Miskolci Egyetem
Gépészmérnöki és Informatikai Kar
Alkalmazott Matematikai Intézeti Tanszék

Szakdolgozat

Miskolci Egyetem

Személyre szabható online társasjátékok

Készítette:

Galán Levente

Programtervező informatikus BSc

Témavezető:

Piller Imre, egyetemi tanársegéd

Miskolc, 2018

Miskolci Egyetem
Gépészmérnöki és Informatikai Kar
Alkalmazott Matematikai Tanszék **Szám:**

Szakdolgozat Feladat

Galán Levente (MZAWSM) programtervezõ informatikus jelölt részére.

A szakdolgozat tárgyköre: Kétszemélyes online társasjátékok tervezése

A szakdolgozat címe: Személyre szabható online társasjátékok

A feladat részletezése:
A dolgozat egy olyan webalkalmazást mutat be, amelyen keresztül a felhasználók különféle társasjátékokkal játszhatnak. Az alkalmazás egyediségét az adja, hogy a játékosok a szabályokat meg is tudják változtatni. A játékosok alapvetően egymás ellen játszanak. Rendezhetnek mérkőzéseket, rangadókat, amely alapján az online felületen megtekinthetők a játékosok ranglistája, illetve különféle statisztikái.

Témavezetõ(k): Piller Imre, egyetemi tanársegéd

A feladat kiadásának ideje:

. . . . . .
szakfelelõs

Eredetiségi Nyilatkozat

Alulírott **Galán Levente**; Neptun-kód: `MZAWSM` a Miskolci Egyetem Gépészmérnöki és Informatikai Karának végzõs, Programtervező informatikus BSc szakos hallgatója ezennel büntetõjogi és fegyelmi felelõsségem tudatában nyilatkozom és aláírásommal igazolom, hogy *Személyre szabható online társasjátékok* címû szakdolgozatom/diplomatervem saját, önálló munkám; az abban hivatkozott szakirodalom felhasználása a forráskezelés szabályai szerint történt.

Tudomásul veszem, hogy szakdolgozat esetén plágiumnak számít:

- szó szerinti idézet közlése idézõjel és hivatkozás megjelölése nélkül;
- tartalmi idézet hivatkozás megjelölése nélkül;
- más publikált gondolatainak saját gondolatként való feltüntetése.

Alulírott kijelentem, hogy a plágium fogalmát megismertem, és tudomásul veszem, hogy plágium esetén szakdolgozatom visszautasításra kerül.

Miskolc, ... év ..... hó ..... nap

.....

Hallgató

1.

szükséges (módosítás külön lapon)

A szakdolgozat feladat módosítása

nem szükséges

....

dátum témavezetõ(k)

2. A feladat kidolgozását ellenõriztem:

témavezetõ (dátum, aláírás): konzulens (dátum, aláírás):

....

....

....

3. A szakdolgozat beadható:

....

dátum témavezetõ(k)

4. A szakdolgozat ..... szövegoldalt

..... program protokollt (listát, felhasználói leírást)

..... elektronikus adathordozót (részletezve)

.....

..... egyéb mellékletet (részletezve)

.....

tartalmaz.

....

dátum témavezetõ(k)

5.

bocsátható

A szakdolgozat bírálatra

nem bocsátható

A bíráló neve: .....

....

dátum szakfelelõs

6. A szakdolgozat osztályzata

a témavezetõ javaslata: .....

a bíráló javaslata: .....

a szakdolgozat végleges eredménye: .....

Miskolc, ...

a Záróvizsga Bizottság Elnöke

Tartalomjegyzék

1. fejezet

Bevezetés

Fejlődő, változó világban élünk, amelyben fejlődik a játékipar is. A napjainkban divatos játékokban a szép grafika, és a nagy méretű pályák jelentik a fő szempontot a játékok megítélése szempontjából. A dolgozat ezzel szemben a hagyományos táblás játékokra koncentrál, amelyek így klasszikus értékeket képviselnek a számítógépes játékok között. Ezek segítik a játékost megtanulni koncentrálni, több lépésre előre gondolkodni szórakoztató formában.

A dolgozatban a jól ismert szabályrendszerek kiegészítési módjára szerepel egy sajátos megközelítés. Ezzel lehetőség adódik a játékosoknak személyre szabni az aktuális játékot. A játékosok igényeitől függően így egyszerűbb, vagy bonyolultabb játék lehet a végeredmény. A bevezetett új szabályokkal a játék jellege lényegesen megváltozhat, így az addig alkalmazott stratégiák már nem minden esetben lesznek alkalmazhatók.

A számítógépes játékok egy kézenfekvő változatát jelentik azok az online játékok, amelyek tulajdonképpen webalkalmazások. A bemutatásra kerülő társasjátékok Java Spring keretrendszer és AngularJS segítségével készültek el. A dolgozat részletesen kifejti a játékok kiválasztásánál szereplő szempontokat, a specifikáció, tervezés és fejlesztés lépéseit.

2. fejezet

Táblás játékok specifikációja

2.1. Általános leírás

A projekt célja egy olyan webalkalmazás elkészítése, ahol emberek emberek ellen játszhatnak klasszikus két személyes társasjátékokat. A játék a játékok webes felületen, böngészőben, telepítés nélkül használhatóak lesznek.

Az alkalmazásban megtalálható játékok:

- amőba,
- dáma

A játékok különlegességét ez esetben az adja, hogy előre definiált opciókat kiválasztva a játékosok megváltoztathatják a játékszabályokat. A játék így humoros, izgalmas helyzeteket teremthet.

Az oldalon csak regisztrált felhasználók játszhatnak. A regisztráció előnye, hogy a felhasználó az általa megadott névvel szerepelhet a játékban, illetve statisztikákat, rangsorokat lehet összeállítani. A felhasználó saját statisztikái segíthetik a többi játékost a megfelelő partner kiválasztásában, például képet kaphatnak az ellenfél erősségéről. A rangsorok egyrészt érdekesek, másrészt meghozzák a kedvet a versengéshez, a még több játékhoz.

Az alkalmazás egyik menüpontjában pedig találunk majd bemutatkozó leírást, az eredeti játékok szabályait, leírást a lehetséges változtatásokról, és lehetőséget a kapcsolatfelvételre (email formájában) észrevételek, hibák bejelentésére.

2.2. Amőba

2.2.1. Játékszabály

Az amőba a legismertebb és legegyszerűbb táblás játékok egyike. Általában négyzetrácsos papíron játsszák. A négyzetháló mérete lehet egyenlő a rendelkezésre álló papír méretével, de kisebb terület is kijelölhető.

A játékosok felváltva helyeznek egy-egy jelet a tábla valamelyik, még üres négyzetébe. Mindenki a saját jelével játszik. (A leggyakoribb jelek az X és az O.) Az nyer, akinek sikerül saját jeleiből ötöt egyenes vonalban - vízszintesen, függőlegesen vagy átlósan - egymás mellé helyeznie [5].

2.2.2. Személyre szabhatóság

Személyre szabhatóság szempontjából a legjobb alapanyag. Nézzük, mi mindennel lehet érdekesebbé tenni a játékot!

- Tiltott mezők:

  A pályán olyan mezőket helyezünk el véletlenszerűen, amelyre nem rakhatunk karaktert.

- Csapda mezők:

  A pályán olyan mezőket helyezünk el véletlen szerűen, amelyekre, ha karaktert tesznek, a mező bepirosodik pár másodpercre és a karakter nem kerül kirajzolásra (végez vele a csapda). Ez után a kör átadódik a másik játékosnak. A csapda aktiválódása után a mező semlegessé válik, tehát ugyan úgy lehet rá karaktert tenni, mint bármelyik üres mezőre.

- Eltüntető karakter

  A játék kezdetekor mindkét játékos kap néhány különleges karaktert, amelyet letéve "kiradírozhat" 1 vagy több karaktert a pályáról.

- Eltűnő karakterek:

  Minden 3-5 kör után mindkét játékosnak eltűnik egy-egy véletlen szerűen kiválasztott karaktere.

- Elhelyezhető csapdák

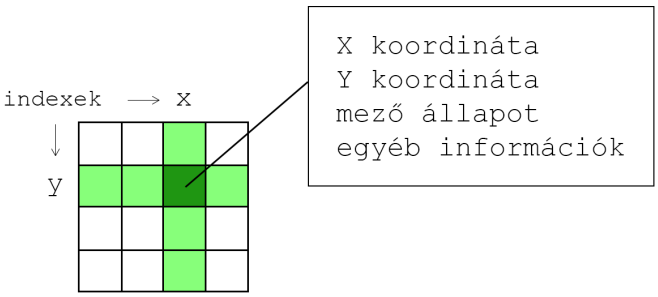
  A játék kezdetekor mindkét játékos kap néhány karaktert, amelyet letéve csapdát helyezhet el.

2.2.3. Tábla reprezentáció

A táblának a megrajzolása egyszerű, csak egy négyzetrácsra van szükség (továbbiakban vászon). A mezők számontartása és indexelése már érdekesebb. Esetünkben egy három dimenziós objektumban rögzítjük a mezőket. Első dimenzió reprezentálja az x sorindexet a vízszintes léptetéshez, második dimenzió az y oszlopindexet függőleges léptetéshez, a harmadik pedig a szükséges adatokat tartalmazó objektumokat jelenti. Szemléletesebb megfogalmazásban lesz egy tömbünk, ami tömböket tárol, amikben adat objektumok lesznek. Ezekben az adat objektumokban tároljuk el a mező koordinátáit a vásznon, a mező állapotát számmal jelölve, és szükség esetén bármilyen a mezőre vonatkozó információt. Egy tábla elvi megjelenítését mutatja be a 2.1 ábra.

A mezők állapotai az alábbiak lehetnek:

- 0 - semleges
- 1 - kezdő játékos
- 2 - másik játékos
- 3 - tiltott mező
- 4 - csapda mező



2.1. ábra. *Tábla reprezentáció*

Fontos különbséget tennünk az index és a koordináták között. Az index az elméleti, a koordináták a fizikai elhelyezést jelölik. Utóbbi a kirajzoláshoz szükséges.

Tehát a 2. sor 3. oszlopában található mező állapotára így hivatkozhatunk: `mező[3][2].állapot`.

2.2.4. Szabály ellenőrzés

Az első dolog, amit meg kell vizsgálnunk, amikor egy játékos letesz egy karaktert, a mező állapota. Ilyenkor lekérjük a kiválasztott mező indexeit, majd ez alapján kikeressük a mező értékeinket tároló objektumból a mező állapotát. Ha a mező semleges, kirakjuk a mezőre a karaktert. Ha csapda, akkor annak megfelelően járunk el. Ha már egy játékos karakterét tartalmazza, vagy a mező tiltott, nem történik semmi, nem kerül ki a karakter, a játékosnak másik mezőt kell választani.

Ha sikerült kitenni a karaktert, ellenőriznünk kell, hogy nem nyert-e a játékos. Ránézésre ugyan nyilván való, és egy szempillantás alatt látszik az eredmény, de hogy magyarázzuk el a számítógépnek?

A nyertes kombinációk kijöhetnek vízszintesen, függőlegesen, vagy átlósan (ebből kettő is van). Tehát ezeket a vonalakat kell megnézni. Ezt tehetjük úgy is, hogy végig nézzük az összes ilyen vonalat, hogy van -e nyertes kombináció a táblán. Egy nagy pálya esetén ez illetlenül erőforrás igényes lenne. Mivel minden karakter lerakást ellenőrzünk, így elég mindig csak a legutoljára letett karaktert és környezetét vizsgálni. Így a vizsgálandó területet máris lecsökkentettük egy 9×9 nagyságú négyzetre. Elkezdjük számolni a játékos karakterét, ha elérjük az öt egyforma karaktert, a játékos nyert. Ebben az esetben az ellenőrzést megszakíthatjuk, nincs már szükség a többi irány ellenőrzésére. Előfordulhat azonban, hogy az öt egyforma karakter nem egymás mellett helyezkedik el, ezért, ha olyan karakter következik, ami különbözik, a számlálót 0-ra kell állítani.

Még egy szempont az ellenőrzés során a pálya széle. Ebben az esetben a vizsgálandó terület túlnyúlhat a pályán, és ebben az esetben az indexek negatívvá válhatnak, így nem létező indexű elemeket keresnénk. Ennek kiküszöbölésének egy módja, ha minden irány ellenőrzése előtt kiszámoljuk az érintett mezők kezdő és záró indexeit a pályán, és csak ezeken iterálunk végig. Erre nehéz általános képletet adni, így én mégis megengedem a negatív indexeket, de az ilyen mezőket kihagyom az ellenőrzésből, ill. nem nyertes mezőként kezelem.

2.3. Dáma (hagyományos)

2.3.1. Játékszabály

A dámajáték a világon mindenütt elterjedt, közismert táblás játék. „Kockás” (négyzetrácsos) táblán játsszák, lapos, korong alakú bábukkal. Sokszor sakktáblát és sakkfigurákat használnak, de elterjedt a sakktáblánál nagyobb, 10x10-es tábla is.

A bábukat változattól függően két vagy három sorban kell felállítani, de az ellenfelek korongjai között legalább két sort üresen kell hagyni.

A hagyományos dámajátékban a sakkal ellentétben a sötét kezd, a játékosok itt is felváltva lépnek. Az egyszerű dámabábu mindig csak egyet léphet előre az ellenfél felé, és csakis átlósan, azonos színű mezőre (úgy tehát, ahogy a gyalog üt a sakkban). Az f3-on álló bábu tehát az e4 és g4 mezőkre léphet.

Ha azon a mezőn, ahová a bábu léphetne, az ellenfél egy bábuja áll, és azonos irányban még egy mezőt tovább haladva üres mező van, akkor ütéskényszer van: az ellenfél bábuját átugorva a mögötte lévő üres mezőre lépünk, az ellenfél bábuját pedig levesszük. Ilyen ütéshelyzet látható a 2. ábrán. A szabályokból értelemszerűen következik, hogy nem lehet leütni egy a tábla szélén álló bábut. A dámajátékban hagyományosan ütéskényszer van, azaz ha egy játékos tud ütni, akkor kötelező ütnie. Ha több ütési lehetősége is van, szabadon dönthet, hogy melyiket választja. Ezt a szabályt azonban nem mindig alkalmazzák.

Ha az ütés után a bábu ismét ütéshelyzetbe kerül, azt az ütést is végre kell hajtani, így ütéssorozat jön létre. Az ütéssorozat addig folytatható (ill. ütéskényszer esetén folytatandó), amíg leüthető bábu van. A 3. ábrán látható táblarészleten sötét egy két ütésből álló ütéssorozatot hajthat végre az X-szel jelölt mezőkre lépve, világos mindkét bábuját leütve.

Az a bábu, amelyik eléri a tábla szemközti sorát, dámává változik (az ábrákon a kis koronával megjelölt korongok jelzik a dámákat). A dámák szabadabban mozoghatnak, mint az egyszerű bábuk: minden dáma léphet és üthet hátrafelé is (ellenkező esetben nem is tudná elhagyni a szélső sort). Elterjedt, bár nem mindig alkalmazott szabály az is, hogy a dáma „nekifutásból” üthet. Erre is három variáció van: az ütést megelőzően, a leütendő bábu előtt tetszőleges számú üres mezőt átugorhat; az ütést követően, a leütendő bábu mögött tetszőleges számú üres mezőt átugorhat; az előző kettő kombinációja.

A baloldalt látható 4. ábrán az 1. esetben a sötét dáma az X-szel jelölt c6-os mezőre léphet, a 3. esetben ezen felül léphet a fekete körökkel jelölt b7-es és a8-as mezőkre is. Egy másik szabályvariáció szerint a dáma egyetlen lépéssel több bábut is le tud ütni akkor is, ha azok között nincs üres mező. Egy ilyen helyzet látható az 5. ábrán. Ennek a szabálynak az alkalmazása független az előző, nekifutásos szabálytól. Ha például az ütés utáni „továbbfutás” és több-bábus ütés egyaránt engedélyezett, a sötét dáma a b7 vagy a8 mezőkre is léphet, mindhárom világos bábu leütésével.

Ha egy bábu ütéssel éri el az utolsó sort, a szabályoknak megfelelően dámává válik. Ha ebből a helyzetből – immár dámaként – újabb ütésre nyílik lehetőség, az ütéssorozat azonnal folytatható.

Az nyeri meg a játékot, aki lépésképtelen helyzetbe szorítja ellenfelét (ez azt is jelentheti, hogy az ellenfélnek már nincsen bábuja, mindet leütöttük) [4].

2.3.2. Személyre szabhatóság

A dámában az egyéni változtatások mellett néhány, az amőbában említettet is érdemes alkalmazni.

- Csak dámák:

  Már a játék kezdetekor minden bábu dáma. Tehát hátra felé is léphet és üthet.

- Nekifutás:

  A játékszabályzatban szó volt a "nekifutásból" ütés 3 formájáról. Hogy lehessen -e nekifutás, és ha igen, akkor melyik, ennek eldöntését a játékosok eldönthetik akár maguk is.

- Csapda mezők:

  A pályán olyan mezőket helyezünk el véletlen szerűen, amelyekre, ha karaktert tesznek, a karakter kitörlődik (végez vele a csapda).

- Elhelyezhető csapdák

  A játék kezdetekor mindkét játékos kap néhány karaktert, amelyet letéve csapdát helyezhet el.

3. fejezet

Tervezés

A fejezet bemutatja a játéknak, mint több felhasználós, online elérhető webalkalmazásnak a tervezési folyamatát. Ez tartalmazza az alkalmazással szemben támasztott funkcionális követelményeket.

3.1. Általános funkciók

3.1.1. Belépés az oldalra

Az oldalra való belépéskor egy kezdőképernyő jelenik meg, ahol a játékos bejelentkezik a saját profiljába. A bejelentkezéshez szükséges megadni a felhasználónevet és a hozzá tartozó jelszót. Ha a felhasználó még nem regisztrált az oldalon, akkor a regisztráció gombra kattintva ezt is megteheti, majd ha a regisztráció sikeres volt, bejelentkezhet. A regisztrációhoz egy felhasználónév, emailcím (kétszer beírva), és jelszó megadása szükséges. Regisztráció és bejelentkezés után elérhetővé válik a személyes statisztika és megjelenés a rangsorban.

3.1.2. Új jelszó igénylése (kijelentkezve)

Előfordul, hogy a felhasználó elfelejti felhasználónevét vagy jelszavát. Ebben az esetben lehetőség van rá, hogy új jelszót igényeljen. Ennek menete, hogy rákattint az "Elfelejtettem a jelszavamat" gombra/linkre, megadja felhasználónevét vagy emailcímét. Ezután emailt küldünk a megadott helyre. Ez az email tartalmaz egy kódot és a játékos felhasználónevét, és egy linket egy oldalra, ahol meg kell adnia az emailben küldött kódot, a felhasználónevét, és az új jelszavát kétszer. A "jelszó megváltoztatása" gombra kattintva az adatok validálódnak, és siker esetén a játékos bejelentkezhet immár új jelszóval.

3.1.3. Jelszó megváltoztatása (bejelentkezve)

A jelszó megváltoztatására lehetőség van bejelentkezett állapotban is. Ez az egyszerűbb eset. A profil ill. beállítások oldalon a "jelszó megváltoztatása" gombra kattintva megjelenik egy űrlap, ahol meg kell adni a régi jelszót, majd az új jelszót kétszer. Az "ok" gombra kattintva az adatok validálódnak, és legközelebb már az új jelszóval léphet be a felhasználó.

3.1.4. Rangsorok, szabályzat, készítői információk megtekintése

Ezek statikus oldalak, melyek információt biztosítanak a felhasználó számára. Interaktivitásra itt nincs lehetőség. A rangsorokat dinamikusabbá lehet tenni, ha lapozós felületet biztosítunk neki, így egyszerre csak egy kiválasztott rangsort jelenít meg. A játék során készített statisztikák:

- összes játszma (globális),
- összes játszma játéktípusonként (globális),
- személyes (a felhasználó saját statisztikái)

A felhasználó saját statisztikái és rangsorai megjelennek a profilon is. Továbbá az elérhető játékosok listáján a többi (éppen bejelentkezett) felhasználó statisztikáit is megtehetjük, amely segít annak felmérésében, hogy a megfelelő szintű játékost választhassuk ellenfélnek.

3.1.5. Hibák észlelése, jelzése

Egy szoftver megírása során rendkívül fontos, hogy az elkészült terméket alaposan tesztelje, mielőtt élesben kiadja azt a felhasználóknak. De alapos tesztelés után is gyakori, hogy maradnak benne kisebb-nagyobb hibák, - akár figyelmetlenségből, akár hardverkülönbségek miatt, akár egy eset különlegessége miatt, stb., - amiket már csak a felhasználók tapasztalnak, vesznek észre. Olyan is lehet, hogy a szoftver kiválóan működik, de egy-egy felhasználónak akad egy jó ötlete, hogy hogyan lehetne még javítani rajta a jobb felhasználó élmény, biztonság, teljesítmény érdekében. Ezek mind-mind fontos információk a fejlesztő számára, így lehetőséget kell teremteni a felhasználóknak, hogy ezeket mind elmondhassák.

E célból kell egy olyan oldal, ahol írhatnak nekünk. Ehhez megadhatunk egy emailcímet is az oldalon, és a felhasználó írhat a saját levelező rendszeréből, de a manapság felhasználói szempontból ez már "túl sok kattintásnak" számít, így érdemesebb egy űrlapot készíteni, ahova a felhasználó leírja az észrevételeit, megnyomja a "küldés" gombot, és már kapjuk is az emailt.

3.1.6. Játék indítása

A játék indítás oldalon több féle képpen is indíthatunk játékot. Legegyszerűbb módja, ha a kihívás listából kiválasztunk egyet. Az "elfogadás" gombra kattintva már indul is a játék.

Ha nem találunk kedvünkre való kihívást, mi magunk is létrehozhatunk egyet. Kiválaszthatjuk, hogy melyik játékkal szeretnénk játszani, és milyen szabályokkal. Létrehozás után a kihívásunk bekerül a listába, és várjuk, míg partnerünk akad. Ha menet közben meggondoljuk magunkat, és mégse szeretnénk kihívást létrehozni, lehetőség van a létrehozás megszakítására, ("megszakítás" gomb). Elfogadás után indul a játék.

3.1.7. Kihívás törlése

Ha létrehozás után meggondoltuk magunkat, és mégsem szeretnénk a kihívásunkat (pl. nincs rá partnerünk vagy változtatnánk a szabályokon), lehetőségünk van kitörölni

a listából a "kihívás törlése" gombbal. Ezután létrehozhatunk egy új kihívást, vagy elfogadhatjuk másét. Továbbá, ha van kihívásunk a listában, és közben elfogadjuk más kihívását, a miénk automatikusan törlődik.

3.1.8. Interaktivitás bábúkkal

Miután betöltődött a játék, a játékosok felváltva, ún. körönként játszanak. Interaktivitásra a saját körünkben van lehetőségünk a játék szabályoknak megfelelően, pl. egy bábú/karakter elhelyezése a táblán, lépés egy bábuval.

3.1.9. Játszma feladása

Ha a vége előtt ki szeretnénk lépni a játékból, a "feladás" gombra kell kattintatunk. Ebben vége a játéknak, az ellenfél nyer, megjelennek az eredmények, majd visszatérhetünk a "játék indítása" felületre.

3.2. Adatbázis

Az adatbázisban két fontos szereplő típust, illetve azok állapotait kell nyilvántartanunk: a felhasználókat és a játékmeneteket. Habár közöttük is van kapcsolat, külön bemutatható és részletezhetők a sémáik.

3.2.1. Felhasználói adatok

A felhasználóhoz tartozó legfontosabb információk, amiket tárolni kell, a felhasználó:

- felhasználó neve,
- email címe,
- jelszavához tartozó hash kód,
- be van-e jelentkezve.

3.2.2. Meccsek adatai

A meccsek adatait érdemes állapotuk, ha úgy tetszik, életciklusuk szempontjából csoportosítani. Egy meccs életének fázisai:

- Kihívás:

  Első a kihívás, amit egy felhasználó hoz létre jelezvén, hogy játszani szeretne. Ez után akár meg is szűnhet, ha a játékos megszakítja (például, hogy új beállításokkal indíthasson játékmenetet), kijelentkezik, vagy ő maga egy másik kezdeményezésre jelentkezik.

- Aktív játékmenet:

  A kihívás aktív játékmenetté válik, ha egy játékos elfogad egy játékmenetet.

- Lejátszott meccs:

  A játék a végén pedig lejátszott meccs lesz. Az előző állapot minden állapotváltáskor törlődik egy kivétellel: a lejátszott meccsek adatai később sem tűnnek el, adataira statisztikai okokból később szükségünk lesz, ugyanis ezek alapján lehet felállítani a rangsorokat és az egyéni statisztikákat.

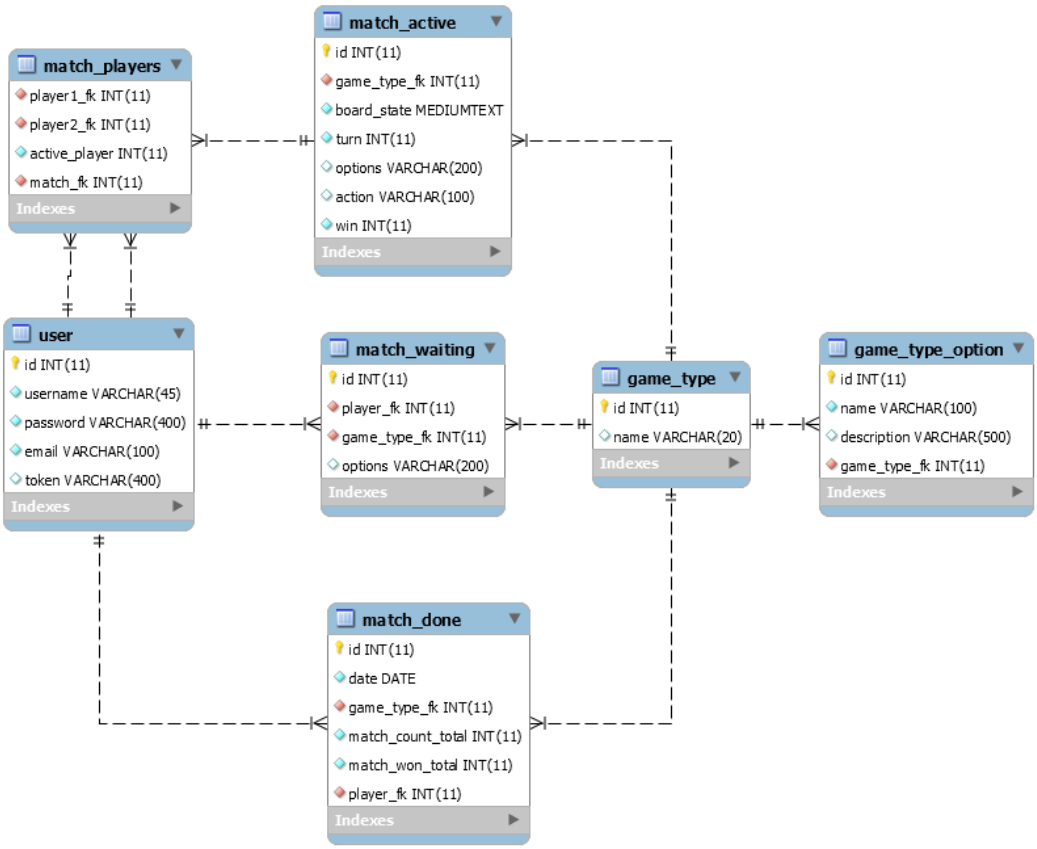
Életciklus alapján tehát megkülönböztetünk: kihívást, aktív játékmenetet, és lejátszott meccset.

Kihíváskor még nem kell sok információ: a kezdeményező játékos neve, a játék típusa, és a játék paraméterei.

A lejátszott meccseknél érdemes az azonos napon játszott, azonos típusú meccseket egy-egy rekordba összevonni, így egy-egy rekord tartalmazza, hogy egy-egy felhasználó melyik napon, milyen típusú játékban hány meccset játszott és hányat nyert meg.

Az aktív játékmenethez tartozik a játék azonosítója, típusa, a játék állása (Json sztringben), a játékosok azonosítói, nevei, és hogy éppen ki jön (aktív játékos). Habár egy játékoshoz egyszerre csak egy játékmenet tartozik, egy játékmenet több játékost is kezel egyszerre amellett, hogy vannak egyszer szereplő adattagjai is. Ennek kezelésére a játékosokat külön táblába emeltem.

Ezen szempontok alapján felvázolhatjuk adatbázisunk tábláit, melyet a 3.1 kép szemléltet.



3.1. ábra. *Az adatbázis ER modellje*

3.3. REST API

Ahhoz, hogy a frontend alkalmazásunk kommunikálni tudjon a szerverrel, egy kommunikációs felületet kell definiálnunk. Ezt a felületet interfésznek nevezzük, és itt kell megadnunk, hogy milyen kérésre mi a teendő, és hogy mi legyen a válasz. Ennek a módszernek még egy nagy előnye, hogy a komponensek, a frontend és a backend könnyen cserélhetővé válik. Feltétele csupán, hogy helyesen implementálja az interfészt.

Napjainkban nem divat a szerver oldalon generált weboldal, webkomponensek átküldése. Már csak az erőforrást (pl. egy lekért adathalmazt) küldenek át, és az oldal a frontenden generálódik. Ez az alapja a REST APInak is. Gyakran bevetett szokás ilyen környezetben, hogy a HTTP metódusokat (GET, PUT, stb.) az ún. CRUD műveletekhez rendelik. A CRUD a create (létrehozás, beszúrás), read (olvasás, lekérdezés), update (frissítés, módosítás) és delete (törlés) angol szavak kezdőbetűiből áll össze. A GET kérés megfelel a readingnek, a POST a létrehozásnak, a PUT a módosításnak, és a DELETE a törlésnek.

Az endpointokat hívhatjuk magyarul végpontoknak is. Egy endpointot tekinthetünk egy csomópontnak, vagy egy erősebb szálnak, ami segít csoportokba rendezni az elérhető szolgáltatásokat. Egyúttal megadják, hogy ezeket a szolgáltatásokat konkrétan milyen webcímen érhetjük el. Az alkalmazás az alábbi endpointokat tartalmazza:

- /user/: A felhasználó ki- és beléptetését, regisztrációját, stb. segíti.
- /match/: A játék elindítást kezeli.
- /fiveinarow/: A malom játékot vezérli.
- /checkers/: A dáma játékot vezérli.
- /stats/: Lekérdezi és összeállítja a statisztikákat.

3.3.1. /user

`POST`: Validálja a felhasználó által megadott értékeket, és elmenti az új profilt az adatbázisba. Siker esetén megerősítő üzenetet küld a mentés sikeréről. Sikertelen művelet esetén hibaüzenetet küld a felhasználónak (3.1 ábra).

```
REQ = {
        'username': <str>,
        'password': <str>,
        'passwordConfirmed': <str>,
        'email' : <str>
}
RES = {
        'success': <boolean>
}
```

Listing 3.1. */user POST kérés és válasz minta*

`PUT`: Validálja a felhasználó által megadott értékeket, és elmenti az új értékeket az adatbázisba. A felhasználó egyelőre csak a jelszavát tudja megváltoztatni. Siker esetén megerősítő üzenetet küld a mentés sikeréről. Sikertelen művelet esetén hibaüzenetet küld a felhasználónak (3.2 ábra).

```
REQ = {
        'userid': <int>,
        'oldPassword': <str>,
        'newPassword': <str>,
        'newPasswordConfirmed': <str>,
}
RES = {
        'success': <boolean>
}
```

Listing 3.2. */user PUT kérés és válasz minta*

`DELETE`: Validálja a felhasználó által megadott értékeket, (a törléshez szükség van a jelszó megadásához), majd törli a felhasználót az adatbázisból. Gyakran bevett szokás, hogy a felhasználó rekordját valójában nem törlik ki, hanem csak egy plusz mezőben átírják az állapotát aktívról töröltre. Ez főként olyan alkalmazásoknál jelentős, ahol fontos a visszakereshetőség biztosítása akár több évre visszamenőleg is, pl. bankoknál. Esetünkben a felhasználó teljesen törlésre kerül. Siker esetén megerősítő üzenetet küld a törlés sikeréről. (Egyúttal illik kiléptetni a felhasználót). Sikertelen művelet esetén hibaüzenetet küld a felhasználónak (3.3 ábra).

```
REQ = {
        'userid': <int>
        'password': <str>
}
RES = {
        'success': <boolean>
}
```

Listing 3.3. */user DELETE kérés és válasz minta*

3.3.2. /user/authenticate

`POST`: Validálja a felhasználó nevét és jelszavát. Siker esetén elküldi az azonosító tokent. Sikertelen művelet esetén hibaüzenetet küld a felhasználónak. (3.4 ábra).

```
REQ = {
        'username': <str>,
        'password': <str>
}
RES = {
        'token': <str>
}
```

Listing 3.4. */user/authenticate kérés és válasz minta*

3.3.3. /user/forgottenpassword

`POST`: Validálja az adatokat, (ellenőrzi, hogy a felhasználó létezik-e az adatbázisban), majd küld egy megerősítő emailt a felhasználó címére, a linkkel, amin tovább haladva

megadhatja új jelszavát. Siker esetén elküldi, hogy a validáció sikeres. (Csak ez után jelenik meg az űrlap az új jelszó megadásához). Sikertelen művelet esetén hibaüzenet küld a felhasználónak (3.5 ábra).

```
REQ = {
          'username': <str>,
          'email'   : <str>
}
RES = {
          'success': <boolean>
}
```

Listing 3.5. */user/forgottenpassword kérés és válasz minta*

3.3.4. match

`GET`: Ellenőrzi a felhasználót, majd lekérdezi és válaszként elküldi a várakozó meccsek (kihívások) adatainak listáját (3.6 ábra).

```
REQ = {
          'userid': <int>
}
RES = {
          'matches': [
                    {
                              'userid': <int>,
                              'username': <str>,
                              'gameType'  : <str>,
                              'setup': [
                                        <int>,
                                        ...
                              ]
                    },
                    {
                              ...
                    },
          ]
}
```

Listing 3.6. */match/list kérés és válasz minta*

`POST`: Inicializál egy új kihívást és elmenti az adatbázisba. (Emellett gondoskodni kell róla, hogy a többi felhasználó számára is megjelenjen az új kihívás). A művelet végén visszaküldi a mentés sikerességét. Sikertelen művelet esetén hibaüzenet küld a felhasználónak (3.7 ábra).

```
REQ = {
          'match': {
                    'userid': <int>,
                    'username': <str>,
                    'gameType'  : <str>,
```

```
                'setup': [
                        <int>,
                        ...
                ]
        }
}
RES = {
        'success': <int>
}
```

Listing 3.7. */match/create kérés és válasz minta*

`DELETE`: Kitöröl egy kihívást és elmenti az adatbázisba. (Emellett gondoskodni kell róla, hogy a többi felhasználó számára is eltűnjön az új kihívás). Siker esetén elküldi, hogy a törlés sikeres. Sikertelen művelet esetén hibaüzenet küld a felhasználónak (3.8 ábra).

```
REQ = {
        'match': {
                'matchid',
                'userid': <int>
        }
}
RES = {
        'success': <boolean>
}
```

Listing 3.8. */match/delete kérés és válasz minta*

3.3.5. match/start

`POST`: Akkor hívódik meg, amikor egy játékos elfogad egy kihívást. Megkapja az ezidáig várakozó játszma és a játékos azonosítóját. Lekéri a várakozó játék adatait, létre hozz egy aktív játszmát, amit elment az adatbázisban is, majd törli a várakozó játékot. Ha bármelyik játékosnak volt aktív kihívása, azt törli az adatbázisból. Sikeres mentés esetén megerősítésképpen visszaküldi a meccs adatait tartalmazó objektumot. Sikertelen művelet esetén hibaüzenetet küld a felhasználónak (3.9 ábra).

```
REQ = {
        'userid': <int>,
        'matchid': <int>
}
RES = {
        'match': {
                'matchid': <int>,
                'players': {
                        'player1': <int>,
                        'player2': <int>,
                        'active_player': <int>
                },
```

```
                'turn': <int>,
                'winner': <int>,
                'gameTypeId': <int>
                'gameTypeName': <str>,
                'options': [
                        <int>,
                        ...
                ]
        }
}
```

Listing 3.9. */match/start kérés és válasz minta*

`GET`: Megkapja a játékos azonosítóját és ellenőrzi, hogy a játékos szerepel-e aktív játékban. Ha a válasz igen, elküldi a meccs adatait tartalmazó objektumot. Ha még nincs játékban, az objektum értéke `null` (3.10 ábra).

```
REQ = {
        'userid': <int>,
}
RES = {
        'match': {
                'matchid': <int>,
                'players': {
                        'active_player': <int>,
                        'player1': <int>,
                        'player2': <int>
                },
                'turn': <int>,
                'winner': <int>,
                'gameType' : <str>,
                'setup': [
                        <int>,
                        ...
                ]
        }
}
```

Listing 3.10. */match/checkstart kérés és válasz minta*

3.3.6. /fiveinarow/action

`POST`: Mikor a soron következő játékos elvégzi a lépést (vagy ezzel egyenértékű cselekvést, pl. elhelyez egy csapdát), megkapja a meccs állását és a cselekvés adatait. Ezek segítségével ellenőrzi a lépés helyességét és hogy nyert -e a játékos, majd elmenti az adatbázisba. Siker esetén elküldi a kiértékelés eredményét, hogy volt-e nyerés, és sikerült -e a mentés. Ha nem sikerült, hibaüzenetet küld (3.11 ábra).

```
REQ {
        'matchid': <int>,
```

```
        'fields': [
                [
                        {
                                'x': <int>,
                                'y': <int>,
                                'value': <int>
                        },
                        {
                                ...
                        }
                ],
                [
                        ...
                ],
        ],
        'action': {
                'x': <int>,
                'y': <int>,
                'value': <int>
        },
        'players': {
                'active_player': <int>,
                'player1': <int>,
                'player2': <int>
        },
        'turn': <int>
}
RES {
        status: <boolean>,
        win: <boolean>
}
```

Listing 3.11. */fiveinarow/action kérés és válasz minta*

3.3.7. /fiveinarow/checkaction

`GET`: Rendszeres időközönként meghívódik annak ellenőrzésére, hogy az ellenfél lépett-e már. Megnézi, hogy az adatbázisban tárolt kör száma nagyobb -e, mint a kapotté. Ha igen, visszaküldi a meccs objektumot, ha nem, akkor az objektum `null` (3.12 ábra).

```
REQ = {
        'userid': <int>,
        'matchid': <int>,
        'turn': <int>
}
RES = {
        'match': {
                'matchid': <int>,
                'players': {
```

```
                        'player1': <int>,
                        'player2': <int>,
                        'activePlayer': <int>
                },
                'turn': <int>,
                'winner': <int>,
                'gameType' : <str>,
                'setup': [
                        <int>,
                        ...
                ]
        }
}
```

Listing 3.12. */fiveinarow/checkaction kérés és válasz minta*

3.3.8. /fiveinarow/timeout

`POST`: Egy játékos inaktivitása esetén kap egy jelzést, majd a kört átadja az ellenfélnek (3.13 ábra).

```
REQ = {
        'userid': <int>,
        'matchid': <int>
}
RES = {
        'success': <int>
}
```

Listing 3.13. */fiveinarow/timeout kérés és válasz minta*

3.3.9. /fiveinarow/giveup

`POST`: Megkapja a felhasználó azonosítóját és a meccs objektumot, és elmenti az eredményt a meccs objektumba. Visszatérési értéke a mentés sikeressége (3.14 ábra).

```
REQ = {
        'userid': <int>,
        'matchid': <int>
}
RES = {
        'success': <int>
}
```

Listing 3.14. */fiveinarow/giveup kérés és válasz minta*

3.3.10. stats/global

`GET`: Lekéri a szükséges adatokat az adatbázisból, majd összeállítja és visszaküldi a rendezett ranglistát (3.15 ábra).

```
REQ = {}
RES = {
        globalStats: {
                'allMatches': <int>,
                'byGameType': [
                        <int>,
                        ...
                ]
        },
        globalRanks: [
                {
                        'userid': <int>,
                        'username': <str>,
                        'allMatches': <int>,
                        'winMatches': <int>,
                },
                {
                        ...
                }
        ]
}
```

Listing 3.15. */stats/global kérés és válasz minta*

3.3.11. stats/globalbygametype

`GET`: Lekéri a szükséges adatokat az adatbázisból, majd összeállítja és visszaküldi a rendezett ranglistát (3.16 ábra).

```
 REQ = {}
RES = {
        'stats': [
                {
                        'userid': <int>,
                        'username': <str>,
                        'results': [
                                {
                                        'gameType': <int>,
                                        'allMatches': <int>,
                                        'winMatches': <int>
                                },
                                {
                                        ...
                                }
                        ]
                },
        ]
}
```

Listing 3.16. */stats/globalbygametype kérés és válasz minta*

3.3.12. stats/personal

`GET`: Megkapja a felhasználó azonosítóját, majd lekéri a szükséges adatokat az adatbázisból, majd összeállítja és visszaküldi a rendezett ranglistát (3.17 ábra).

```
REQ = {
        'userid': <int>
}
RES = {
        'stats': {
                'all': {
                        'userid': <int>,
                        'username': <str>,
                        'allMatches': <int>,
                        'winMatches': <int>
                },
                'byGameType': [
                        {
                                'gameType': <int>,
                                'allMatches': <int>,
                                'winMatches': <int>
                        },
                        {
                                ...
                        }
                ]
        }
}
```

Listing 3.17. */stats/personal kérés és válasz minta*

3.4. Komponensek

A készülő alkalmazás felépítésének tervezéskor két szempontot érdemes figyelembe venni: az egyik a HTTP kliens-szerver architektúra, a másik az MVC architektúra.

3.4.1. HTTP kliens-szerver architektúra

Minden egyes HTTP protokollt használó kommunikáció esetében van egy kérés és egy válasz. Az ügyfél böngészője egyetlen kérést intéz a szerverhez, amely alapján a szerver meghatározza a kért erőforrást és válaszul visszaküldi a kért állomány tartalmát. Ezt követően az ügyfél böngészője értelmezi a választ és megjeleníti a kapott tartalmat. A kérés az igényelt erőforráson kívül más információkat is tartalmaz. Ilyen információ például az ügyfél böngészőjének a típusa. A kérés általában sima szöveges információ, a válasz pedig lehet szöveg, illetve bináris adat is [11].

3.4.2. MVC architektúra

Az alapgondolat az, hogy egy alkalmazás adatait, ezek megjelenítését, illetve ezek kapcsolatát három önálló egységre bontja szét, amelyek nevei: Model (modell), View (megjelenítés), illetve Controller (vezérlés). A szétbontás célja az, hogy rugalmas alkalmazásokat készíthessünk, például kicserélhessük az adatok megjelenítését a többi rész változtatása nélkül.

A modell részt rendszerint az adatok és ezek elérési logikája képezi. Ez a rész az, amely a legritkábban változik. A vezérlés a közvetítő a modell és a megjelenítés között, amely vagy aktualizálja a modellt, vagy pedig egy új nézetet jelenít meg. A megjelenítés pedig sokféle lehet. Ha időközben ugyanazon alkalmazáshoz egy új kommunikációs eszközt akarunk csatlakoztatni, akkor csak az ennek megfelelő megjelenítési réteggel kell kiegészíteni az alkalmazást. Tehát ennek a tervezési mintának a betartása is nagymértékben hozzájárul az alkalmazások rugalmasságának növeléséhez.

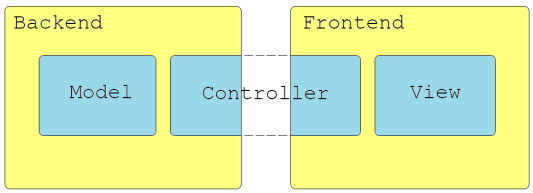
Az MVC tervezési minta először grafikus interfészek (desktop alkalmazások) készítésével kapcsolatban jelent meg, később alkalmazták például webalkalmazások architektúrájaként is. Webalkalmazások esetében a vezérlési réteg feladata a kérések feldolgozása és továbbítása a megfelelő komponens fele. Így a vezérlési rész felügyeli az alkalmazást, a modell pedig az alkalmazás állapotát tárolja. Összefoglalva, a következő előnyökkel rendelkezik egy MVC architektúrájú alkalmazás:

- Lehetővé teszi több, különböző megjelenítés illesztését ugyanazon modellhez.
- Szétválasztja az alkalmazás üzleti logika, adatelérés, illetve megjelenítés részeit, amely növeli az alkalmazás rugalmasságát [11].

3.4.3. Szerkezet összegzés

A két fent említett szerkezetet alkalmazva az alábbi alkalmazás vázat kapjuk:

A HTTP protokollnak megfelelően először definiáljuk a klienst (fronendet) és a szervert (backendet), majd ezekben helyezzük el az MVC komponenseket. Magától értetődő, hogy a View teljes mértékben a frontend része, a Model, - ami gyakorlatilag az adatbázist és adat kezelő logikát jelenti-, pedig a backendé. Az érdekes rész a Controller, mivel esetünkben a lépéseket és az űrlap adatait mindkét oldalon ellenőrizni kell, így a frontend és a backend is tartalmaz üzleti logikát (3.2. ábra).

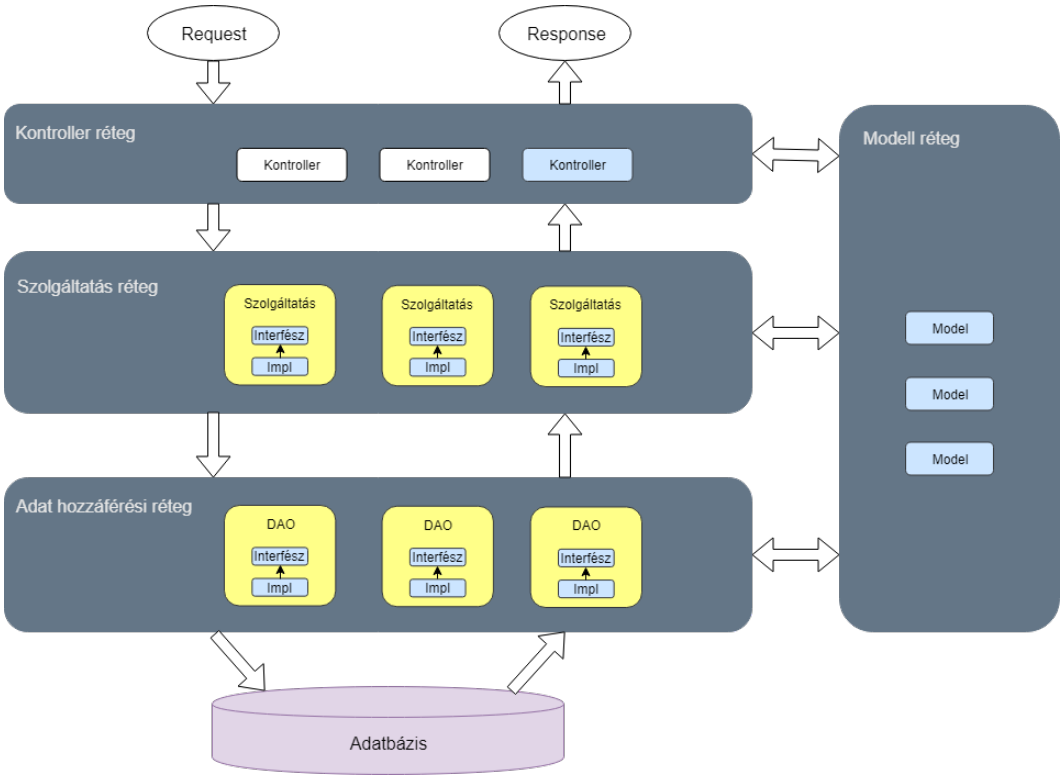


3.2. ábra. *Az MVC megoszlása a backend és a frontend között*

3.5. Backend felépítése

3.5.1. Réteges architektúra

A *backend* vagy más néven szerver oldali alkalmazás, egy kiszolgáló alkalmazás, amelyet (első sorban) webböngészővel érhetünk el. Feladata a kliens (frontend) által küldött adatok feldolgozása, majd a megfelelő válasz, ill. erőforrás visszaküldése. Mint általában a szoftvereknél, a szerver alkalmazást is érdemes jól elkülöníthető rétegekre bontani, melyet a 3.3 ábra szemléltet.



3.3. ábra. *Szerver rétegek*

Az első réteg, a web réteg, vagy más néven kontroller (controller) réteg. Mikor a böngésző a korábban már definiált API alapján elküld egy kérést a szervernek, az alkalmazás megkeresi a hívás címén keresztül a megfelelő kontrollert. A kontrollerek feladata a kérések fogadása és a válaszok visszaküldése. A kettő között azonban szükség van az adatok feldolgozására, melyet már a következő szinten, a szolgáltatás (service) rétegben végzünk. Ennek szolgáltatásait a kontrollerek veszik igénybe. Az adat feldolgozás során gyakran van szükség az adatok tartós tárolására és karban tartására, tehát szükség van egy adatbázisra is. Ahhoz, hogy a szolgáltatások és az adatbázis között létrejöhessen a kapcsolat, be kell hát vezetnünk még egy szintet, ez lesz az adat hozzáférési réteg (persistence). Feladata, a kapcsolat kiépítése az alkalmazás, majd az adatbázis között és elküldeni az adatbázisnak az elvégzendő műveleteket, és fogadni annak eredményét. Van még egy réteg, mely hierarchiában nem ennyire meghatározható, hiszen bármely réteg szükség szerint hozzáférhet, ez pedig a modell réteg. Ebben egyszerű objektumokat találunk, melyeknek a szerepe, hogy meghatározzák az objektumok struktúráját.

Az MVC architektúra előnyei között említettük a rugalmasságot, vagyis a komponensek cserélhetőségét. A szerver esetében is szokás ilyen szempont szerint tervezni.

Ennek a legfontosabb eszközei az interfészek, melynek segítségével a komponensek bármikor cserélhetőek úgy, hogy az alkalmazást csak egyetlen ponton, a példányosítás megadásakor kell módosítani. Ezek a pontokat rendszerint annotációk, (mára ez az elterjedtebb), vagy xml fájlok segítségével konfiguráljuk.

3.5.2. Kontrollerek meghatározása

Mint láthattuk az alkalmazásnak sok kisebb funkciója van külön fajta kérésekkel, de felesleges lenne minden kéréshez külön külön kontrollert fenntartani, így érdemes jellegük szerint csoportosítani őket, és néhány kontrollerrel kezelni őket. Így az alábbi kontrollereket kapjuk:

- felhasználói (továbbiakban UserController),
- meccs (továbbiakban MatchController),
- amőba (továbbiakban FiveInARowController),
- dáma (továbbiakban CheckersController).

Ezek mélyebb ismertetését a következőkben olvashatjuk.

UserController

A UserController fő feladata magával a felhasználó személyével kapcsolatos kérések kezelése. Ide tartozik

- a regisztráció,
- a bejelentkezés,
- a kijelentkezés,
- jelszó megváltoztatása,
- a elfelejtett jelszó kezelése,
- és a felhasználó törlése.

MatchController

A MatchController, ahogy a neve is sugallja a meccsek kezelésével kapcsolatos kéréseket dolgozza fel. Ennek feladata

- a kihívások (meccs kezdeményezések) létrehozása,
- törlése,
- a kihívások listájának lekérdezése,
- és az aktív játékmenet létrehozása elfogadáskor.

FiveInARowController

Az amőba kontrollere értelem szerűen már a megkezdett, amőba játéktípusú meccs kéréseit kezeli. Ezek az alábbiak:

- az aktív játékos lép,
- lépett -e az ellenfél,
- lejárt a lépéshez megengedett idő (timeout),
- játék feladása

CheckersController

Hasonló jellegű kéréseket kell teljesítenie, mint egyéb típusú játékok esetén, a játék sajátosságai miatt minden játéktípusnak külön kontrollerre és metódusokra van szüksége.

StatsController

Feladata a lejátszott meccsek adataiból összeállítani statisztikákat, rangsorokat, majd ezt az adathalmazt elküldeni a frontendnek.

3.5.3. Modell osztályok

Miután felvázoltuk az adatbázist és meghatároztuk az eltárolandó adatot, ez alapján rögzíthetjük a szerver oldalon szükséges modell osztályokat, amelynek a következőket kell tartalmaznia: azonosító, felhasználónév, a jelszó hashkódja, emailcím.

Felhasználó modell

Elsőként a User (felhasználó) osztállyal kell foglalkoznunk, ami minden más osztállyal kapcsolatban áll. (3.18 ábra).

```
User {
        id: <int>,
        userName: <string>,
        passwordHash: <string>,
        email: <string>
}
```

Listing 3.18. *A felhasználó modell osztály adattagjai*

Kihívás modell

A kihívás modell a várakozó meccseket reprezentálja, amelyeket a felhasználók hoznak létre. Tartalma a kihívás azonosítója, a felhasználó azonosítója, a játék típusa, és egy tömbben a kiválasztott beállítások. (3.19 ábra).

```
MatchWaiting {
        'id': <int>,
        'userId': <int>,
```

```
        'gameType': <enum>,
        'options': [
                <enum>,
                ...
        ]
}
```

Listing 3.19. *A kihívás modell osztály adattagjai*

Aktív meccs modell

Az aktív meccsek adatmezői a meccs, a játékosok azonosítói, a soron következő játékos száma (1 vagy 2), a játék típusa, a játék állása, és hogy hanyadik körnél tartunk (3.20 ábra). Mivel az adatbázisban az aktív meccsek játékosait külön táblába kellett emelnünk, a könnyebb konverzió érdekében ezeket a modellben is egy külön osztályba emeljük ki. A játék állását az adatbázisba mindig sztringként tároljuk el, ám a könnyebb kezelhetőség és az ellenőrzések érdekében objektumként is szükségünk van rá, ezért mindkét formában benne kell lennie.

```
MatchActive {
        'matchid': <int>,
        'gameType': <enum>,
        'boardState': <string>,
        'fields': <Fields>
        'players': <Players>
        'turn': <int>
}
```

Listing 3.20. *Az aktív meccs modell osztály adattagjai*

Játékosok modell

A játékosok modell szerkezete egyszerű, csak három azonosítót tartalmaz: a két résztvevő és a soron következő játékos számát (3.21 ábra).

```
Players {
        'player1': <int>,
        'player2': <int>,
        'activePlayer': <int>
}
```

Listing 3.21. *A játékosok modell osztály adattagjai*

Játékállás modell

Mivel a különböző játékokhoz különböző táblák tartozhatnak, így minden típusra érdemes külön modell osztályt írni.

Az amőba játékállás reprezentálását már a 2.2.3 pontban tárgyaltuk. Az így megadott mátrixban tárolt objektum halmazt a 3.22 ábra szemlélteti.

```
Fields {
        'fields': [
                [
                        {
                                Field,
                                ...
                        },
                        {
                                Field,
                                ...
                        }
                ],
                [
                        ...
                ],
        ],
}
```

Listing 3.22. *A játékállás modell osztály adattagjai*

Lejátszott játék modell

A statisztikák generálásához szükséges megőriznünk a játékeredményeket (3.23 ábra).

```
MatchDone {
        'matchid': <int>,
        'date': <date
        'gameType': <enum>,>,
        'matchCountTotal': <int>
        'matchWonTotal': <int>
        'playerId': <int>
}
```

Listing 3.23. *A lejátszott játék modell osztály adattagjai*

Felsorolások

A sztringek kezelése kissé nehézkesebb, mint a számoké, ugyanakkor érdemes rögzíteni, hogy melyik szám melyik játékra utal. Ezért a `játéktípusokat`, és az egyéni szabályok beállításait érdemes enumokban tárolni.

Mivel a különböző játéktípusokhoz különböző egyéni szabályok tartozhatnak, így a `játékszabályoknak` játéktípusonként egy enum osztályt kell készítenünk.

És mivel a pálya mezőinek is kötött, hogy milyen értékeket lehet adni, így játéktípusonként szükségünk lesz egy-egy `érték` enumra is.

3.6. Frontend alkalmazás részei

A *frontend* ez esetben a webböngészőben futó JavaScript alkalmazást jelenti. A következő szakaszokban ezen alkalmazás részeinek és működésének a bemutatására kerül

sor.

3.6.1. Komponensek

A frontend a korábban megfogalmazott igényeket figyelembe véve az alábbi oldalakból fog állni:

- bejelentkezés (főoldal),
- regisztráció,
- elfelejtett jelszó,
- bemutatkozás,
- játékszabályok,
- használati feltételek,
- profil,
- beállítások,
- játék indítás,
- aktív játéktér,
- statisztikák.

A játéktér kivételével minden oldalon megjelenik egy menüsor, ami segítheti a felhasználót a navigációban. Mivel bizonyos funkciókhoz bejelentkezés szükséges, így a menüsornak is alkalmazkodni kell a felhasználó állapotához. Ezt tokenek segítségével valósítjuk meg, melyről később bővebben is esik szó.

Továbbá a játéktérre csak a játék indítás oldalon keresztül lehet eljutni, az elfelejtett jelszóhoz és a regisztrációhoz pedig lesz egy-egy link a főoldalon, így a menüsorban ezek nem jelennek meg. Ha játéktérre a megengedettől eltérő módon lépne be valaki, a hiányzó adatok és ellenfél híján hibát kapnánk. Megoldásként, bármilyen hiba esetén a felhasználót átirányítjuk a játék indítás oldalra.

Vendég felhasználóként (tehát bárki számára) csak az alábbi linkek érhetők el:

- bejelentkezés,
- regisztráció,
- elfelejtett jelszó,
- bemutatkozás,
- játékszabályok,
- használati feltételek.

Bejelentkezés

Ez a főoldal, ami néhány mondatban köszönti a vendég felhasználót az oldalról, aki itt bejelentkezhet az alkalmazásba egy űrlapon keresztül. Sikeres bejelentkezés esetén megjeleníti a felhasználó profilját és a menüben megjelenik a kijelentkezés gomb. További linkek az oldalon:

- regisztráció,
- elfelejtett jelszó.

Elfelejtett jelszó esetén újabb űrlap jelenik meg, amivel a felhasználó elküldheti felhasználónevét és/vagy jelszavát.

Regisztráció

Egy formot jelenít meg, amelyben a felhasználó megadhatja a regisztrációhoz szükséges adatokat. Elküldés után a felhasználó visszajelzést kap a regisztráció sikerességéről.

Elfelejtett jelszó

Ez az oldal akkor jelenik meg, ha az elfelejtett jelszó kapcsán küldött emailben található linkre kattintunk. Itt a felhasználónak egy újabb űrlapon kétszer meg kell adnia az új jelszót. Elküldés után a mentés sikerességéről értesítést jelenít meg.

Bemutatkozás, Játék szabályok, Használati feltételek

Általános, tájékoztató jellegű leírások. Statikus adatokat megjelenítő oldalak, akció nincs.

Profil

(Felhasználó)név szerint köszönti a bejelentkezett felhasználót. Megjelenik egy menü a további lehetőségekről, és akár néhány személyes információ is (pl. avatar, nyerési arány, egy véletlen szerű vicc, stb).

Játék kiválasztása

Itt két fontos opciót találhatunk, amely a képernyőt két részre osztja.

Az egyik oldalon táblázat formájában böngészhetünk a meccsek közül. Ezek soronként tartalmazzák a kihívó játékos nevét, a játék típusát, és a lehetséges szabálymódosításokat. Ha találunk kedvünkre való meccset, kiválasztva azt, majd az "ok" gombra kattintva kezdhetjük is a játékot.

Ha egyik felkínált lehetőség sem tetszik, tekintsünk a másik oldalra, ahol magunk is létrehozhatunk egy meccset. Ezen az oldalon megjelenik egy menü a játéktípusok felsorolásával és a választható szabályváltoztatásokkal. Ha minden opciót beállítottunk, az ehhez az oldalhoz tartozó "ok" gombbal hozzáadhatjuk kihívásunkat az említett kihívás listához. Egy felhasználó egyszerre csak egy kihívást kezdeményezhet, ilyenkor a kihívás hozzáadása gomb tiltásra kerül. Saját kihívásunkat a lista más színnel jelöli, mint a többit. Saját kihívásunkat kijelölve megjelenik egy törlés gomb.

Innen kezdve 3 lehetőségünk lesz:

- várunk egy ellenfélre,
- töröljük a kihívásunkat, (majd létrehozhatunk egy másik kihívást),
- választunk egy meccset a listából, (ebben az esetben az általunk létrehozott meccs törlődik).

Játéktér

Megjelenik a játéktér a játéktípushoz tartozó interaktív elemekkel.

A játék végén (ha a játék szabályosan véget ért vagy az egyik játékos feladta) megjelenik az eredmény-hirdetés. A tovább gombra kattintva a Játék indítása oldalra visz.

Rangsorok

Itt jelennek meg a játék során készített statisztikák: összes játszma száma (globális) játszmák száma játéktípusonként (globális) nyerési arány összesen (felhasználónként) nyerési arány játéktípusonként (felhasználónként) A statisztikák elkészíthetők úgy is, hogy egyszerre csak egy (kiválasztott) listát mutasson.

Beállítások

A felhasználó bizonyos mértékig személyre szabhatja az alkalmazást. Egyelőre csak a jelszavát tudja majd megváltoztatni, de később implementálni lehet pl. avatar képek feltöltését, szín séma megváltoztatását, stb. A változások mentése előtt meg kell adni az érvényes jelszót.

A mentések sikerességéről visszajelzést jelenít meg (sikeres vagy hibaüzenet).

Kijelentkezés

A kijelentkezés gombra kattintva a felhasználó kijelentkezhet az alkalmazásból. Az alkalmazás ekkor törli a bejelentkezéskor kapott tokent, majd átirányítja a felhasználót a bejelentkezés oldalra. Külön oldal nem tartozik hozzá.

3.7. Alkalmazás logika

A komponenseket ugyan megterveztük, de rengeteg kérdés merül fel azzal kapcsolatban, hogy hogyan fognak ezek a komponensek együtt működni. Hogyan szinkronizáljuk a játék menetét, hogy mindkét játékos (lehetőleg) valós időben lássa a másik lépéseit? Szinkronizáljuk a böngészők eseményeit? Egyáltalán milyen feladatok merülnek fel, amikre a tervezés legelején nem gondolna az ember? Nézzük sorban!

3.7.1. A backend és a frontend kapcsolata

Az első és legfontosabb kérdés, hogy hogyan fogja látni az egyik böngésző, hogy mi történik a másikban, pl. ha lép a játékos, hogyan jut át az információ? Ahogy egy pincér egy jó étteremben, a szerver (pincér) kiszolgál, a kliens pedig rendel. Tehát minden eseményt vagy kérést a klienst küld el a szervernek, ill. ha a szerverhez befut egy információ, a szerver nem tolakodik a böngészőhöz, hogy átadja az információt,

sőt azt sem tudja, hogy a frontendnek szüksége van rá. Ezért az alkalmazásban minden kommunikációt a frontend kezdeményez.

A komponenseket ugyan megterveztük, de rengeteg kérdés merül fel azzal kapcsolatban, hogy hogyan fognak ezek a komponensek együtt működni. Hogyan szinkronizáljuk a játék menetét, hogy mindkét játékos (lehetőleg) valós időben lássa a másik lépéseit? Szinkronizáljuk a böngészők eseményeit? Egyáltalán milyen feladatok merülnek fel, amikre a tervezés legelején nem gondolna az ember? Nézzük sorban!

3.7.2. A backend és a frontend kapcsolata

Az első és legfontosabb kérdés, hogy hogyan fogja látni az egyik böngésző, hogy mi történik a másikban, pl. ha lép a játékos, hogyan jut át az információ? Ahogy egy pincér egy jó étteremben, a szerver (pincér) kiszolgál, a kliens pedig rendel. Tehát minden eseményt vagy kérést a klienst küld el a szervernek, ill. ha a szerverhez befut egy információ, a szerver nem tolakodik a böngészőhöz, hogy átadja az információt, sőt azt sem tudja, hogy a frontendnek szüksége van rá. Ezért az alkalmazásban minden kommunikációt a frontend kezdeményez.

3.7.3. Játék indítása

A játék elindításához el kell navigálnunk a *Játék indítása* oldalra. Belépéskor elindul egy időzített funkció, ami 5 másodpercenként ellenőrzi, hogy a játékosnak van -e már aktív játékmenete. Minden ilyen kéréskor a szerver ellenőrzi az adatbázisban, hogy van -e olyan aktív meccs, amelyikben a játékos szerepel. Ez a módszer két okból is előnyös lesz.

Az egyik, hogy előfordulhat, hogy a játékos akarva vagy akaratlanul (internet kimaradás, ügyetlen kattintás, stb.) kilép a játékból. Így, ha tovább szeretne játszani, visszatérhet a játékba, ha pedig nem, akkor is feladni a játékot, mint szó nélkül ott hagyni az ellenfelet, aki a lépésre vár. Természetesen a játéknak érzékelnie kell a tartós távol maradást is. Ebben az esetben a meccsnek vége lesz, és az ellenfél nyer, mintha feladás történt volna.

A másik előny, hogy ha egy játékos elfogad egy kihívást, a játéknak át kell navigálnia mindkét játékost a játéktérre. A kihívott részéről egyszerű a dolgunk, az elfogadáskor egy állapot és útvonal változtatással gyorsan megoldhatjuk, de a kihívót aszinkron módon kell bejuttatni az aktív meccsbe. Ez az időzített funkció ellátja ezt a feladatot is.

Tehát ha a felhasználónak már van aktív meccse, mindig átkerül a játéktérre, és ilyenkor az időzített funkció leáll, nem hívódik meg többet.

3.7.4. Inicializálás

A meccs elindulásakor a szerver létrehoz egy aktív játékmenet objektumot, amelyet kezdő adatokkal lát el, majd elmenti az adatbázisba. Bele kerülnek a játékosok, hogy ki kezd (véletlenszerűen eldönti, hogy ki az aktív játékos), a játék típusa, a beállított szabályok, ennek megfelelően generál egy pályát, és alapértelmezettre állítja a lépést (üres sztring), a kör sorszámát (1), és 0-ra a nyertest, (nyerés esetén a játékos sorszáma kerül bele).

Elfogadott kihívás esetén ezt az objektumot kapják meg a játékosok válaszként az időzített, érdeklődő funkcióra. Így egyszerre kapnak választ, hogy játékban vannak -e, és inicializálódik a játék.

3.7.5. Játék közben

Ebben a szakaszban a következő szinkronizációs problémákat kell megoldanunk lépés esetén: Backend oldalon:

- Ellenőrizni kell a felhasználót, van -e jogosultsága a megadott lépéshez.
- Ellenőrizni kell, hogy a
- Ellenőrizni kell, hogy a lépés érvényes -e.
- Ellenőrizni kell, hogy nyert -e a játékos.
- És a beérkezett információkat rögzíteni kell az adatbázisban.

rFontend oldalon:

- A várakozó játékos lépési lehetőségét le kell tiltani, a lépő játékosét pedig engedélyezni.
- Frissíteni kell az időtúllépés stopperét.
- Frissíteni kell a pálya állását, és megjeleníteni az utolsó lépést.
- Ha nyert valaki, akkor befejezni a meccset.

Mindezek alapja ismét egy időzítő funkció, ezúttal azt ellenőrizzük másodpercenként, hogy lépett -e már az ellenfél. Ezt úgy tudja megmondani a szerver, hogy megnézi az adatbázisban, hogy az elmentett meccs körének száma nagyobb -e már, mint amikor ő lépett, vagy hogy lejárt-e a megengedett idő az ellenfél lépésére. Ha az ellenfél még nem lépett, újra meghívódik, ha lépett, vagy az ideje lejárt, az időzítő leáll, és a visszakapott adatokból kinyerjük a meccs frissítéséhez szükséges információkat. Ezek alapján el tudja végezni a fent említett funkciókat.

3.7.6. Játék vége

Egy meccset több féle képpen be lehet fejezni, történhet feladás, nyerés, vagy időtúllépés. A különbség, hogy mi váltja ki. Feladáskor a játékos a feladás gombra kattint, és egy kérést küld a szervernek a játék befejezésére. Ebben az esetben a játékos veszít. Nyerés esetén a szerver maga állapítja meg a játék befejezését.

Amiben mind egyezik, az az, ahogy a szerver lekezeli. Az aktív meccset szokás szerint frissíti, de ezúttal beállítja a nyertest is. Ezután létre hozza vagy frissíti a megfelelő adatokat az adatbázis lejátszott meccseket tartalmazó táblájában. Itt minden sor egy-egy felhasználó egy napi eredményeit tartalmazza egy adott játék típusban. Ezen tábla alapját állítja össze a szerver a statisztikai adatokat.

Mivel a játék indítás oldal úgy van megtervezve, hogy átvigye a játékost a játéktérre, ha aktív meccse van, így meccs végeztével nem árt kitörölni az aktív meccset az

adatbázisból, hogy újat tudjon később kezdeni. De mikor és hogyan lehet ezt megvalósítani? Az adatforgalom a frontend és a szerver között aszinkron folyamat, és mindkét játékosnál meg kell jeleníteni az eredményt, addig bent kell tartani az adatbázisban. De honnan tudja a szerver, hogy mikor törölheti? Bevezethetünk egy mezőt az adatbázisba, de plusz mezők tárolása helyett inkább megváltoztattam a játék indítás oldal aktív meccs ellenőrzését. Ezután ha talál olyan játékot, amelyben a játékos szerepel, megnézi azt is, hogy van -e már nyertes a játékban. Ha van, törli az aktív meccset, és a felhasználót a *Játék indítása* oldalon hagyja, hogy új játékot kezdhessen.

3.8. Authentikáció

3.8.1. Token használat

Mi lenne, ha nem lenne munkamenetünk, vagy ismertebb nevén, sessionünk?

A weboldalak a HTTP alapjain nyugszanak. Egy kérés, egy válasz. Ha egy erőforrást el akarunk érni, intézünk egy kérést a szerverhez, amire válaszként megkapunk egy HTML dokumentumot. Vagy egy képet. Vagy egy videót. A probléma ott van, hogy a szerver nem tudja ezeket a lekérdezéseket egymáshoz kapcsolni. Nem tudja azt, hogy aki az előző lekérdezésben a helyes jelszót küldte, az ugyanaz az ember, aki most a szupertitkos fájlokhoz hozzá szeretne férni.

Ezt a problémát oldja meg a session. Az első válasszal küldünk a böngészőnek egy sütit, benne egy azonosítóval, amit az innentől minden újabb kéréssel visszaküld. Szerveroldalon ehhez az azonosítóhoz rendeljük azokat az adatokat, amik ahhoz a sessionhöz, ahhoz a munkamenethez tartoznak.

Ez elméletben szép és jó, azonban egy igen csúnya probléma van vele. A programozó. Kényelmes módszer, hogy bejelentkezéskor szépen betöltjük az egész user objektumot, majd eltároljuk a sessionben. És ha már ott tartunk, az összes megrendelését is. És a session fájl csak nő, és nő, és nő. Amellett, hogy rengeteg helyet foglal a diszken, komoly logikai problémákat is okozhat a sessionök ilyen jellegű használata. Hiszen mi történik, ha a felhasználó egy másik eszközről is bejelentkezik, teszem azt a telefonjáról, és onnan módosítja az adatait? Vagy mi történik akkor, ha egy felhasználó párhuzamosan két lekérdezést indít?

mindkét folyamat betölti a session adatokat, majd mindkettő elkezd dolgozni a saját feladatán. A feldolgozás végeztével mindkettő visszaírja a sessiont a fájlba vagy adatbázisba. Vegyük észre azonban, hogy a második lekérdezés felülírja az első által végzett módosításokat. Vagy a munkamenet-kezelő ezt elkerülendő zárolja az adatokat, és egyszerre csak egy folyamatnak biztosít hozzáférést a sessionhöz, ami nem skálázódik túl jól.

Nézzünk tehát alternatívát a bejelentkezésre! Amikor a felhasználó bejelentkezik felhasználónévvel és jelszóval, kiadunk egy egyedi azonosítót, egy tokent. Munkamenet-azonosító helyett ezt tároljuk el a sütiben. Amikor a felhasználó egy olyan oldalra téved, ahol a felhasználói adataira van szükség, a süti kiolvasásra kerül, és a token alapján betöltjük a felhasználó adatait az adatbázisból.

Vagyis minden lekérdezésre kénytelenek vagyunk az auth tokent ellenőrizni. Elsőre azt sejtenénk, hogy ez nagyon nem hatékony, de ha jobban belegondolunk, a sessionöket ugyanúgy be kellett tölteni. Annyi a különbség, hogy most nem egy hatalmas amorf adathalmazt próbálunk kiszedni az adatbázisból, hanem egy nagyon is konkrét céllal

nyúlunk hozzá. Az eredmény az, hogy ehhez a célhoz tudunk megfelelő adatbázist választani és optimalizálni.

Sőt mi több, még egy nagyon fontos lehetőség nyílik meg előttünk. Mivel az auth tokeneket a felhasználóhoz kötötten tároljuk, lehetőségünk nyílik a felhasználó aktív tokenjeit kilistázni, hasonlóan mint ahogy a Facebook is csinálja.

Ezen felül használhatjuk még űrlapok kezelésére és állapot megőrzésére, de ez még kevéssé támogatott [9].

4. fejezet

Implementáció

4.1. Felhasznált technológiák

4.1.1. MySQL

A MySql az egyik legnépszerűbb nyílt forráskódú adatbázis kezelő rendszer, amelyet az Oracle Corporation fejlesztett és támogat. Relációs adatbázisokkal dolgozik, tehát egy nagy adathalmaz helyett az adatokat táblákban tárolja az adatokkal, a táblázat szerkezetekkel, a táblák kapcsolataival, és referencia kulcsaival együtt. Gyors, megbízható, skálázható, és könnyen tanulható [8].

4.1.2. Java EE

A Java a Sun által kifejlesztett programozási nyelv és futtatóplatform. A Java nyelv alapjaiban a C++-ra hasonlít, de azzal szemben magas szintű fejlesztést tesz lehetővé. Natív módon támogatja a többszálú programozást és objektum-orientált megoldásai is jóval túlmutatnak a korábbi nyelvekén.

A Java futtatóplatform alapját a Java bájtkód értelmezésére képes JVM képezi, amely gyakorlatilag minden elterjedt operációs rendszerre elérhető [7].

A Java Platform, Enterprise Edition, röviden Java EE, több programkönyvtárat tartalmaz és támogatja a többrétegű, elosztott alkalmazások készítését.

4.1.3. Spring

A Spring Framework egy moduláris Java keretrendszer, amit azzal a szándékkal alkottak meg, hogy a Java EE alkalmazások programozását könnyebbé tegyék. Néhány előnye:

- Könnyű súlyú, kevéssé invazív fejlesztés a régi jó Java objektumokkal (POJO - plain old Java object).
- Egyszerűbb teszt írás és kód újra felhasználhatóság.
- Loose coupling. Laza kapcsolódást jelent, ami a komponensek függetlenségére utal.
- Függőségek egyszerűbb hozzáadása a komponensekhez. (Dependency Injection).

- Ismétlődő, redundáns kódrészek (boilerplate kódok) redukálása.
- Tranzakció kezelés segítése [12].

4.1.4. Hibernate

A Hibernate egy Spring által támogatott ORM technikát megvalósító program könyvtár. Az adatbázisból való lekérésekhez, és az adatbázisban való módosításhoz az objektumainkat tábla oszlopokká kell alakítanunk, lekéréskor pedig fordítva, tábla oszlopainkat kell objektummá alakítani. Ezt hívják ORM-nek (Object Relation Mapping). A Hibernate ezen felül összeállítja SQL parancsainkat, lekérdezéseinket, és kezeli a kapcsolatkiépítést az adatbázissal. Jelentősen csökkenti az adatkezeléshez szükséges kód mennyiségét [12].

4.1.5. Maven

Az Apache Maven (röviden csak Maven) egy projekt kezelő és értelmező eszkösz, amely a project object model (POM) koncepciójára épül. A Maven irányítja a projekt buildelését, fordítását megfelelő formátumra, a függőségek importálását, tehát a projekt felépítését futtatható alkalmazássá [6].

4.1.6. TomCat

Az Apache Tomcat szoftver egy alkalmazás szerver, amely a Java Servlet, JavaServer Pages, Java Expression Language és Java WebSocket egy nyílt forráskódú implementációja [2].

4.1.7. HTML, CSS, JavaScript

Minden mai weboldal a HTML, a CSS, és a JavaScript. A HTML az oldal szerkezetét írja le, a CSS a megjelenését, a JavaScript pedig a viselkedést.

4.1.8. AngularJs

Az AngularJS egy Google által fejlesztett, nyílt forráskódú JavaScript keretrendszer dinamikus webes alkalmazásokhoz. Segítségével nagyban egyszerűsödik a webes alkalmazások frontend fejlesztése és az alkalmazások komponensei egyértelműen elkülönülnek, és rengeteg felesleges boilerplate kód elhagyható [1].

4.1.9. Git

A git egy ingyenes és nyílt forráskódú, elosztott verzió kezelő rendszer, amit arra terveztek, hogy a legkisebbtől a legnagyobb projektekig mindent gyorsan és hatékonyan kezeljen [3].

4.2. JSON Web Token

Korábban volt szó a token alapú azonosítás előnyeiről. Alkalmazásomban erre a célra JSON Web Tokent használtam (röviden JWT), amely egy nyílt szabvány, ami egy kompakt, önleíró, és önvalidáló eszköz, amellyel biztonságosan, JSON objektumként lehet adatot továbbítani két fél között. Megbízható módszer, mivel digitálisan "alá van írva".

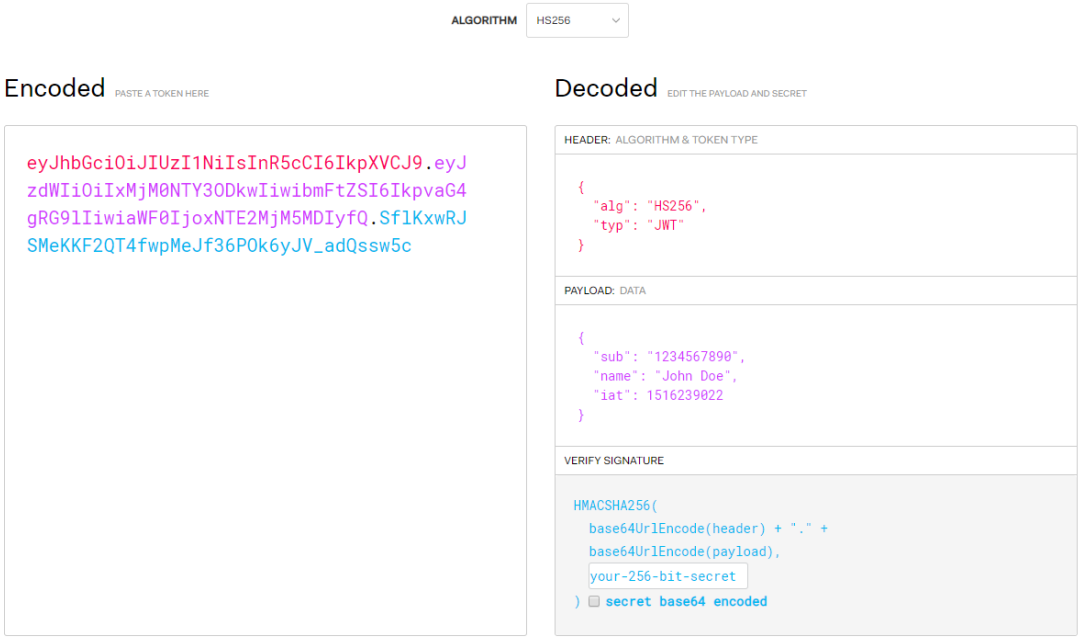
A JSON Web Tokenek 3 fő részből állnak, melyeket ponttal választanak el egymástól: Header (fejléc), Payload (rakomány), Signature (aláírás).

A header tipikusan két információt tartalmaz: a token típusát, ami JWT, és az alkalmazott titkosító algoritmust. A payload második rész, ami egy entitásról (jellemzően a felhasználóról) tartalmaz információkat. Ebben a részben megadhatunk lejárati időt is. Signature: Hogy ezt létrehozzuk, először szükségünk lesz a Base64Url kódolású headerre, szintén így kódolt payloadra, egy secretre (egy karakterlánc, amit csak a kibocsátó ismer). Majd ezeket titkosítjuk: az algoritmus egyik paramétereként megadjuk a headert és a payloadot ponttal elválasztva, másik paramétereként a secretet.

A signature-nek fontos szerepe van, ugyanis visszafejtéskor ennek a segítségével megállapíthatjuk, hogy a tartalmat valóban nem módosították, és hogy a küldő valóban az, akinek mondja magát. Ha visszafejtük a teljes tokent, megkapjuk a header.payload kódot és a signature-t. Ha visszafejtjük a signature-t is, szintén megkapjuk a header.payload kódot. Ha a két kód megegyezik, az üzenet érintetlen, hiszen a tokent már egyetlen elírt karakter is érvényteleníthet.

A tokenből továbbá visszakaphatjuk a payloadban található adatokat is, innen kiolvashatjuk, hogy ki a felhasználó, aki küldte.

Az 4.1 ábra jobb oldalán láthatjuk egy token tartalmát részekre bontva, bal oldat pedig az ezekből az adatokból generált tokent.



4.1. ábra. *Egy JSON Web Token felépítése*

A token fő szerepe az alkalmazásban az azonosítás, és ezáltal a jogosultságok ellenőrzése. A tokent a felhasználó bejelentkezéskor kapja meg, majd minden kérés során

elküldi a headerben. Így visszafejtés után a szerver azonosítani tudja a felhasználót, és ellenőrizni tudja a jogosultságát. Ezekre az ellenőrzésekre szükség lehet például, amikor a felhasználó az alkalmazás egy másik oldalára lép, vagy ha a játék során beérkezik egy lépés. A JavaScript futása manipulálható, ezért a szervernek fokozottan kell ellenőrizni a küldő kilétét.

4.3. Backend

Ebben a fejezetben az alkalmazásból kiemelt kódrészletek segítségével szeretném bemutatni, hogy hogyan is néz ki egy-egy szerver oldali komponens. A rétegeket fentől lefelé haladva mutatom be.

Fontos kiemelni, hogy az ilyen Javas, Springes alkalmazásokban jellemzően először minden osztályhoz írnak egy Java interfész osztályt, amelyből az implementációkat leszármaztatják. Ez azért jó, mert jelentősen megkönnyíti a komponensek cserélhetőségét. A cserének csupán annyi a feltétele, hogy az új komponens implementálja az interfészt, (ez biztosítja, hogy az új komponens át tudja venni a régi helyét), és regisztrálva legyen a rendszerben (annotációk vagy XML konfiguráció segítségével). Továbbá az annotációk és az interfészeknek köszönhetően nincs szükséges nekünk megadnunk példányosításkor a dinamikus típust, hanem a rendszer maga felismeri az implementáló osztályt. Ennek köszönhetően sokkal kevesebb helyen kell beleírni az alkalmazásba csere esetén, ami jelentősen csökkenti a hibák előfordulásának számát.

A kódrészleteken keresztül végig követhetjük, hogyan kommunikálnak egymással a rétegek, hogyan ellenőrzi a szerver, hogy a felhasználó, akinek az azonosítóját paraméterként megkapta, jogosult -e új játék kezdésére.

4.3.1. Kontroller réteg

A kontroller réteg feladata a kommunikáció, tehát fogadja a kérést, szükség esetén ellenőrzi a paramétereket és a kérés küldőjét, majd átadja a szolgáltatás rétegnek feldolgozásra. A visszakapott értéket visszaküldi a feladónak válaszként, 4.1 ábra.

```
package hu.lev.onlinegames.controller;

import org.springframework.beans.factory.annotation.Autowired
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.ResponseBody;
import org.springframework.web.bind.annotation.RestController;

import hu.lev.onlinegames.model.MatchActive;
import hu.lev.onlinegames.service.MatchService;

@RestController
public class MatchController {

        @Autowired
        private MatchService matchService;
```

```
    // CHECK START
        @RequestMapping(value = "/match/start/{userId}",
    method = RequestMethod.GET)
        @ResponseBody
        public MatchActive checkStart(@PathVariable int userId
    ) {
                MatchActive match = matchService.checkStart(
    userId);
                return match;
        }

}
```

Listing 4.1. *Egy kontroller osztály szerkezete*

4.3.2. Szolgáltatás réteg

A szolgáltatás réteg feldolgozza és összeállítja az adatokat. A szükséges adatbázis műveletekhez meghívja az adatbázis hozzáférési réteg metódusait, 4.2 ábra.

```
package hu.lev.onlinegames.service;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import hu.lev.onlinegames.model.MatchActive;
import hu.lev.onlinegames.persist.MatchDao;

@Service
public class MatchServiceImpl implements MatchService {

        @Autowired
        private MatchDao matchDao;

        public MatchServiceImpl() {
                super();
        }

        @Override
        public MatchActive checkStart(int userId) {
                MatchActive match = null;
                int matchId = matchDao.getMatchActiveId(userId
    );
                if(matchDao.isWinner(matchId)){
                        matchDao.deleteMatchActive(matchId);
                } else {
```

```
                        match = matchDao.getMatchActive(
    matchId);
                }

                return match;
        }

}
```

Listing 4.2. *Egy szolgáltatás osztály szerkezete*

4.3.3. Adat hozzáférési réteg

Az adat hozzáférési réteg feladata a kapcsolat felvétele az adatbázissal, lekérések, tartalom módosítása, adatkonverziók elvégzése, és az eredmény visszaküldése a szolgáltatás rétegnek, 4.3 ábra.

```
package hu.lev.onlinegames.persist;

import org.hibernate.Query;
import org.hibernate.Session;
import org.hibernate.Transaction;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Repository;

import hu.lev.onlinegames.manager.SessionManager;
import hu.lev.onlinegames.model.MatchActive;
import hu.lev.onlinegames.model.Players;
import hu.lev.onlinegames.model.User;

@Repository
public class MatchDaoImpl implements MatchDao {

        @Autowired
        SessionManager sm;


        public MatchDaoImpl() {
                super();
        }

        @Override
        public MatchActive getMatchActive(int matchId) {

                MatchActive match = null;

                Transaction tx = null;
                try {
                        Session session = sm.getNewSession();
```

```
                        tx = session.beginTransaction();

                        match = session.get(MatchActive.class,
 matchId);

                        tx.commit();
                         session.close();

                } catch (Exception e) {
                        match = null;
                        e.printStackTrace();
                        tx.rollback();
                }
                return match;
        }

        @Override
        public int getMatchActiveId(int userId) {

                int matchId = 0;

                try {
                        Session session = sm.getNewSession();
                        Transaction tx = session.
beginTransaction();

                        Query q = session.createSQLQuery("
select * from match_players where player1_fk = :a or
player2_fk = :a");
                        q.setParameter("a", userId);
                        Object[] result = (Object[]) q.
uniqueResult();

                        if(result != null) {
                                matchId = (int) result[3];
                        }

                        tx.commit();
                         session.close();

                } catch (Exception e) {
                        e.printStackTrace();
                }
                return matchId;
        }

        @Override
        public boolean checkAction(int matchId, int turn) {
```

```
                boolean isAction = false;

                try {
                        Session session = sm.getNewSession();
                        Transaction tx = session.
beginTransaction();

                        Query q = session.createSQLQuery("
select * from match_active where id = :a and (turn > :b ||
win > 0)");
                        q.setParameter("a", matchId);
                        q.setParameter("b", turn);
                        Object[] result = (Object[]) q.
uniqueResult();

                        if(result != null) {
                                isAction = true;
                        }

                        tx.commit();
                         session.close();

                } catch (Exception e) {
                        isAction = false;
                        e.printStackTrace();
                }

                return isAction;
        }

        @Override
        public boolean deleteMatchActive(int id) {
                boolean success = false;

                try {
                        Session session = sm.getNewSession();
                        Transaction tx = session.
beginTransaction();

                        Players players = new Players();
                        Query q = session.createSQLQuery("
delete from match_players where match_fk = :a");
                        q.setParameter("a", id);
                        q.executeUpdate();

                        MatchActive match = new MatchActive();
                        match.setId(id);
```

```
                        session.remove(match);

                        tx.commit();
                         session.close();

                        success = true;
                } catch (Exception e) {
                        e.printStackTrace();
                }
                return success;
        }

        @Override
        public boolean isWinner(int matchId) {
                boolean isWinner = false;
                try {
                        Session session = sm.getNewSession();
                        Transaction tx = session.
    beginTransaction();

                        Query q = session.createSQLQuery("
    select * from match_active where id = :a and win > 0");
                        q.setParameter("a", matchId);
                        Object[] result = (Object[]) q.
    uniqueResult();

                        if(result != null) {
                                isWinner = true;
                        }
                        tx.commit();
                        session.close();

                } catch (Exception e) {
                        isWinner = false;
                        e.printStackTrace();
                }

                return isWinner;
        }
}
```

Listing 4.3. *Egy adat hozzáférési osztály szerkezete*

4.3.4. Nyerés ellenőrzés

A 4.4. ábrán az amőba nyerés ellenőrzésének implementációját láthatjuk.

```
@Override
```

```
    public boolean checkWin(FiveInARowField[][][] fields,
int player, FiveInARowAction action) {

            boolean win = false;

            int startX = action.getX() - 4;
        // init min and max indexes, so we check fields in
board
            int startY = action.getY() - 4;
            int endX = action.getX() + 4;
            int endY = action.getY() + 4;

            // diagonals first, there is a bigger chance
to win this way, save some energy
            // diagonal from top-left
            int lengthSoFar = 0;
            for(int i = startX, j = startY; i<=endX && j<=
endY; i++){
                    if (i >= 0 && i < fields.length && j
>= 0 && j < fields[0].length){
                            if (fields[i][j].getValue() ==
 player){
                                    lengthSoFar++;
                            } else if (lengthSoFar < 5){
                                    lengthSoFar = 0;
                            }
                    }
                    j++;
            }
            if (lengthSoFar >= 5){ win = true; }

            // diagolal from bottom-left
            lengthSoFar = 0;
            for(int i = startX, j = endY; i<=endX && j>=
startY; i++){
                    if (i >= 0 && i < fields.length && j
>= 0 && j < fields[0].length){
                            if (fields[i][j].getValue() ==
 player){
                                    lengthSoFar++;
                            } else if (lengthSoFar < 5){
                                    lengthSoFar = 0;
                            }
                    }
                    j--;
            }
            if (lengthSoFar >= 5){ win = true; }
```

```
                // vertical
                lengthSoFar = 0;
                for(int j=startY; j<=endY ; j++){
                        if (j >= 0 && j < fields[0].length){
                                if (fields[action.getX()][j].
    getValue() == player){
                                        lengthSoFar++;
                                } else if (lengthSoFar < 5){
                                        lengthSoFar = 0;
                                }
                        }
                }
                if (lengthSoFar >= 5){ win = true; }

                // horizontal
                lengthSoFar = 0;
                for(int i=startX; i<=endX ; i++){
                        if (i >= 0 && i < fields.length){
                                if (fields[i][action.getY()].
    getValue() == player){
                                        lengthSoFar++;
                                } else if (lengthSoFar < 5){
                                        lengthSoFar = 0;
                                }
                        }
                }
                if (lengthSoFar >= 5){ win = true; }

                // no winning found
                return win;
        }
```

Listing 4.4. *Az amőba győzelem ellenőrzésének implementálása*

4.4. Frontend

A frontendnek a jelentősebb részei a játék vezérlő komponensek. Sok lenne itt részletesen bemutatni mindet, ezért csak a játékmenet eseményeinek szinkronizációját biztosító függvényeket emelem ki. Ők végzik el a munka nagyját, 4.4 ábra.

```
// Add event listener for 'click' events.
elem.addEventListener('click', function(event) {
        if(!vm.disable){
                vm.disable = true;

                var x = event.pageX - elem.offsetLeft;  // get
    canvas x,          elem.offsetLeft - canvas origo x
```

```
                var y = event.pageY - elem.offsetTop;    // get
 canvas y          elem.offsetTop - canvas origo y
                var field = man.getClickedField(vm.board, x, y
);                                                       // get
clicked field by index

                if(field != null){
                        field.value = match.fields[field.x][
field.y].value;
        // get value of field
                        // console.log(field.x + "," + field.y
 + ' - ' + field.value);

                        if([0,4].includes(field.value)){
                                match.action = field;
                                var data = man.
getMatchReadyToSend(match);

                                $http.post(baseUrl + '/
fiveinarow/action', data)
                                .then(function(result){
                                        var field = match.
fields[match.action.x][match.action.y];

                                        if(match.action.value
== 4){
                                                man.
activateTrap(field, vm.board, ctx);
                                                field.value =
0;
                                        }

                                        if(match.action.value
== 0){
                                                man.
drawCharacter(field, vm.board, match.players.activePlayer,
ctx);
                                        }

                                        if(result.data){
                                                checkAction();
                                        } else {
                                                vm.disable =
false;
                                        }
                                });
                        }else {
                                vm.disable = false;
```

```
                        }

                }
        }
}, false);

function checkAction(){
        vm.promise = $interval(function(){
                // console.log(baseUrl + '/fiveinarow/
    checkaction');
                $http.get(baseUrl + '/fiveinarow/checkaction',
     {
                        params: {
                                matchId: match.id,
                                turn: match.turn + 1
                        }
                })
                .then(function(result){
                        if(result.data != null && result.data
    != ""){
                                vm.initMatch = result.data;
                                match = man.reload(vm.
    initMatch, match, vm.board, ctx);

                                if (match.players.activePlayer
     == 1) {
                                        vm.disable =
    $localStorage.currentUser.userid == match.players.player1.
    id ? false : true;
                                } else {
                                        vm.disable =
    $localStorage.currentUser.userid == match.players.player2.
    id ? false : true;
                                }

                                stopcheckAction();
                        }
                });
        }, 1000, false);
}

function stopcheckAction(){
        // console.log("checkAction OFF");
        $interval.cancel(vm.promise);
}

function initiateMatch(initMatch, match, board, ctx){
```

```
        if(typeof initMatch.options == "string"){
                // convert string to int array
                if(initMatch.options != null){
                        var op = initMatch.options;
                        op = op.substring(1, op.length-1);
                        op = op.split(',').map(function(item)
    {
                                return parseInt(item, 10);
                        });
                        initMatch.options = op;
                }
        }

        var tempFields = resetFields(board, 0);
        initMatch.fields = JSON.parse(initMatch.boardstate);

        for (var i = 0; i < initMatch.fields.length; i++) {
                for (var j = 0; j < initMatch.fields[i].length
    ; j++) {
                        tempFields[i][j].value = initMatch.
    fields[i][j].value;
                }
        }

        initMatch.fields = tempFields;
        match = initMatch;

        drawBoard(board, ctx);
        drawFields(match, board, ctx);

        return match;
}

function reload(initMatch, match, board, ctx){
        initMatch.fields = initMatch.boardstate == "" ? null :
     JSON.parse(initMatch.boardstate);
        initMatch.action = JSON.parse(initMatch.action);
        match = initMatch;
        // console.log("WIN: " + match.win);

        initMatch.fields.forEach(function(fieldsCol){
                fieldsCol.forEach(function(field){
                        if(field.value == -1){
                                field.value = 0;
                        }
                });
        });
```

```
        // convert string to int array
        if(match.options != null){
                var op = match.options;
                op = op.substring(1, op.length-1);
                op = op.split(',').map(function(item) {
                        return parseInt(item, 10);
                });
                match.options = op;
        }

        if(match.action){
                var field = match.fields[match.action.x][match
.action.y];

                switch (match.action.value) {
                        case 0:
                                var player = match.players.
activePlayer == 1 ? 2 : 1;
                                drawCharacter(field, board,
player, ctx);
                                break;
                        case 4:
                                activateTrap(field, board, ctx
);
                                field.value = 0;
                                break;
                        default:
                                break;
                }
        }

        if(match.win && match.win > 0){
                if(match.win == 1){
                        $localStorage.currentUser.userid ==
match.players.player1.id ? winMsg = "Gratulalok, nyertel!"
: winMsg = "Sajnos vesztettel!";
                } else {
                        $localStorage.currentUser.userid ==
match.players.player2.id ? winMsg = "Gratulalok, nyertel!"
: winMsg = "Sajnos vesztettel!";
                }
                alert(winMsg);
                $state.go('welcome');
        }

        return match;
}
```

Listing 4.5. *Az amőba fontosabb függvényei*

5. fejezet

Tesztek, eredmények

5.1. Teszt alapú fejlesztés

Egy alkalmazás elkészítésekor fontos szempont a minőségbiztosítás. Erre manapság rengeteg fejlesztési modellt kitaláltak, melyeknek szerves része a tesztelés. Legtöbb esetben a fejlesztés sorrendje több-kevesebb eltéréssel: tervezés, fejlesztés, tesztelés. Ennek egyik nagy hátránya, hogy ebben a szakaszban már sokkal nehezebb javítani egy hibát, mint az elején, hiszen ekkorra már sokkal nagyobb lehet a program, nehezebb átlátni, és a függőségek miatt több helyen kell változtatni a kódon, nem is beszélve a plusz költségekről.

Rendkívül fontos tehát a tesztelés minél korábbi elkezdése, sőt, a teszt alapú tervezés. Ez úgy történik, hogy megállapítjuk a feature-öket, vagyis az elvárt funkciókat. Ezek általában már önmagukban is túl nagyok, hogy egyben írják meg, így kisebb részegységekre kell bontani. Ezekhez az egységekhez pontosan megfogalmazott, elfogadhatósági kritériumokat kell állítani. Ezen kritériumok alapján pedig meghatározatók, sőt, el is készíthetők a tesztek is már a tervezés legelején.

Amikor tesztekről beszélünk, főleg a tervezés elején, elsősorban automatizált tesztekről beszélünk. Ennek több előnye is van. Lássunk is néhányat a tervezés korai szakaszában megírt automatizált tesztek előnyei közül a teljesség igénye nélkül:

- A tesztek kivitelezéséhez az elkészítés után nincs szükség emberi erőforrásra.
- Kevesebb hiba lehetőség a tesztelés során.
- Korán felismerhető, így gyorsabban, olcsóbban javítható hibák.
- A tesztek minden kódváltoztatás után, fordításkor automatikusan lefutnak.
- Információt szolgáltathatnak a feature-ök, komponensek komplexitásáról, a fejlesztéshez szükséges időről és egyéb erőforrásokról. Azokat a funkciókat, amelyekre nehéz tesztet írni, jó eséllyel lefejleszteni is bonyolult lesz.

Ezt a tesztelési módszertant acceptance-test-driven developmentnek (ATDD), azaz magyarítva kb. átvételi vizsgálatokon alapuló fejlesztésnek hívják. [10]

5.2. Tesztelés az alkalmazásban

Habár az alkalmazásomba nem kerültek be automatizált tesztek, fontos szempont volt, hogy egy-egy nagyobb egység is kisebb, önmagukban is tesztelhető egységekből épüljön

fel. Ezt az elvet főként az amőba frontendes direktíva kódjában tekinthetjük meg, ahol a működést kisebb egységekre, függvényekre bontottam, és az alkalmazás többi részében is ez a követendő példa. A függvények csak egy-egy kisebb feladatot látnak el, és pontosan meghatározható, hogy milyen bemenetre milyen kimenetet kell adniuk.

5.3. Eredmények

Az alapos tervezésnek, és implementálásnak köszönhetően elkészült egy bemutatható változat, amely tartalmaz egy teljes, működő adatbázist, a szervert, ami a frontenddel és az adatbázissal is kommunikál, és a frontend, ami szépen megjeleníti a tartalmakat a felhasználónak, tartalmaz autentikációt, és elvégzi a kommunikációt a szerverrel. A játékok közül az amőba került implementálásra, amelyhez két extra szabályváltozat készült, amelyekből kiválaszthatunk akár egyet, akár kettőt.

A fejlesztés során a legnagyobb nehézségek forrása a konfiguráció, és a kommunikáció helyes működésének megteremtése volt, vagyis az alkalmazás alapjainak elkészítése. Konfigurálni kellett az alap szerver alkalmazást, annotálni a komponenseket, beállítani a dispatcher szervletet, amely a kéréseket a megfelelő kontrollerhez irányítja, hozzá kellett adni az adatbázist, meg kellett alapozni a frontendet is. Kezdetben úgy indult, hogy a backen és a frontend külön projektben szerepeljen, külön futó alkalmazásokként érjék el egymást. Ebben az állapotban szükség volt még egy egyszerű külön szervert biztosítani a frontendnek is. Később a backend és a frontend összevonásra került, így a frontendes kis szerverre már nem volt szükség.

Egy másik, ami valamelyest kötődik a konfigurációhoz is, az adatkonverzió. Az alkalmazás rengeteg adatkonverziót tartalmaz. Egy kérés előtt az adatokat ellenőrizni kell, és megfelelő formába kell rendeznünk. A frontend az adatokat JSON objektumként küldi el. Amikor a szerver ezt megkapja, egy kezdetleges Java objektummá kell alakítania, hogy kinyerhesse belőle az információkat. A kinyert adatok alapján a szerver összegyűjti a műveletek elvégzéséhez szükséges objektumokat, és elvégzi a kívánt műveleteket. Ezek során több adatbázis műveletre is szükség lehet, melyek során az objektumot a Java és az ER modellnek megfelelő formátumoknak megfelelően kell konvertálni. A Java objektumok az ER modellel ellentétben nem egyszerű típusú azonosítót használnak, hanem a hivatkozó táblát reprezentáló objektum típusát, amelyeket nagyon körültekintően kellett annotálni, konfigurálni, különben nem működtek. Továbbá az idegen kulcsoknak nem csak a hivatkozó, hanem a hivatkozott osztályban is szerepelni kellett. Mivel ezek egymásra mutatnak, így könnyen előfordulhat, hogy egy végtelen hivatkozás sorozatot indítunk, amely hibás működést eredményez. Ezek miatt gondoskodni kellett róla, hogy a válasz (JSON) objektumba csak a megfelelő adattagok kerüljenek be, csak az egyik objektum hivatkozzon a másikra.

Az adatkonverziók, adatformátumok meghatározását nagyban segítette és gyorsította a tervezés fázisban előre meghatározott REST API.

6. fejezet

Összefoglalás

A dolgozat és a program megírásához sok féle tudásra volt szükség, rengeteg tapasztalatot és gyakorlati készséget adott. Meg kellett ismerni a technológiákat, melyeknek egy részét kezdő szinten ismertem, némely részével pedig korábban egyáltalán nem találkoztam. Szükség volt a jó tervezés készségének elsajátítására is, mert bár elsőnek nem tűnik bonyolultnak az alkalmazás, időnként meglepően komplex problémákat kellett megoldani. A legbonyolultabb részek a konfiguráció, az adatkonverziók, és a folyamatok szinkronizációja volt, hiszen minden kérést, folyamatot számos rétegen kellett végig vinni.

A dolgozat megírása során sikerült működőképest programot tervezni, és ez alapján egy működő képes demót összerakni. A dolgozat során tervezett funkciók közül ugyan nem készült el mind. Ezek nem problémaként jelennek meg, hanem tervezetten egy későbbi fejlesztési fázis részei lesznek majd. Az alkalmazás tartalmaz regisztrációt, bejelentkezést, a bejelentkezésnek megfelelően változó menüsort. A korábban tervezett játékok közül az amőba készült el kettő pluszban választható játékszabály módosítással. Elkészültek a rangsorok is, így nyomon követhető, hogy kik a legjobb játékosok, és a felhasználók képet kaphatnak saját teljesítményükről is.

Adathordozó használati útmutató

A szakdolgozatomhoz mellékelt adathordozó eszközön található adatok struktúrája:

- online-games.zip - Az elkészült alkalmazás tömörített formátumban
- dolgozat.zip - A dolgozat forráskódja tömörített formátumban
- dolgozat.pdf - A dolgozat .pdf formátumban
- manual.txt - telepítési útmutató egyszerű szöveges formátumban

6.1. Telepítési útmutató

Csomagoljuk ki az online-games.zip fájlt a kívánt helyre!

Töltsünk le az Apache TomCat oldaláról a TomCat alkalmazás szerver 9.0.4 verzióját!

Csomagoljuk ki a letöltött fájt, és másoljuk a kívánt helyre! Végezzük el az alábbi módosításokat a TomCat conf könyvtárában: Adjuk a tomcat-users.xml fájl végét jelző tag elég az alábbi szöveget: <role rolename="manager-gui"/> <user username="admin" password="password" roles="manager-gui"/> A server.xml fájlban módosítsuk a Connector tagben található port paramétert 9000-re. Navigáljunk a TomCat alkalmazás szerver bin mappájába, majd indítsuk el a startup.bat fájlt, (linux esetén startup.sh).

Töltsük le a MySQL holnapjáról a Workbench alkalmazás 8.0 CE verzióját, és telepítsük!

Hozzunk létre egy új kapcsolatot (connection) az alábbi adatokkal:

- Connection Name: online_games
- Hostname: 127.0.0.1
- Port: 3306
- Username: og_manager
- Password: onlinegames
- Connection Method: Standard (TCP/IP)

Csatlakozzunk a létrehozott Connectionhöz.

Futtassuk le a játék mappájában az sql mappában található create-db.sql és fill-db.sql fájlokat.

Győződjünk meg róla, hogy fut az adatbázis szerver (Workbench, Server menü, Server status).

Ha nem fut a szerve, indítsuk el.

A játék mappájában (online-games) találunk egy deployer.cmd fájlt. Nyissuk meg egy szövegszerkesztőben, majd állítsuk be a könyvtárak elérési útvonalait a számítógépünknek megfelelően, majd mentsük el.

- server_dict: Az a mappa, ahol a játk mappájában taláható pom.xml fájl található.
- tomcat_dict: A TomCat alkalmazás szerver webapps mappája, ide kerül a fordítás során generált .war file.
- war_file: A keletkező .war file elérési útvonala, ami a játék (online-games) mappa target könyvátárában található.
- tomcat_file: A tomcat_dict változóban megadott mappában található .war file elérési útvonala, (ha létezik).

Nyissunk egy konzolt, majd navigáljunk a játék mappájába, majd futtassuk le a deployer.cmd fájlt.

Ekkor a program lefordul, és a keletkezett .war fájlt bemásolja a TomCat alkalmazás szerver webapp mappájába. Ezután a TomCat automatikusan feldolgozza a fájt, amelyet nyomon követhetünk a futó startup program konzoljában. Amint a TomCat végzett vele, a játék használatra kész.

Ezek után a játék az alábbi címen válik elérhetővé:
http://localhost:9000/online-games/WebContent/index.html#!/login

Minden használatkor győződjünk meg, hogy fut az adatbázis szerver és a TomCat szerver!

Irodalomjegyzék

[1] Angularjs — superheroic javascript mvw framework. https://angularjs.org/. [online].

[2] Apache tomcat - welcome! http://tomcat.apache.org/. [online].

[3] Git. https://git-scm.com/. [online].

[4] Kisokos - a dámajáték szabályai. http://mek.oszk.hu/00000/00056/html/133.htm. [online].

[5] Kisokos - az amőbajátékok szabályai. http://mek.niif.hu/00000/00056/html/132.htm. [online].

[6] Maven – welcome to apache maven. https://maven.apache.org/. [online].

[7] Mi a java ? - szótár - pc fórum - pc fórum. https://pcforum.hu/szotar/Java. [online].

[8] Mysql :: Mysql 8.0 reference manual :: 1.3.1 what is mysql? https://dev.mysql.com/doc/refman/8.0/en/what-is-mysql.html. [online].

[9] Sessionmentes weboldalak · weblabor. http://weblabor.hu/cikkek/sessionmentes-weboldalak. [online].

[10] Tesztelni az első pillanattól. https://computerworld.hu/tech/tesztelni-az-elso-pillanattol-109370.html. [online].

[11] Antal Margit. *Java web technológiák*. Scientia Kiadó, 2010.

[12] Craig Walls. *Spring In Action third edition*. Manning Publications Co., 2011.