de ez ne ijesszen meg senkit. A séma a következő kulcsokat tartalmazza:



```
const roomSchema = new mongoose.Schema({
    players : {
        type : [ userSchema ]
    },
    maxPlayerCount : {
        type : Number,
        default : 10,
        max : 10,
        min : 2,
    },
    lang :{
        type : String,
        required : true
    },
    DrawTime : {
        type : Number,
        default : 60,
        min : 15,
        max : 120,
    },
    currentTime : {
        type : Number,
        default : 60,
    },
    maxRound : {
        type : Number,
        default : 3,
        min : 1,
        max : 5
    },
    currRound : {
        type : Number,
        default : 1,
    },
```

players: Ez a kulcs felelős arról, hogy tudjuk ki van az adott szobában. Ez az egyetlen eset, ahol a kulcs típusa eltér az előre létrehozott típusoktól, ugyanis ez egy tömb, amiben az előző fájlban szerepelt séma az adattípus.

lang: Az eredeti játékban lehetőség volt arra, hogy külön szobák jöhessenek létre külön nyelveknek. A program fejlesztése alatt használva volt, viszont a végére ez a lehetőség nem került bele a programba. A típusa szöveges, és a megadása szükséges.

DrawTime: Minden szoba egyéni rajzolási időt tud megszabni a játékosainak, amit valamilyen módon el kell mentenünk, ennek a kulcsnak pont ez a feladata. A kulcs típusa szám érték, alapértelmezett értéke pedig 60. Itt találkozunk két új beállítással, ami a 'min' és 'max'. Jelentésük magért beszél, minimum és maximum értéket szabhatunk meg az adott kulcs számára, ebben az esetben minimum 15 és maximum 120,

currentTime: Ha egy szobában elkezdődik a menet, a számláló el kezd visszaszámolni, viszont, ha eközben egy újabb játékos csatlakozik, valahogy tudatnunk kell vele a jelenlegi időt. Amikor az idő változik, akkor ezt a kulcsot is megváltoztatjuk. A kulcs típusa szám érték, alapértelmezett értéke pedig 60.

maxRound: A szobák különböző maximális körszámmal rendelkezhetnek, melyeket minden szoba esetében el kell tárolni, és ezt a feladatot pont ez a változó látja el. Típusa szám érték, alapértelmezett értéke pedig 3. Szabályozva van a szám értéke is, minimum 1 és maximum 5 lehet.

currRound: Ahogy az idő esetében is, a jelenlegi kört is tudnunk kell, hogy hol tart a játék, és ezt a feladatot ez a kulcs látja el. Típusa szám érték, alapértéke pedig 1.

turnIndex: Ez a kulcs felelős azért, hogy tudjunk ki a következő rajzoló játékos. Ezt az értéket minden menet végén növekszik, a kör végén pedig visszaesik az alapértékére. Típusa szám érték, alapértelmezett értéke pedig 0.

word: Minden menet elindításánál a felhasználó választ egy szót, de mégis azt a szervernek honnan kéne tudnia? Ez a kulcs a jelenlegi választott szót menti el, természetesen ez minden menet kezdetekor változni fog. A kulcs típusa szöveges.

guessedCounter: Fontos lenne tudni azt is minden szoba esetében, hogy jelenleg már mennyi játékos találta ki a szót. Ezt a feladatot ez a változó látja el. Típusa szám érték, alapértelmezett értéke pedig 0.

```
    turnIndex : {
        type : Number,
        default : 0
    },
    word : {
        type : String,
    },
    guessedCounter : {
        type : Number,
        default : 0
    },
    gameState:{
        type : Number,
        default : 0
    },
    wordToChooseFrom : {
        type : [ Number ],
        default : [ 1 , 2 , 3 ]
    },
    prevRoundInfo : {
        type : [{}],
    },
    helpingLetter : {
        type : [Number],
        default : []
    }
});

module.exports = mongoose.model('Room',roomSchema);
```

gameState: Amikor egy menet elindul, akkor változik a játék állapota, amikor vége a menetnek akkor is, de ezeket alapesetben nehezen tudnánk megmondani. Ezen kulcs 3 olyan értéket tud felvenni, amihez a szerver különböző állapotokat tud kötni. A típusa szám érték, és alapértelmezett állapota 0. A 0 az a még el nem indított játékot jelenti, az 1 a szó választási folyamatát jelöli, a 2 pedig, hogy a menet éppen fut.

wordToChooseFrom : Minden ment elején felkínál a játék 3 választható szót az éppen soron következő rajzoló számára, de ezekre a szavakra szükségünk lehet, így ezeket a szavakat ebbe a változóba mentsük el. Viszont ahelyett, hogy a szavakat mentenénk el, helyette eltudjuk menteni azon szavak helyét a tömbön belül. Típusa szám érték típusú tömb.

prevRoundInfo: Minden kör végén az adott kör ki lesz értékelve, majd el lesz küldve minden felhasználó részére. Viszont, ha eközben új felhasználó érkezik, akkora a szoba már az új menetre váltott, és az előző kör kiértékelése elveszett. Ez a változó az előző kör kiértékelt információit menti el. Típusa egy objektum típusú tömb.

helpingLetter: Minden körben a játék segít azon felhasználóknak, akik még nem találták ki a szót. Viszont a játéknak emlékeznie kell, már melyik betűt adta meg a játékosoknak,

pontosabban, hogy milyen helyeken adott meg betűket a szóból. A kulcs típusa szám érték típusú tömb, alapértéke pedig egy üres tömb.

ValidLetters.js/word.js bemutatása

Az egész program tervezése alatt ebbe a két JavaScript fájlba lett a legkevesebb idő beletéve, de azért említésre méltóak.

ValidLetter.js: A fájl csupán egy tömböt tartalmaz, melyben az összes olyan karakter szerepel, mely nem számít speciális karakternek. A fájl a szerver oldalon csupán arra használatos, amikor a játék betűkkel segít a játékosoknak, ugyanis, ha szerepel a szóban szóköz, kötőjel, perjel, vagy egyéb speciális karakter, akkor azt ne válassza a játék.

word.js: A fájl szintén csak egy tömböt tartalmaz, ami fel van töltve különböző témájú szavakkal. Ezek közül a szerver minden menet elején kiválasztja azt a három szót, amelyek közül a rajzoló majd választ egyet.

index.js bemutatása

Az egész játék mögött egyetlen egy script felelős azért, hogy a játék tökéletesen fusson, ez pedig az index.js fájl. A fájl tartalmaz függvényeket, de főképp socket.io-s kéréseket és válaszokat láthatunk benne. Elsőként a függvények bemutatásával kezdek, így később már tudom rájuk csak névvel hivatkozni.

```
const JoinRoom = async function( username , socketid , lang , body , eye , mouth , cb) {
        try{
            //The order really matters in the query, first lt, then size and not reverse
            const room = await Room.findOne({lang : lang, players : {$lt : {$size : '$maxPlayerCount'}}});
            if(room){

                room.players.push({username : username,socketid : socketid, body_index : body, eye_index : eye, mouth_index : mouth});
                await room.save();
                return room;

            }
            else{
                const room = await createRoom( username , socketid , lang , body, eye, mouth, cb);
                return room;
            }
        }catch(err){
            console.error(err);
        }
}
```

JoinRoom: A függvény azért felelős, hogy egy új játékos csatlakozásakor a szerver találjon egy szobát a felhasználó számára. A függvény a következő adatokat várja el: a csatlakozó felhasználó neve, a socket modul által megadott azonosító, a kiválasztott nyelv, a felhasználó által létrehozott karakter teste, szeme és szája. Ezen kívül egy visszahívó függvény, amin keresztül megkapja a felhasználó a feladatok befejezte után a kívánt adatokat. A függvény a következő folyamatokat hajtja végre:

- Próbál keresni egy olyan szobát melynek nyelvezete egyezik az általa kiválasztott nyelvel, emellett, hogy a szoba jelenlegi mérete kevesebb legyen, mint a maximális.
- Találat esetén a felhasználót belehelyezi a szóbába, és továbbítja őt oda.
- Ha nincs találat, akkor készít neki egy szobát.

CreateRoom: A függvény feladata, hogy egy új szobát hozzon létre. Az elvárt adatok megegyeznek a 'JoinRoom' függvény által elvárt adatokkal. A függvény a következő folyamatokat végzi el:

- Létrehoz egy szobát melynek nyelvezete egyezik azzal, amit a felhasználó választott ki.
- Belehelyezi a felhasználót, majd a felhasználót továbbítja abba szobába.

TurnIsOver: A függvénynek egyetlen feladata van, megmondani, hogy az adott menet véget ért e vagy sem. Ami a függvénynek kell az az adott szoba, amiről megmondja, hogy vége vagy sem. A függvény megszámolja, hogy hány ember találta már el a szót helyesen, ebbe beleérti a rajzoló játékost is. Ha ez az összeg egyenlő az adott szoba létszámával, akkor a menet véget ért, ellenkező esetben pedig nem.

```
const CalculatePoints = async (room) => {
    //If noone guessed, than for now we don't do anything
    room.prevRoundInfo = [];
    await room.save();
    let points_data = [];
    if(room.guessedCounter == 0){
        for(let i = 0; i < room.players.length;i++){
            points_data.push({name : room.players[i].username , point_gain : 0});
        }
        return points_data;
    }

    const maxPoint = (room.guessedCounter + 1) * 80;
    for(let player of room.players){
        if(player.guessedIndex == 0){
            player.points += maxPoint - 80;
            points_data.push({name : player.username , point_gain : (maxPoint - 80)});
            continue;
        }
        if(player.guessedIndex != -1){
            console.log(player.guessedIndex);
            player.points += maxPoint - (player.guessedIndex - 1) * 80 ;
            points_data.push({name : player.username , point_gain : (maxPoint - (player.guessedIndex - 1) * 80 )});
            continue;
        }
    }
    await room.save();
    room.prevRoundInfo = points_data;
    await room.save();
    //console.log(room);

    return points_data;
}
```

CalculatePoints: Minden menet végén ez a függvény értékeli ki a jelenlegi kört, még mielőtt az a következőre váltana. A függvény csak azt a szobát várja el bemenetnek, amit ki szeretnénk értékelni. A függvény a következőket teszi:

- Elsőnek ellenőrzi, hogy volt e legalább egy ember ki kitalálta a szót, ha nincs akkor a függvény visszaküld egy objektumokkal teli tömböt, amiben senki sem szerzett pontot.
- Ellenkező esetben kiszámolja mennyi volt a maximálisan megszerezhető pont, és az alapján számítja ki, hogy ki mennyi pontos szerzett.

- Ezt egy objektumokkal teli tömb formájába küldi vissza a felhasználónak, emellett az adatbázisba is elmenti.

SetToStart: A függvény feladata, hogy alaphelyzetbe hozza az adatbázis változóinak egy részét. Csak a szobát várja el bemeneti változónak, és a következő elemeket változtatja meg:

- Minden játékos esetében a 'guessedIt' és 'isDrawing' kulcsot hamisra, a 'guessedIndex' kulcsot pedig -1-re állítsa.
- A szoba jelenlegi idejét a rajzolási időre állítja, a 'guessedCounter' változót pedig 0-ra.
- A választható szavaknak három új érréket ad meg, és a szerver által segített betűk tömbjét is lenullázza.

ResetGame: A függvény az előző függvényhez hasonlóan jár el. Az előző függvényhez hasonlóan elvégzi azokat a feladatokat, viszont egy – két változónak más értéket ad, emellett lét új elemnek változtatja meg az értékét.

- Az előző függvényben a szoba állapotát 1-re állította, ami a szó kiválasztását jelenti, itt viszont 0-ra állítja, ami a játék végét jelenti.
- Emellett két új változót állít 0-ra, ami a 'currRound' és a 'turnIndex'.

setGameState: a függvény csupán a szoba állapotát állítja be. Két változót vár el, az adott szoba azonosítóját, amit az adatbázis generált, hogy meg tudjuk találni a szobát az adatbázisba, és az állapotot, amire be szeretnénk állítani.

IncreaseGuessedUsers: A függvény növelni tudja, hogy hány ember találta már ki a szót az adott menetben, az adott szobában. Amit meg kell adnunk az szoba, és a felhasználó socket azonosítója. A függvény a következő feladatokat végzi el:

- Megkeresi a felhasználót az adatbázisból, ha valamiért nem találná, akkor a függvény nem folytatódik
- Ha megtalálja, akkor növeli a szoba 'guessedCounter' változóját egyel, és ezt a változót megadja a felhasználó 'guessedIndex' változójának.

SendToGuessedUsers: A függvény egy egyszerű feladatot lát el, egy felhasználó által megadott szöveget elküld mindenkinek egy adott szobán belül, aki vagy rajzol, vagy már kitalálta a szót. Három adatot vár a függvény, a szobád, a felhasználó nevét, aki írta, és az üzenetet, amit írt.

```
const EndGame = async (room) => {
    try{
    console.log('End of the game!');
    const point_gains = await CalculatePoints(room);
    await ResetGame(room);
    await setGameState(room._id,0);
    room = await Room.findOne({_id : room._id});
    io.to(room._id.valueOf()).emit('end-of-game',room);
    io.to(room._id.valueOf()).emit('round-over',room,'game-over',{point_gains : point_gains,rightWord : room.word});
    io.to(room._id.valueOf()).emit('new-message-to-user','',`The game is over!`,'Drawing');
    }catch(err){
        console.log(err);
    }
}
```

EndGame: Ha az összes kör véget ért, akkor a játék véget ér. Ez a függvény végzi el a szükséges lépéseket. Beviteli adatnak csak a szobát várja el, amit alaphelyzetbe tud tenni. A függvény a következőket teszi:

- Kiértékeli az előző menetet a 'CalculatePoints' függvénnyel, alaphelyzetbe állítja a 'SetToStart' függvénnyel, majd a 'setGaneState' függvénnyel beállítja a megfelelő állapotot.
- Ezután elküldi a kívánt adatokat a felhasználóknak, majd a játék újra elkezdhető lesz.

deleteRoom: A függvény csupán törli az adatbázisból azt a szobát amelyiket mi szeretnénk. A függvény egyetlen adatot vár el, a szoba azonosítóját, hogy megtudjuk keresni a szobát és ki tudjuk azt törölni.

```
const ChangeRound = async (room) => {
    try{
        if(room.currRound < room.maxRound){
            const point_gains = await CalculatePoints(room);
            await SetToStart(room);
            await setGameState(room._id,1);
            room.turnIndex = 0;
            room.currRound++;
            io.to(room._id.valueOf()).emit('round-over',room,'round-over',{point_gains : point_gains,rightWord : room.word,wordstochoosefrom : [words[room.
            wordToChooseFrom[0]],words[room.wordToChooseFrom[1]],words[room.wordToChooseFrom[2]]]});
            room.players[0].isDrawing = true;
            room.players[0].guessedIt = false;
            room.players[0].guessedIndex = 0;
            await room.save();
            return;
        }
            return await EndGame(room);
    }catch(err){
        console.log(err);
    }
}
```

ChangeRound: Minden menet végét ellenőrizni kell, hogy az adott kör véget ért e, ha pedig igen akkor az adatbázisba a jelenlegi kört növelni kell. Ezt a feladatot ez a függvény látja el. Bemeneti adatoként csak a szobát várja, amit ellenőrizni fog. A függvény a következőket teszi.

- Ellenőrzi, hogy a játék véget nem ért el, ha igen akkor meghívja az 'EndGame' függvényt,
- Ha még nem, akkor kiértékeli a jelenlegi menetet, és frissíti az adatbázist.
- A 'SetToStart' függvény segítségével alaphelyzetbe állítja a menetet, majd a 'SetGameState' függvénnyel beállítja a megfelelő játék állapotot.
- Növeli a jelenlegi kört tároló változót, majd jogot ad a soron következő rajzolónak a rajzolásra.

- Ezután pedig elküldi az adatokat a felhasználóknak.

```
const ChangeTurn = async (room) => {
    try{
        if(room.turnIndex + 1 < room.players.length){
            const point_gains = await CalculatePoints(room);
            await SetToStart(room);
            await setGameState(room._id,1);
            room.turnIndex++;
            await room.save();
            room = await Room.findOne({_id : room._id});
            io.to(room._id.valueOf()).emit('turn-over',room,'turn-over',{point_gains : point_gains,rightWord : room.word,wordstochoosefrom : [words[room.
            wordToChooseFrom[0]],words[room.wordToChooseFrom[1]],words[room.wordToChooseFrom[2]]]});
            room.players[room.turnIndex].isDrawing = true;
            room.players[room.turnIndex].guessedIt = false;
            room.players[room.turnIndex].guessedIndex = 0;
            await room.save();
            return;
        }
        //If the turnindex + 1 is not less than players.length, than it must be equal,
        //Since we only increment by one every time
            return await ChangeRound(room);
    }catch(err){
        console.log(err);
    }
}
```

ChangeTurn: Ha a jelenlegi menet véget ért, ezt a függvényt hívja meg a szerver, hogy a játék folytatódjon a következő menettel. Elvárt adatként azt a szobát kell megadnunk, ahol a következő menetre szeretnénk váltani. A függvény ezt így teszi meg:

- Ellenőrzi, hogy ebben a körben mindenki rajzolt e.
- Ha nem, akkor kiértékeli a jelenlegi kört, és elmenti az adatbázisba.
- Az adatbázist a menet előtti kezdeti helyzetbe teszi, és beállítja a megfelelő játék állapotot.
- Elküldi a kívánt adatokat a felhasználóknak, majd a soron következő rajzoló
- Ha igen, akkor a kör véget ért, és meghívja a 'ChangeRound' függvényt.

Ez az összes függvény, amit a szerver használ. Most pedig következzen a socket.io-s kérések és válaszoknak az áttekintése:

A socket.io úgymond 'hallgat', hogy mikor kap üzenetet, ezekre pedig tud reagálni más üzenetekkel, adatokkal, amiket a leírt függvényekkel tud kiszámolni. Hallgatni az 'on' függvénnyel, üzenete küldeni pedig az 'emit' függvénnyel képes. Ami viszont fontos mindkettő esetben az az, hogy milyen címkére hallgatunk vagy hogy milyen címkével üzenünk. Ezek a címkék csak szövegek, de ezekkel lehet megkülönböztetni az üzeneteket. Különböző címke esetén más eseményt hajt végre a szerver, és ezzel együtt más adatot ad vissza. A következő címkék a következő feladatokat végzik el:

'Join': Ha ilyen címkével érkezik üzenet, akkor valaki csatlakozni akar egy szobába. Az ilyenkor várt adatok az egy objektum benne fontosabb információval, és egy visszahívó függvény, amely segítségével visszajuttatjuk az adatot a felhasználónak. Ilyenkor a következő folyamatok mennek végbe:

- Ellenőrizzük, hogy a felhasználó átadta-e a felhasználónevét, a választott nyelvet és a socket azonosítóját, ha nem akkor hibát küldünk vissza a felhasználónak, és a függvény leáll.

- Ellenkező esetben meghívjuk a ’JoinRoom’ függvényt, hogy csatlakoztassuk a felhasználót egy szobához.
- Ezután egyrészt elküldjük a felhasználónak az adatokat az ’updateRoom’ címkével, és csatlakoztatjuk a felhasználót a megfelelő socket.io-s szobába is.
- Másfelől megnézzük, hogy a szoba éppen milyen állapotban van, és az alapján küldünk neki adatot ’turn-over’ címkével.
- Ezek után pedig, ha éppen rajzolás közbe léptünk be, akkor elkérjük a rajzolótól, hogy eddig mit rajzolt le.

’createRoom’: Ha ilyen címkével találkozunk, akkor a felhasználó egy új szobát szeretne létrehozni. Ilyen esetben a felhasználó adatait várja el a függvény, ezen kívül pedig egy visszahívó függvényt. Meghívjuk a ’createRoom’ függvényt a megfelelő bevitt adatokkal. Ha minden jól sikerült, akkor a visszahívó függvénybe elküldjük a várt adatokat, ellenkező esetben hibát küldünk vissza.

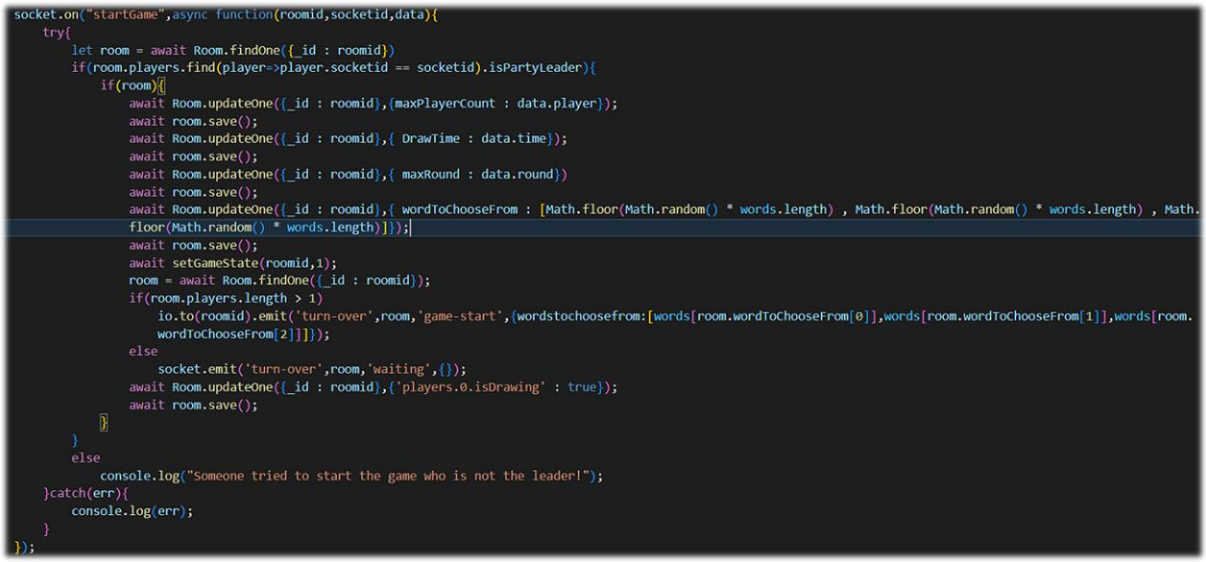
’disconnect’: Ilyen üzenet csak abban az esetben jön, amikor megszakad a kapcsolat a felhasználóval, vagy amikor egy játékos kilép. Itt nem várunk semmilyen adatot, és nem is csinál sok mindent, csupán kiírja a konzolra, hogy egy felhasználó kilépett.

’leave’: Nos ez a címke általában egyszerre hívódik meg az előbb leírt ’disconnect’ címkével. Amikor valaki kilép a játékból, akkor ezzel a címkével küld adatokat a szerver felé, a szerver dolga pedig a felhasználó kilépésének lekezelése. Ilyen esetbe a következő feladatokat hajtja végre a szerver:

- Megkeresi a szobát, amiből a felhasználó kilépett a kapott szoba azonosítóval, emellett a socket.io-s szobából is kivesszük a felhasználót.
- Megnézzük, hogy hány ember van a szobába, ha csak egy, akkor az egész szobát töröljük az adatbázisból, hisz csak mi voltunk benne.
- Ellenkező esetbe kivesszük azt a felhasználót a listából, aki kilépett, és ezt megosszuk a szoba többi felhasználóival.
- Emellett ellenőriznünk kell, hogy aki kilépett, az rajzoló volt-e, és ha igen, akkor ’ChangeTurn’ függvényhez hasonlóan megnézzük, hogy a kör véget ért e, és ha nem akkor elvégezzük a szükséges számításokat.
- Ha pedig véget ért, akkor meghívjuk a ’ChangeRound’ függvényt.

’new-message-to-server’: Ha valaki üzenetet küld, akkor ezzel a címkével kap üzenetet a szerver. Ilyenkor a szerver lekezeli a felhasználó üzenetét, hogy ez volt-e a kitalálandó szó, vagy hogyha már kitalálta akkor kinek mehet át az üzenet. A várt adatok a felhasználótól a szoba azonosítója, a felhasználó socket azonosítója, a neve, az üzenet, kitalálta-e már a szót, éppen rajzol-e és egy visszahívó függvény.

- A függvény elsőnek megkeresi a szobát, majd megnézi, hogy a szoba állapota 0 vagy 1-e. Ha igen, akkor automatikusan elküldi az üzenetet mindenkinek, majd a függvény leáll.
- Ha nem állt le a függvény akkor megnézi, hogy a felhasználó rajzol-e vagy kitalálta-e már szót, ha igen akkor elküldi az üzenetet a 'SendToGuessedUsers' függvény segítségével.
- Ha pedig még nem találta ki a szót, akkor megnézi a szerver, hogy a felhasználó kitalálta-e ezzel az üzenettel.
- Ha igen akkor növeljük a kitalált felhasználók számát az 'IncreaseGuessedUsers' függvénnyel.
- Ezután megnézzük, hogy a menet véget ért, e, ha igen, akkor meghívjuk a 'ChangeTurn' függvényt, emellett elmondjuk mi volt a helyes üzenet.
- Ha nem találta ki a szót pontosan, akkor megnézzük mennyire közelítette meg, és ha csak egy eltérés van, akkor ezt jelzi a szerver a felhasználónak.
- Ellenkező esetben a felhasználó üzenete továbbítva lesz mindenki számára.



```
socket.on("startGame",async function(roomid,socketid,data){
    try{
        let room = await Room.findOne({_id : roomid})
        if(room.players.find(player=>player.socketid == socketid).isPartyLeader){
            if(room){
                await Room.updateOne({_id : roomid},{maxPlayerCount : data.player});
                await room.save();
                await Room.updateOne({_id : roomid},{ DrawTime : data.time});
                await room.save();
                await Room.updateOne({_id : roomid},{ maxRound : data.round})
                await room.save();
                await Room.updateOne({_id : roomid},{ wordToChooseFrom : [Math.floor(Math.random() * words.length) , Math.floor(Math.random() * words.length) , Math.
                floor(Math.random() * words.length)]});
                await room.save();
                await setGameState(roomid,1);
                room = await Room.findOne({_id : roomid});
                if(room.players.length > 1)
                    io.to(roomid).emit('turn-over',room,'game-start',{wordstochoosefrom:[words[room.wordToChooseFrom[0]],words[room.wordToChooseFrom[1]],words[room.
                    wordToChooseFrom[2]]]});
                else
                    socket.emit('turn-over',room,'waiting',{});
                await Room.updateOne({_id : roomid},{'players.0.isDrawing' : true});
                await room.save();
            }
        }
        else
            console.log("Someone tried to start the game who is not the leader!");
    }catch(err){
        console.log(err);
    }
});
```

'StartGame': Amikor a szoba vezére elindítja a játékot, akkor ezzel a címkével együtt küldi a kezdéshez szükséges adatokat. Ilyen adat a szoba adatbázis azonosítója, a felhasználó socket azonosítója, ez biztonsági szempontból kell, majd adatok a szoba beállításairól, ilyen a maximális létszám, a rajzolási idő és a körök száma. Ilyen címke esetén a következő folyamat megy végbe:

- A szerver megkeresi a szobát, amit a vezér el akar indítani. Ha megtalálja, le ellenőrzi, hogy tényleg a vezér akarta-e elindítani a szervert és nem más. Ilyen esetben hibát dob vissza a szerver és a függvény leáll.
- Ha az azonosítás sikeres volt, a szerver beállítja a vezér által megadott szoba adatait, átállítja a szoba állapotát a megfelelőre, majd ellenőrzi, hány ember van a szobába.

- Ha csak még a vezér van benne, akkor a szerver várakozásra kéri meg a felhasználót, ha pedig már van elég, akkor a játék elindul, és a szerver kiválaszt 3 szót a rajzolónak.

'start-turn-to-server': Ha a rajzoló kiválasztotta a szót, amit le szeretne rajzoló, akkor ezzel a címkével érkezik az üzenet a szerver felé. A kért adatok a szoba azonosítója, és a választott szó. A szerver ilyenkor a következő folyamatokat végzi:

- A szerver megkeresi a szobát, ha nem találja, akkor a hibával tér vissza a függvény.
- Ha találat történt, beállítja a kiválasztott szót, frissíti a szoba állapotát, majd elküldi a megfelelő adatokat a felhasználóknak, és a játék elindul.

'paint-to-server': Amikor a játék megkezdődik, a rajzolónak rajzolnia kell valamit a vászonra, hogy a többi játékos kitalálja a szót. Ahhoz, hogy a többi felhasználó tudja milyen rajz készül, ahhoz a rajzolónak el kell küldenie a rajzolási adatot a szervernek, ami tovább küldi majd a felhasználónak. A függvény két adatot vár, egy eszközt, amit a rajzoló éppen használt, és az adatot, amit át tud adni a többi felhasználónak. Az adatot azt nem a szerver, hanem a felhasználó felüli JavaScript dolgozza fel és jeleníti meg a felhasználó részére.

```
socket.on('Change-Timer',async (roomid,time)=>{
    let room = await Room.findOne({_id : roomid});
    if(room){
        room.currentTime = time;
        await room.save();
        if(time != room.DrawTime && room.gameState == 2 &&time % (room.DrawTime / (room.word.length / 2 + 1)) == 0){
                console.log("Helping letter is being created!");
                let PossibleLetters = [];
                for(let i = 0 ; i < room.word.length ; i++){
                    if(!room.helpingLetter.includes(i) && ValidLetters.includes(room.word.charAt(i)))
                        PossibleLetters.push(i);
                }
                await Room.updateOne({_id : room._id},{$push:{helpingLetter : PossibleLetters[Math.floor(Math.random() * PossibleLetters.length)]}});
                await room.save();
                room = await Room.findOne({_id : room._id});
                socket.to(roomid).emit('Change-Timer',time,room.helpingLetter);
        }
        else
            socket.to(roomid).emit('Change-Timer',time);

        if(time == 0 && room.gameState == 2){
            io.to(roomid).emit('new-message-to-user','',`The word was <spam>'${room.word}'</spam>`,'Right');
            await ChangeTurn(room);
            console.log("Turn is Over!");
            return;
        }
        else if(time == 0 && room.gameState == 1){
            const index = room.wordToChooseFrom[Math.floor(Math.random() * room.wordToChooseFrom.length)];
            await Room.updateOne({_id : room._id},{word : words[index]});
            await room.save();
            room = await Room.findOne({_id : room._id});
            await setGameState(room._id,2);
            io.to(roomid).emit("start-turn-to-user",room);
        }
    }
});
```

'Change-Timer': Amikor a menet elindul, akkor egy számláló elindul a rajzolónál, és ez minden másodpercben csökken. A felhasználó minden ilyen változásnál egy üzenetet küld a szerver felé ezzel a címkével. A várt adatok a szoba azonosítója, és a felhasználónál lévő idő, amit továbbítani szeretnénk a felhasználóknak. A szerver ilyenkor a következő feladatokat végzi el:

- Az azonosító alapján megkeresi a szobát, majd miután megtalálja, beállítja az adott szoba jelenlegi idejét arra, amit a felhasználótól kapott.

- Ezután ellenőriz néhány dolgot a szerver a jelenlegi idővel és szoba állapottal kapcsolatba.
- Elsőnek ellenőrzi, hogy kell-e a szervernek segítenie a felhasználóknak egy betű megadásával, ha kell, akkor választ egy még nem választott betűt, majd elküldi az időt és a már megadott betűket a felhasználóknak.
- Ha pedig nem kell, akkor csak az időt továbbítja.
- Ezután megnézi, hogy az idő letelt-e, és hogy milyen a szoba állapota. Ha az idő letelt és a szoba állapota 2, akkor a menet véget ért. Üzen a szerver a felhasználóknak, és meghívja 'ChangeTurn' függvényt.
- Ha az idő 0. és a szoba állapota 0, akkor a rajzoló nem választott időbe betűt. Ilyenkor a szerver választ helyette, ezt elmenti az adatbázisba, megváltoztatja a szoba állapotát, üzen a felhasználóknak, majd elindítja a játékot.

'paint-data-to-server': Ha már egy menet közben csatlakozik be egy játékos, akkor a szerver küld egy üzenetet a rajzolónak 'paint-data-request' címkével, amelyben az eddigi rajzoltási adatokat kéri el a rajzolótól. Erre a kérésre 'paint-data-to-server' címkével válaszol a rajzoló. Ebben elküldi magát az adatot, emellett, hogy kinek kell elküldeni. A szerver ilyenkor csak tovább küldi az adatot a megfelelő felhasználónak 'paint-data-to-user' címkével.

Ezek lettek volna a szerver oldali fájlok és függvények. Első olvasásra biztosan nehezen lehet megemészteni, főleg, hogyha az ember nem jártas a témában. Igyekeztem minél köznyelvibb szavakat és kifejezéseket használni, hogy még a legbonyolultabb függvényeket is érthetővé tegyem.

Kliens oldal

A kliens oldal az, amit minden felhasználó máshogy tud tapasztalni, hisz azt nem feltétlenül a szerver irányít, hanem a felhasználó, és a mögötte lévő JavaScript függvények és metódusok. Amikor kliens oldalakról beszélünk, akkor el tudjuk különíteni két oldalt egymástól, az egyik oldal az, amit a felhasználó lát, és képes vele interakcióba lépni, ezt nevezzük **Frontend**nek, amelynek fejlesztése tud nehézségeket okozni.

A mások oldal az az oldal, ahol a különböző JavaScript fájlok futnak, és lekezelik a felhasználó cselekvéseit, esetleg egyéb folyamatokat futtatnak a háttérben, amit alapesetben a felhasználó nem lát, és nem tapasztal. Ez az oldal a **Backend**, amelynek fejlesztése szintén tud kellemetlen és nehéz lenni. Elsőnek a kliens oldali Backendet tárgyalom ki, majd utána a Frontendet.

Backend

A felhasználó oldali JavaScript fájlok felelősek azért, hogy a felhasználó kommunikálni tudjon egy távoli szerverrel, számításokat végezzen, esetleg, hogy segítséget nyújtson a felhasználónak, ha valami egyszerű problémába ütközik. Ezen játék esetében is szükség volt a szerver – felhasználó közti kapcsolat kiépítésére, melyekben a következő fájlok segítettek:

- Colors.js
- draw.js
- game.js
- generateName.js
- RenderPaletta.js
- mouse.js
- script.js
- socket.io.js
- és a ValidLetters.js.

A scriptek kőzött egy olyan script szerepel, amit alapesetben le kéne kérnie és töltenie a szervertől, de ha alapból le van töltve, akkor nem kell erre is várnia a felhasználónak. Ez a fájl a socket.io.js, ami azért felelős, hogy a szerver és a felhasználó közti WebSocketre épülő kapcsolatot fenntartsuk. Ahogy a szerver esetében is, itt is tudjuk használni a socket.io-s 'on' és 'emit' függvényeket, hogy üzenetekre hallgassunk és hogy üzeneteket küldjünk.

Ezen kívül pedig van egy olyan script, ami semmi célt nem szolgál, létre lett hozva, de nem lett végül vele kezdve semmi. Már fogalmam sincs mit akartam vele kezdeni.

A többi fájlnak viszont mind fontos, persze van néhány, ami esetleg csak 1 vagy 2 esetben használatos. Elsőnek ezekkel a scriptekkel kezdek, és haladok a fontosabbak felé.

Colors.js: A fájl csupán egy tömböt tartalmaz, amibe a palettánál használt színek vannak. Igazából ennyi, fel vannak benne sorolva azok a színek, vagy legalábbis megközelítő színek, amelyek az eredeti játékban is voltak.

ValidLetters.js: Ez a fájl név már ismerős lehet, ugyanis ezt a nevet már olvashattuk a szerver oldali JavaScript fájlok között. Nos ezt azért van, mert a két fájl megegyezik. A fájlban egy tömb létezik, amelyben az összes magyar betű kis és nagy mérete van.

Vectors.js**:** A fájlt egy előző projektemből emeltem be, ugyanis szükség volt arra, hogy koordinátákat összeadjak, kivonjak, szorozzak és osszak. A fájlban szerepel 2, 3 és négydimenziós vektor osztály, emellett a hozzájuk tartozó műveletek is. A műveletek között szerepelnek más függvények is, de nem szeretnék ezzel senkit sem untatni.

```
function generatename(){
    const firstnames = ['funny','sad','red','strange','happy','orange','bald','white','black','small','big','angry','pink'];
    const lastnames = ['gecko','elephant','cat','dog','man','woman','boy','girl','giraffe','buttterfly','goblin','panther'];

    const first = firstnames[Math.floor(Math.random() * firstnames.length)];
    const last = lastnames[Math.floor(Math.random() * lastnames.length)];

    return first+" "+last;
}
```

generateName.js: Ez az első fájl, ami már komoly jelentőséggel rendelkezik, ugyanis ez a fájl generál számunkra felhasználó nevet, ha a felhasználó nem adott meg egyet sem. A fájlba egy függvény szerepel, ezen belül van két tömb. A két tömb közül az egyikbe a név első része,

a másikba pedig a második része szerepel. Az elsőben inkább jelzők és tulajdonságok szerepelnek, míg a másodikba állati, tárgyi nevek szerepelnek. A függvény választ egy véletlen elő és utótagot, majd az összerakva küldi vissza a felhasználónak.

draw.js: A fájl a különböző eszközökhöz tartozó függvényeket tartalmazza, ezek a 'Draw',' Erase' és 'Fill' függvény, melyek közül az utóbbi nem került használatra a játékban sehol sem. Haladjuk sorrendbe a két használt függvény megvizsgálása során.

```
const Draw = (ctx,color,pos1,pos2,canvas_width) => {
    let RATIO = canvas.width / canvas_width;
    //Canculating the new positions of the points,
    //relative to the drawer's canvas size
    //(if wider, we pull the points apart,
    //if thinner, we pull them closer together)
    const new_pos1 = new vec2(pos1.x * RATIO,pos1.y);
    const new_pos2 = new vec2(pos2.x * RATIO,pos2.y);
    const DX = new_pos2.x - new_pos1.x;
    const DY = new_pos2.y - new_pos1.y;
    const DISTANCE = Math.sqrt((DX * DX + DY * DY));
    const SUB_STEP = DISTANCE;
    RATIO = 1.0 / SUB_STEP;
    ctx.beginPath();
    ctx.fillStyle = color;
    for(let i = 0;i < SUB_STEP;i++){
        ctx.moveTo((new_pos1.x + DX * RATIO * i),(new_pos1.y + DY * RATIO * i));
        ctx.arc((new_pos1.x + DX * RATIO * i),(new_pos1.y + DY * RATIO * i),3,0,Math.PI * 2.0);
        ctx.fill();
    }
    ctx.closePath();
}
```

Draw: A függvény azért felel, amikor a rajzoló éppen rajzolni szeretne a vászonra, az ott jelenjen meg ahol kell. Ezt a függvényt nem csupán az a felhasználó tudja meghívni, aki éppen rajzol, azok is meghívják, akik próbálják azt kitalálni. Ugyanis a szerver oldali részben említve lett egy 'paint-to-server' címkével jelölt függvény, ami tovább küldi a rajzoló által használt rajzolási adatot. Nos amikor megérkezik az adat, ezt a Draw függvényt hívja meg a script a kapott adatokkal. A függvény 5 adatot vár, a vászon úgynevezett rajzolási kontextus, ezzel tudunk rajzolni a vászonra. A szín, amivel rajzolni szeretnénk, az a két koordináta, ahonnan szeretnénk, hogy elkeződjön és befejeződjön a rajzolás. Emellett a rajzoló vászonának a szélessége. A függvény a következő folyamatokat végzi el:

- Kiszámol egy arányt, ami a saját és a rajzoló vászonának méretéből jön. Ez azért kell, mert, ha a vászon mérete eltér, akkor lehet technikailag olyan koordinátán rajzol a rajzoló, amit valaki nem is lát. Ha néhol ezzel az aránnyal beszorzunk, akkor ennek a veszélye eltűnik.
- Ezután a program be is szorozza a jelenlegi koordinátákat ezzel az aránnyal.
- Ezután kiszámolja a két koordináta közti x és y különbséget, ezeknek a segítségével pedig a távolságot számolja ki. A távolság értéket pedig egy 'SUB_STEP' nevű változóba helyezzük.

- Egy ciklus következik, ami 0-tól meg egész a 'SUB_STEP' változó értékéig. Ez azért kell, hogy a két koordináta közti helyen le tudjunk helyezni több pontot is, így amikor rajzolunk, nem lesz szaggatott.
- Meghatározzuk az adott cikluskörben hogy milyen koordinátán vagyunk, majd oda egy kört helyezünk azzal a színnel kitöltve, amilyet a függvénynek megadtak.
- A ciklus vége után lezárul a függvény.

Erase**:** Írhatnék a függvényről, legalább annyit, mint az előzőnél, de felesleges lenne. A két függvény szinte megegyezik, annyi különbséggel, hogy ez a függvény nem kér rajzolási színt. Egyébként ugyan az a folyamat meg végbe mind a két esetben.

game.js / script.js: Ez a két fő fájl, ami a játékmenet működéséért felel. Szép számban szerepel benne változó és mind sima, mind socket.io-s függvény is. Mivel sok alkalommal hívok meg egy függvényt, ami a másikban lett létrehozva, ezért az egészet egybe veszem. Elsőnek a változókkal, majd sima függvényekkel és utána a socket.io-s függvényeket mutatom be. Itt is próbálom a kisebb és kevésbé használtabb függvényeket elsőként elmagyarázni.

socket**:** Ebbe a változóba érjük el azokat a socket.io-s metódusokat, amiket sikeres kapcsolódás esetén kapunk. A kliens oldalon csupán két fő függvényt tudunk használni, a már ismert 'on' és 'emit' függvényt.

socketid**:** A változó a felhasználó socket azonosítóját tárolja. Ez néhány szerver oldali függvény esetében szükséges.

connected**:** A változóval csupán arra szolgál, hogy meg tudjuk határozni éppen csatlakozva vagyunk e bármilyen szobához.

username**:** Amikor a felhasználó belép egy szobába, ebbe a változóba tároljuk el milyen nevet adott meg vagy kapott.

roomid**:** A változóba annak a szobának az azonosítóját tárolja, amiben mi éppen vagyunk. Ez sokat segít abba, hogy megtaláljuk a megfelelő szobát különböző szerver oldali számítás esetén.

isLeader**:** A változó azt tárolja, hogy a felhasználó vezére e a szobának, vagy sem. Ez az információ a megfelelő elemek megjelenítésében segít.

current_room**:** Ebben a változóban arról a szobáról kapunk egy másolatot, amibe éppen csatlakozva vagyunk. Ennek a változtatása nem befolyásolja az adatbázist, de ettől függetlenül előnyre tehet szert egy már tapasztalt játékos.

isGuessed**:** Arról kapunk információt, hogy már kitaláltuk-e a jelenlegi szót.

isDrawing**:** Azt az adatot hordozza, hogy mi rajzolunk e a jelenlegi menetben vagy sem.

GameIsOn**:** Azt tudhatjuk meg vele, hogy a játék már elkezdődött e. Ez a változó a tényleges kezdéstől egészen a lezárásig igaz.

Timer: Amikor a felhasználó rajzol, ő diktálja az időt, vagy hát legalábbis a háttérben futó script. Ez a változó a jelenlegi időt tárolja, és ezt küldi el minden másodpercben a szervernek, hogy az továbbítsa minden felhasználónak.

MyTimer: Ez a változó tárolja azt a függvényt, amely másodpercenként számol vissza. Ha nem mentenénk ki a függvényt ebbe a változóba, akkor a függvényt a végtelenségbe számolna, így viszont le tudjuk állítani.

playerCount: A változó csupán a szoba létszámát menti el. Ez a 'GameIsOn' változó értékének a meghatározásához szükséges.

TOOL: Ez a változó azt annak az eszköznek a nevét tárolja, amit a rajzoló éppen használ.

Paint_Data: Ez a változó a rajzolók esetében használandó. Amikor egy rajzoló rajzol, akkor ebbe a tömbbe menti el minden egy rajzolási adatait azért, hogy ha menet közben csatlakozna egy új játékos, akkor nekik el tudja küldeni az eddig rajzoltakat.

```
let STATES = {
    MOUSEDOWN : false,
    MOUSEPREV: false,
    COLOR : 0,
    CURR : new vec2(0,0),
    PREV : new vec2(0,0)
}
```

STATES: Ez egy objektum, amely fontos értékeket tárol el. Ilyen, hogy az egér éppen le vagy e nyomva vagy sem, hogy az előző képkockánál le volt e nyomva az egér, kiválasztott szín indexe, a jelenlegi kurzor és az előző kurzor pozíció. Ezeket a változókat főképp csak a rajzoló használja, de mivel mindenkinek kell egyszer rajzolnia, így mindenki legalább egyszer használja őket.

window_width: Az ablak méretét menti el. Ez azért szükséges, mert így tudjuk megállapítani, hogy tényleg történt-e ablakméret változás.

body_index, eye_index, mouth_index: Belépéskor mindig egy véletlen karakter kapunk, amit tovább is változtathatunk. A véletlen karaktert ez a három változó szabályozza, ezen kívül ez változik, ha a felhasználó módosít a karakteren.

Ez lett volna az összes olyan változó, amelyek használatba vannak a játék során. Következzenek a függvények, szintén nagyjából fontossági sorrendbe.

setColor: Amikor a rajzoló kiválaszt egy színt, akkor azt a színt el kell mentenünk valamilyen módon. A függvény a 'STATES' objektum 'COLOR' változójába ment el a bekért értéket. Ez egy szám, ami a színnek az indexe a ***Colors.js*** fájlban lévő tömbben.

setCurrentTool: Ezzel tudjuk beállítani, hogy a felhasználó éppen milyen eszközt használ. A függvény az eszköz nevét kéri, amit elment a 'TOOL' változóba,

SwitchTools: A függvény megtudja jeleníteni és el tudja rejteni a rajzoló által használt palettát.

RenderPaletta: Ahhoz, hogy ne kelljen minden színt egyesével megírni a palettánál, helyette programozottan generáljuk. Ez a függvény pont ezt csinálja. Le generálja a HTML

elemeket, majd belehelyezi azokat a megfelelő elemhez. Minden elemnek odaadja a 'setColor' függvényt a megfelelő értékkel, hogy kattintásra átállítsa a színt

gameon: Miután a felhasználó sikeresen belépett egy szobába, ez a függvény állítja át láthatóvá a weboldal megfelelő elemeit. Ezen felül a 'Joined' változót igazra állítja.

gameoff: A függvény az előző ellentétét csinálja, azokat az elemeket amiker a 'gameon' megjelenít, azokat eltünteti, és a kezdő oldalt jeleníti meg.

ClearCanvas: Amikor a rajzoló törölni akarja a vászon tartalmát, akkor a szemetesre kattintva ezt a függvényt hívja meg. A vászon rajzolási kontextusának segítségével az egész vászon tartalmát fehérre állítja.

StartTurn: Ha a rajzoló kiválasztotta magának a szót, akkor a szóra kattintva ezt a függvényt hívja meg. A függvény csupán elküldi a kívánt adatokat a szervernek 'start-turn-to-server' címkével.

```
const RollUp = () => {
    room_maker.classList.remove('roll-down');
    room_maker.classList.add('roll-up');
}
```

RollUp: Amikor egy menet vagy kör véget ér, akkor a lebegő ablak, amibe az adatok megjelennek lefelé és felfelé lebeg. Ez a függvény a felfelé lebegést állítja be. A függvény ellek a lebegő ablaknak ad egy osztályjelölőt, amivel képes 'leúszni'.

RollDown: A függvény pont a ***RollUp*** ellentéte. Ahelyett, hogy a lebegő ablak felfelé menne, helyette lefelé megy, amit szintén az osztálykijelölők kicserlésével hajtja végbe.

validateName: Ezzel tudjuk ellenőrizni, hogy a felhasználó megfelelő nevet adott e meg. Egy adatot vár el, az ellenőrizni kívánt nevet. Ellenőrzi a függvény, hogy ne csak szóközből álljon és hogy a hossza 1 és 30 közé essen.

renderTimer: A függvény csupán a jelenlegi időt jeleníti meg az arra alkalmas szövegdobozba. Bemenetként pedig azt az értéket kapja, amit be szeretne állítani.

Disconnect: A függvény a felhasználó lecsatlakozásakor hívódik meg. Ha már csatlakozva volt egy szobához, akkor küld egy üzenetet a szervernek 'leave' címkével és elküldi a szükséges információt, emellett meghívja a 'gameoff' függvényt, hogy a megfelelő elemek látszódjanak.

ChangeTimer: A függvény akkor kerül meghívásra, amikor egy menet elindul, ezen belül is csak a rajzolónál. Ez diktálja az idő telését, így minden másodpercben meghívásra kerül. Amíg az idő nem éri el a nullát, addig minden másodpercben csökkenti, elküldi ezt a szervernek

```
function StartGame(){
    let player = document.querySelector("#player-count").value;
    let time = document.querySelector("#drawing-time").value;
    let round = document.querySelector("#round-count").value;
    socket.emit("startGame",roomid,socketid,{player,time,round});
}
```

’Change-Timer’ címkével, majd ezután megjeleníti a frissített időt a ’renderTimer’ függvénnyel.

StartGame**:** A függvény akkor hívódik meg, amikor a szoba vezére elindítja a játékot. Ilyenkor a függvény kiolvassa az általa meghatározott szobabeállításokat, ezután ezeket és még néhány szükséges adatot elküld a szervernek ’startGame’ címkével.

EndOfGame**:** A függvény csupán egyszer játszódik le, akkor is a játék legeslegvégén. Ilyenkor leállítja a még futásba lévő számlálót, feltéve, hogy fut. Három változót állít hamisra, a ’GameIsOn’, az ’isGuessed’ és a ’isDrawing’ változót. Ezután meghívja a ’SwitchTools’ függvényt.

```
const ShowPointGains = (rightword,point_gains) => {
    room_maker.innerHTML = `
    <p id='rightword'>The right word was <span>${rightword}</span></p>
    <p>Time is up!</p>
    <table>
        ${point_gains.map(info=>`
        <tr>
            <td>${info.name}</td>
            <td class='${info.point_gain > 0 ? 'Gained' : 'NotGained'}'>${info.point_gain > 0 ? '+' : ''}${info.point_gain}</td>
        </tr>
        `).join('')}
    </table>
    `;
}
```

ShowPointsGains**:** Amikor egy menet véget ér, akkor a felhasználók részére meg kell jeleníteni, mi volt a helyes szó, és hogy ki, mennyi pontot szerzett abban a körben, amit ez a függvény végez el. Bemenetként a helyes szót, és a felhasználónkénti pont növekedéseket kapja meg. Ezeket már csak behelyettesíti a megfelelő helyre.

ShowCurrentRound**:** Ha egy új körbe lép a játék, akkor a lebegő ablaknak azt is meg kell jelenítenie, hogy most melyik körbe léptünk. A függvény a jelenlegi kört és a maximális kört kapja meg bemeneti adatnak, amit beleír a lebegő ablakba.

ShowFinalResult**:** A játék végén megjelenik, hogy ki nyerte meg a játékot, és azt is hogy mi milyen helyezést ért el. Ez a függvény ezt a feladatot látja el. Bemenetként az összes felhasználót várja, majd a következőket végi el rajtuk:

- Növekvő sorrendbe helyez a pontjuk alapján.
- Ezután végig megy minden felhasználón, és olyan sorrendbe adja meg a helyezésüket.
- Mivel egyetlen sorba nem férne ki sok ember, ezért direkt úgy generálja a HTML elemeket, hogy az első sorban a három dobogós és az őket követő sorokba maximum 4 felhasználó legyen.

ShowWordChoosing**:** Minden menet kezdete előtt a rajzolónak választania három szó közül egyet, amelyiket le akarja rajzolni. Ez a függvény ezeket a lehetőségeket jeleníti meg a rajzoló számára. Bemenetként a három lehetőséget várja el.

- Elsőnek azt kell megtudni, hogy a felhasználó rajzoló e, ha igen, akkor felkínáljuk neki a három szót.
- Végig megyünk a három szón, és mindegyikhez hozzá kötjük a ’StartTurn’ függvényt a megfelelő adatokkal, majd elindítjuk a visszaszámlálót, ami másodpercként hívja meg a ’ChangeTimer’ függvényt.
- Ha viszont a felhasználó nem is rajzoló, akkor csak tudatjuk vele, hogy éppen kicsoda is választ, emellett megjelenítjük a karakterét a ’CreateAvatarText’ függvény segítségével.

ShowRoomMaking**:** Amikor még a játék kezdése előtt vagyunk, akkor a szoba vezérének lehetősége van arra, hogy néhány beállítást megváltoztasson a szobával kapcsolatba. Ezeket a beállításokat ez a függvény jeleníti meg. Lehetőséget ad a maximális létszám, a köröm száma, és a rajzolási idő beállítására. Alul pedig egy zöld gomb ad lehetőséget a játék elindítására, ami a ’StartGame’ függvényt hívja meg.

ShowRollDown**:** Sok eseménykor a lebegő ablaknak meg kell jeleníteni valamilyen adatot, de mégis honnan kéne tudnia, mit jelenítsen meg? Ez a függvény pont ezt a feladatot lássa el, minden eseménynél kap egy változót, ami alapján képes a helyes folyamatot végrehajtani. Bemenetnek kér a függvény egy esemény típust, a kör végi pontszerzési információt, a helyes szót és a lehetőségeket, amik közül a rajzoló választani tud. Összesen 6 fajta esemény típust ismer a függvény, amiket így kezel le:

- ’room-making’ esetén meghívja a ’ShowRoommaking’ függvényt, és eközben az ablak lehúzódik.
- ’waiting’ esetén csak egy szöveget jelenít meg a lebegő ablakba, ami várakozásra kéri a játékost.
- ’game-start’ esetén megjeleníti a jelenlegi kört a ’ShowCurrentRound’ függvénnyel, majd felkínálja a rajzolónak a választási lehetőséget a ’ShowWordChoosing’ függvény segítségével.
- ’turn-over’ esetén megjeleníti a szerzett pontokat a ’ShiwPointGains’ függvénnyel, és felkínálja a választási lehetőséget.
- ’round-over’ esetén ez megegyezik az előzővel, de a két esemény között a jelenlegi jelenlegi kört is.
- ’game-over’ esetén a megjeleníti a szerzett pontokat, majd megjeleníti a végső helyezéseket a ’ShowFinalResult’ függvény segítségével. Ezután pedig újra megjelenik a szoba beállításaival kapcsolatos ablak.

```
const CreateAvatarText = function(body,eye,mouth,Leader){
    return `
    <div class='avatar'>
        <div class='body' style='background-position : -${(body % 10) * 100}% -${Math.floor(body / 10) * 100}%'></div>
        <div class='eye' style='background-position : -${(eye % 10) * 100}% -${Math.floor(eye / 10)  * 100}%'></div>
        <div class='mouth' style='background-position : -${(mouth % 10) * 100}% -${Math.floor(mouth / 10)  * 100}%'></div>
        ${Leader ? `<div class='leader'></div>` : ""}
    </div>
    `;
}
```

CreateAvatarText**:** Minden felhasználó rendelkezik egy saját karakterrel/avatárral, amit mindig a neve mellett jelenítünk meg. Ez a függvény egy szöveget hoz létre, amiben pontosan le van generálva a karakterhez szükséges HTML kód. Bemenetként a karakter test, szem és száj indexét, emellett, hogy szoba vezér e a felhasználó. Az indexek segítségével egy hatalmas úgynevezett 'kép atlaszból' kiválasztja a megfelelő testet, szemet és szájat. Szoba vezér esetén pedig egy koronát is kap.

ChangeAvatar**:** Még a szoba csatlakozása előtt lehetőség van arra, hogy a felhasználó megváltoztassa az avatárját, amit ez a függvény végez el. Bemenetként egy számot kap, amiből tudja, hogy a 6 nyíl közül éppen melyiket nyomtuk meg, ebből pedig tudjuk kötni, hogy mit változtatott meg. A változásokat a már említett 'body_index', 'eye_index' és 'mouth_index' változóba menti el.

```
function sendMessage(event,obj){
    if(event.key == 'Enter'){
        event.preventDefault();
        const text = obj.value;
        if(text.trim().length > 0){
            socket.emit("new-message-to-server",roomid,socketid,username,text,isGuessed,isDrawing,function(err,res){
                if(err){
                    console.log(err);
                    return;
                }
                isGuessed = res;
                console.log(isGuessed);
            });
            obj.value = '';
        }
    }
}
```

sendMessage: Amikor valaki üzenetet küld, akkor elsőnek ezt a függvényt hívja, meg és ez továbbítja az üzenetet a szerver felé. A függvény minden billentyű lenyomására meghívódik, de csak Enter lenyomás esetén ellenőriz. Megnézi, hogy a szöveg hossza nagyobb e mint 0, és ha igen, csak akkor továbbítja a szervernek 'new-message-to-server' címkével. Elküldi a megfelelő adatokat a szerver számára, majd a visszahívó függvénybe kapott adatot értékeli ki. Hiba esetén kiírjuk a hibát a konzolra, ellenkező esetben pedig arról kapunk információt, hogy kitaláltuk e a szót, amit az 'isGuessed' nevű változóba mentünk.

renderPlayers**:** Amikor belépünk egy szobába, akkor láthatjuk a többi felhasználó adatait helyezését és karakterét a bal oldalsó oszlopban. Ezt a feladatot látja el ez a függvény. Bemenetként a jelenlegi szobát várja el, de abból csak a felhasználókat tároló tömbbel fog dolgozni. A függvény a következőket végzi:

- Elsőnek minden felhasználó pontját kimenti egy tömbbe, kiszűri a duplikált értékeket, amit majd szortíroz. Ez a lépés a felhasználók helyezésének meghatározására lesz hasznos.
- Ezután végigmegy az összes felhasználón egy ciklussal, és minden felhasználónak generál egy HTML elemet, amiben megjelenik a helyezése, felhasználó neve, elért pontja és karaktere.
- Ezután ezeket az elemeket mind beleteszi a bal oldalsó oszlopba.

JoinRoom**:** A függvény akkor kerül meghívásra, amikor a felhasználó éppen be szeretne lépni egy szobába. Elsőnek lekéri a függvény, milyen nevet adott meg a felhasználó. Ha megadott, ellenőrizzük, és ha helyes akkor elmentjük a 'username' változóba. Ha nem, ezt jelezzük a felhasználónak. Ha nem adott meg nevet, akkor generálunk neki egyet a 'generateName' függvény segítségével.

Ezután egy üzenetet küldünk a szervernek 'Join' címkével, melyben elküldjük a szükséges adatokat. Itt is alkalmazzuk a visszahívó függvényt, aminél hiba esetén jelzünk a felhasználónak. Ellenkező esetben beállítunk néhány alap változó értéket.

Ilyen a 'roomid', 'isLeader', 'isDrawing', 'playerCount', 'GameIsOn', 'current_room', 'Joined' és 'Timer' változót.

Ezen kívül meghív még néhány függvényt is, ezek a 'Rolldown', 'SwitchTools', 'ClearCanvas', 'renderPlayers' és a 'gameon' függvényt.

CreateRoom**:** Az előszobában lehetősége van a felhasználónak arra, hogy saját szobát készítsen, amit ez a függvény tud megtenni. Elküld egy üzenetet a szervernek 'createRoom' címkével és a szükséges adatokkal, a visszahívó függvénybe pedig a választ várja. Hiba esetén ezt kiírja konzolra, ellenkező esetben pedig ugyanazokat a lépéseket teszi, mint amit a 'JoinRoom' esetében végez

Ezek lettek volna a sima függvények, most pedig következzen a socket.io segítségével használtak.

'Change-Timer'**:** Mivel a rajzoló küldi a szervernek az időt, a szerver tovább küldi a felhasználóknak ezzel a címkével. Ilyenkor érkezik az idő, és betűk, amikkel segít a szerver a játékosoknak. Ha a felhasználó nem rajzol, és érkezett segítség, akkor az eddig kitalált betűket frissíteni kell.

'end-of-game'**:** A felhasználó a játék végével kapja ezzel a címkével ellátott üzenetet. Ilyenkor megkapja a szobát, megjeleníti a felhasználókat a 'renderPlayers' függvénnyel, és meghívja az 'EndOfGame' függvényt.

'connect'**:** Ha sikeresen csatlakozott a felhasználó a szerverhez, ezzel a címkével kap üzenete. Ilyenkor a script elmenti a socket azonosítót a 'socketid' változóba, és a 'connected' változót is igazra állítja.

'updateRoom'**:** Ha valamilyen frissítés történi az adott szobába, akkor a szerve ezzel a címkével küld üzenetet a felhasználóknak. Ilyenkor megkapják az új szobát, s néhány változó értékét újra ellenőrzik és beállítják. Ilyen a 'IsLeader', 'isDrawing', 'current_room', 'playerCount' és a 'GameIsOn'. Ezen kívül frissíti a bal oldali felhasználók listáját a 'renderPlayers' függvénnyel.

’start-turn-to-user’**:** Amikor a menet elindul, a szerver küld minden felhasználónak egy üzenetet ezzel a címkével, amiben a legfrissebb szoba adatok vannak. Ilyenkor a felhasználó sok adatot újra ellenőriz, emellett függvényeket is meghív. Le ellenőrzi a ’current_room’, ’isLeader’, ’isDrawing’,’Timer’,’isGuessed’,’playerCount’ és a ’GameIsOn’ változót. Leállítja az éppen futó időzítőket, hogy azok a megfelelő idővel induljanak újra. Meghívja még a ’Switchtools’, ’RenderPlayers’ és ’ClearCanvas’ függvényeket is. Ezen kívül a felhasználó jogai alapján beállít néhány szöveget és lehetőséget.

```
socket.on('turn-over',function(room,type,datas){
    console.log('Turn Changed!');
    current_room = room;
    ShowRollDown(type,datas.point_gains,datas.rightWord,datas.wordstochoosefrom);
    console.log(room);
    isLeader = room.players.find(user=> user.socketid == socketid).isPartyLeader;
    isDrawing = room.players.find(user=> user.socketid == socketid).isDrawing;
    SwitchTools();

    renderPlayers(room);
    // ClearCanvas();

    clearInterval(MyTimer);
    MyTimer = null;

    Timer = room.DrawTime;
    isGuessed = false;
    playerCount = room.players.length;
    GameIsOn = playerCount > 1 ? true : false;
});
```

’turn-over’ / ’round-over’**:** Ha egy menet vagy kör véget ér, azt a szerver ezzel a kettő címkével szokta jelezni. Ilyenkor elküldi a szoba adatait, hogy milyen típusú üzenetet kell a lebegő ablaknak megjelenítenie, és az előző körben szerzett pontokat.

Mindkettő esetben a következő változókat ellenőrzi a két függvény: ’curent_room’, ’isLeader’, ’isDrawing’,’Timer’,’MyTimer’,’isGuessed’,’playerCount’ és a ’GameIsOn’.

A jelenlegi időzítőt leállítja, a felhasználókat újra megjeleníti immáron friss adatokkal, ezen felül meghívja a ’showRollDown’ függvényt a megfelelő adatokkal.

’paint_to_user’**:** Amikor a rajzoló festési adatot küld a szervernek, az tovább lesz küldve a többi felhasználónak ezzel címkével. Megkapja a felhasználó milyen eszközt használt a felhasználó és az azzal kapcsolatos adatokat. Mindig a eszköznek megfelelő függvényt hívja meg a script.

’new-message-to-user’**:** Amikor új üzenet érkezik a szerverre, azt feldolgozza, majd ezzel a címkével küldi tovább. Megkapjuk a felhasználó nevét, az üzenetet és az üzenet típusát. A függvény helybe generál egy szövegdobozt, amelybe elhelyezi a felhasználó nevét és üzenetét, ami még ellátt a megfelelő osztálykijelölővel. Ezután ezt hozzá adja a csevegő részhez.

’paint_data_request’**:** Ezzel a címkével csak az aktuális rajzoló találkozhat, ugyanis ilyenkor egy újonnan csatlakozott felhasználó kéri az eddig lerajzolt műről az adatokat. Ebben az esetben a felhasználó visszaküldi a kívánt adatot a szervernek ’paint-data-to-server’ címével.

’paint-data-to-user’**:** Ezzel pedig csak a frissen csatlakozott felhasználó találkozhat, aki menet közben lépett be. Ilyenkor egy tömböt kap, tele rajzolási adatokkal, Itt egy ciklussal végig megy az összes adaton, majd minden rajzolási adatnál a megadott eszköznek megfelelő függvényt hívja meg. Toll esetében a ’Draw’, radír esetébe pedig az ’Erase’ függvényt hívja meg.

```
const loop = () => {
    if(isDrawing){
        if(window_width != window.innerWidth){
            console.log("Window size changed!");
            canvas_rect = canvas.getBoundingClientRect();
            window_width = window.innerWidth;
        }
        if(MyTimer == null && GameIsOn && playerCount > 1){
            console.log('This is true');
            MyTimer = setInterval(ChangeTimer,1000);
        }
        if(STATES.MOUSEDOWN && STATES.MOUSEPREV){
            if(TOOL == 'pen'){
                const offset = new vec2(canvas_rect.x,canvas_rect.y);
                const new_prev = Vec2.Sub(offset,STATES.PREV);
                const new_curr = Vec2.Sub(offset,STATES.CURR);
                Draw(ctx,COLORS[STATES.COLOR],new_prev,new_curr,canvas.width);
                    console.log("this code runs!");
                    Paint_Data[Paint_Data.length - 1].push({ tool : TOOL,color : COLORS[STATES.COLOR],prev : new_prev,curr : new_curr,width : canvas.width});
                    socket.emit('paint_to_server',TOOL,{room : roomid , color : COLORS[STATES.COLOR] , pos1 : new_prev , pos2 : new_curr , width : canvas.width});
            }
            else if(TOOL == 'eraser'){
                const offset = new vec2(canvas_rect.x,canvas_rect.y);
                const new_prev = Vec2.Sub(offset,STATES.PREV);
                const new_curr = Vec2.Sub(offset,STATES.CURR);

                Erase(ctx,new_prev,new_curr,canvas.width);
                Paint_Data[Paint_Data.length - 1].push({ tool : TOOL,color : COLORS[STATES.COLOR],prev : new_prev,curr : new_curr,width : canvas.width});
                socket.emit('paint_to_server',TOOL,{room : roomid , pos1 : new_prev , pos2 : new_curr , width : canvas.width});
            }
        }
        if(TOOL == 'trash'){
            socket.emit('paint_to_server',TOOL,{room : roomid});
            Paint_Data = [];
            Paint_Data.push([]);
            ClearCanvas();
            TOOL = 'pen';
        }
    }
    requestAnimationFrame(loop)
}
```

loop**:** Ez a függvény kulcs fontosságú szerepet játszik, ha a felhasználó rajzol, ellenkező esetben pedig nem sokat. A függvény több alkalommal is lefut egy másodperc alatt, lehetőleg 60 alkalommal. Alapesetben csak akkor néz bármit is ha a felhasználó rajzol.

Ilyen esetben figyeli, hogy változik-e az ablak mérete, ugyanis ez a rajzolásnál a ’Draw’ és ’Erase’ függvényben számít.

Itt indul el minden esetben a visszaszámláló is, ha néhány feltétel teljesül. Ilyen, hogy még nem indult egy sem, és fut a játék.

Ha a felhasználó ténylegesen rajzolni próbál, akkor a kiválasztott eszköz alapján elvégez néhány folyamatot. Toll esetén számol egy ’offset’ értéket, ez egy vector amiben szerepel mennyire kell eltolni a kurzor pozícióját annak érdekében, hogy az a vásznon a kurzor alatt jelenjen meg. Meghívja a ’Draw’ függvényt, emellett el is menti magának a rajzolási adatot abban az esetben, ha egy új játékos elkérné. Ezután továbbítjuk a szervernek ’paint-to-server’ címkével a szükséges adatot**.**

Radír esetében ugyan az a folyamat, csak a rajzolási adatban az eszköz radír lesz.

Egyedül a szemetes esetében nem kell sokat számolni, ilyenkor az eddigi elmentett rajzolási adatot mind kitöröljük, továbbítjuk a szervernek a szükséges adatot, majd letisztítjuk a vászont a 'ClearCanvas' függvény segítségével.

Továbbfejlesztési lehetőségek:

- Jelenleg a játék biztonság téren nem a legmegbízhatóbb, mivel több lehetőség is fennáll, hogy játékon belüli szabálytalanságot kövessenek el a játékosok. Ennek a továbbfejlesztése az egyik legfontosabb feladat lenne.
- A program hatékony, de nem eléggé. Néhány függvény átírásával a kódot gyorsabbá lehetne tenni.
- A kliens oldal további szépítést igényel. Nem csúnya, de biztosan sokkal szebb és letisztultabb eredményt lehetne létrehozni.
- Mivel az eredeti játék sok kijelző méreten is megállta a helyét, cél még a játék kiterjesztése más eszközökre is, emellett, hogy több webböngésző is támogassa azt.
- Jelenleg a most használt adatbázis jól teljesít, de majdnem felesleges, hisz ezen projekt esetében az adatbázis nem lenne szükséges, és így is ez veszi el a szerveren futó folyamatok jelentős részét.

Források:

https://code.visualstudio.com/

https://nodejs.org/en

https://socket.io/

https://www.mongodb.com/

https://stackoverflow.com/

https://www.youtube.com/watch?v=afCVHB2xm-g&pp=ygUXc2tyaWJibCBpbyB3aXRoIGZsdXR0ZXI%3D