**Eötvös Loránd Tudományegyetem**

**Informatikai Kar**

**Média- és Oktatásinformatika Tanszék**

**MusicBoard**

**Témavezető:**

Daiki Tennó

mestertanár

**Szerző:**

Gyugyi Péter

Programtervező informatikus Bsc.

Budapest, 2020

**Témabejelentő**

A technológia fejlődése az utóbbi évtizedekben hatalmas változást hozott a zeneiparnak minden téren. Az egyik változás, ami talán leginkább vált a zeneszerzők előnyére, hogy már hangszer nélkül is tudunk zenét készíteni. Számtalan szoftver és plugin áll már rendelkezésünkre, amivel zenét programozhatunk. Egy számítógép elég már egy teljes zenei mű kidolgozására. Ezek a megoldások azonban kevésbé támogatják azokat az embereket, akik nem egyenként szeretnék a hangokat leprogramozni és visszahallani, hanem valós időben szeretnének, bármilyen extra eszköz nélkül zenélni. A szakdolgozatom erre nyújt megoldást. Ez a webes alkalmazás lehetővé teszi, hogy billentyűzeten is zenélhessünk, minőségi hanganyaggal, amit én készítek el. Mivel a zeneszerkesztő programok mind bonyolultak, kezelésük sok tanulást igényel, cél, hogy az applikáció egyszerű legyen, letisztult kezelőfelülettel. Az alkalmazás alapvetően már több hangszert fog kínálni, de természetesen hosszútávon rengetek egyéb hangszerrel és funkcióval lehet bővíteni. A nem regisztrált felhasználók számára az alapértelmezett billentyűkiosztással fog működni, de a regisztrált felhasználók el is menthetnek személyre szabott konfigurációkat, így nem kell minden alkalommal újra beállítani, elég lesz bejelentkezni és kiválasztani a kívánt konfigurációt. Azért lesz webes az alkalmazás, hogy mindenki számára könnyen hozzáférhető legyen. A backend Java, a UI pedig JavaScript alapú lesz.

**Tartalom**

[1. Bevezetés 5](#_Toc58582638)

[2. Felhasználói dokumentáció 7](#_Toc58582638)

[2.1 Rendszerkövetelmény, Elérés 7](#_Toc58582639)

[2.2 Grafikus felület 7](#_Toc58582640)

[2.2.1 Zongora 8](#_Toc58582641)

[2.2.2 Gitár 11](#_Toc58582642)

[2.2.3 Logger 13](#_Toc58582643)

[2.2.4 Felhasználó-kezelés 15](#_Toc58582644)

[3. Fejlesztői dokumentáció 19](#_Toc58582638)

[3.1 Feladatok megfogalmazása, Tervezés 19](#_Toc58582645)

[3.1.1 Alapanyagok, Kódolás előtti teendők 19](#_Toc58582646)

[3.1.2 Verziókezelés 19](#_Toc58582647)

[3.1.3 Backend 20](#_Toc58582648)

[3.1.4 Frontend 21](#_Toc58582649)

[3.1.5 Deployment 21](#_Toc58582650)

[3.2 Backend implementáció 21](#_Toc58582651)

[3.2.1 Projekt inicializálás, Dependencyk 22](#_Toc58582652)

[3.2.2 Szerkezet 22](#_Toc58582653)

[3.2.3 Adatbázis-kezelés, Service-ek 24](#_Toc58582654)

[3.2.4 Controller 26](#_Toc58582655)

[3.2.5 Security 26](#_Toc58582656)

[3.3 Frontend implementáció 27](#_Toc58582657)

[3.3.1 Projekt inicializálás, Dependencyk 27](#_Toc58582658)

[3.3.2 Szerkezet 27](#_Toc58582659)

[3.3.3 Autentikáció, API Context 28](#_Toc58582660)

[3.3.4 Navigálás 30](#_Toc58582661)

[3.3.5 Logger 30](#_Toc58582662)

[3.3.6 Song Handler 32](#_Toc58582663)

[3.3.7 Hangfájlok importálása 32](#_Toc58582664)

[3.3.8 Piano 32](#_Toc58582665)

[3.3.9 Guitar 35](#_Toc58582666)

[3.4 Tesztelés 38](#_Toc58582667)

[3.5 Deployment 44](#_Toc58582668)

[4. Konklúzió 45](#_Toc58582669)

[5. További fejlesztési lehetőségek 46](#_Toc58582670)

[6. Forrásjegyzék 47](#_Toc58582671)

# Bevezetés

A digitális technológia számtalan változást hozott minden művészeti ágban, így a zenében is. Ma már egyáltalán nem meglepő, hogy a modern zenék többsége csak kis mértékben, vagy akár egyáltalán nem tartalmaz valós hangszert. Minden lehetőség adott ahhoz, hogy az ember a szobájában, teljesen egyedül alkosson, akár egy komolyzenei mű komponálásról, akár egy pop szám programozásáról van szó. Ehhez első lépésként elengedhetetlen egy DAW (Digital Audio Workstation), mint például a Cubase, Reaper, FL Studio stb. Ez önmagában csak egy környezet a zeneszerkesztéshez, amit már eleve rendkívül bonyolult használni, elsajátítása sok időt vesz igénybe, de elengedhetetlen. Ebben a környezetben tudunk használni a plugineket, amikből ma már szinte végtelen létezik.

Minden adott tehát az otthoni zeneíráshoz/zeneszerkesztéshez. Viszont számolni kell azzal, hogy sok energiát kell fektetni egy DAW kiismerésébe, meg kell venni (persze csak miután van egy PC, ami el is bírja), és ráadásul minden egyes plugint külön-külön meg kell vásárolni (bár vannak ingyenesek is, de igen szűk lehetőségekkel, egy profi hangmérnök tipikusan rengeteg fizetett libraryt használ). És ez még magához a zenéléshez nem is elég, csak ahhoz, hogy minden hangot egyesével beprogramozzunk. Egy MIDI (Musical Instrument Digital Interface) szintetizátorra is szükség van a valós idejű zenéléshez.

Tehát egy minőségi környezet kialakításához nem kevés pénzt és energiát kell belefektetni hobbinkba, bár nyilván még mindig jóval egyszerűbb, mint minden hangszert megvenni és külön-külön felvenni.

Érdemes lenne egy lehetőséget biztosítani azok számára is, akiknek csak egy PC és internetelérés adott. Az én webapplikációm célja, hogy nagyon egyszerűen, jó hanganyaggal tudjunk zenélni. Célom, hogy ez ne csak arra legyen használható, hogy az amatőr zenészek lepötyögjék a Süss fel napot, hanem azoknak a zeneszerzőknek is hasznára váljon, akik nem férnek hozzá mindig a professzionális környezetükhöz. Én például néha hosszú utakon szívesen ötletelnék, de egy laptop önmagában nem ad erre lehetőséget. Ezzel az applikációval elég billentyűzeten gombokat nyomogatni ahhoz, hogy valós időben visszahalljuk a zenét, amit játszunk. Illetve, ha van egy konkrét ötlet, amit mindenképp szeretnénk rögzíteni is, hogy ne felejtsük el, mire legközelebb hangszerhez vagy DAWhoz jutunk, el is tudjuk menteni.

Persze nem én vagyok az első, aki gondolt erre. Viszont az ehhez hasonló, már létező webappok lehetőségeit szűkeknek találom, többségük nem ad lehetőséget az ötleteink elmentésére, a hangszerek teljes hangtartományának kihasználására, illetve ami számomra nagyon zavaró, hogy többnyire nagyon gépies MIDI hanganyagot használnak. Továbbá az sincs kihasználva, hogy egy webappról van szó, és nem egy letölthető, offline alkalmazásról. Ezeket a szempontokat figyelembe véve a MusicBoard célja, hogy kitűnjön ebből a kategóriából.

1. **Felhasználói dokumentáció**

Mint már említettem, fontos szempont, hogy a felhasználó egy egyszerű, nagyon letisztult felülettel találkozzon, amikor az oldalt megnyitja. Egyes funkciókat elláttam labelökkel, de a használat tulajdonképpen 1-2 percnyi ismerkedés után magától értetődő kell, hogy legyen. Természetesen a kezelőfelület angol nyelvű.

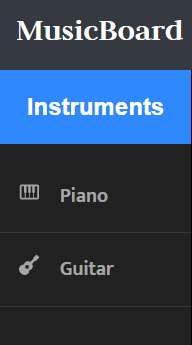
## Rendszerkövetelmény, Elérés

Mivel webes applikációról van szó, szimplán csak egy böngészőre (Google Chrome, Microsoft Edge vagy Safari ajánlott a tökéletes működéshez) és internetelérésre van szükség. Szintén magától értetődő, hogy valamilyen audio perifériára is szükség lesz (hangfal, fülhallgató, fejhallgató). Az alábbi linkről elérhető:

<https://music-board.netlify.app/>

Az oldal megnyitása után lehet, hogy kell várni pár másodpercet, amíg betöltenek az audio fileok. Az applikáció PCs felhasználásra készült, a megjelenítés az átlagos képernyőfelbontásokkal (1920x1080, laptopon 1280x720) megfelelő, ennél kisebb méretű kijelzőn már nem fog elférni minden elem. A hangszerek grafikus megjelenítése miatt mobilos használatra alkalmatlan (egyébként sem lenne egyszerű telefonos billentyűzettel használni).

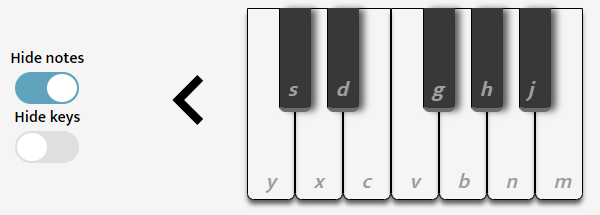
## Grafikus felület

****Ez egy single page app, egy fix headerel és sidebaral. Utóbbi ad lehetőséget a hangszerek közötti navigálásra, melyekből egyet mindig magunk előtt látunk. A header Login fülén van lehetősége a felhasználónak regisztrálni/bejelentkezni, erre egy későbbi fejezetben térek ki. A sidebar egyértelműen jelzi kis ikonokkal és a hangszerek neveivel, hogy hova fog irányítani. Az oldal már megnyitáskor az első hangszerre, a zongorára fog alapértelmezettként navigálni.

1. ábra Sidebar

### Zongora

Elsőként a zongora használatát és beállításait mutatom be. Először is egy sávot látunk sok gombbal, ez a field univerzális, minden hangszernél elérhető lesz, így erre majd később térek ki. Ez alatt látunk egy (illetve két) egyszerű, néhány gombbal személyre szabható zongorát. A bal- és jobb oldali zongora tulajdonképpen egy, csak a két kezünk különböző pozícióit reprezentálják.

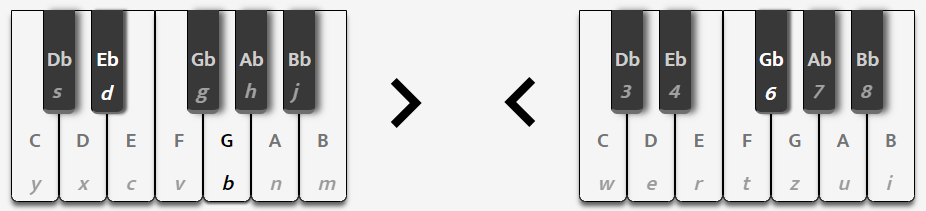
Alapértelmezetten segítségként a program a zongora billentyűire égeti, hogy milyen hangot szólaltatnak meg, illetve melyik gombra reagál a billentyűzeten. Balra található 2 switch, ami ezeket külön-külön kezelve engedi a felhasználónak, hogy elrejtse őket. Ez azonban elsősorban csak tapasztaltabb usereknek lehet hasznos, akik már jól ismerik a hangszer beállításait, és esetleg zavarja őket ez a kis segítség. Értelemszerűen a ’Hide notes’ switch a zongora-billentyűhöz tartozó hangjegyet (felső sor) rejti el, a ’Hide keys’ pedig a billentyűzeten lenyomandó gombot (alsó sor). Természetesen ez a beállítás mindkét ábrát manipulálja.

2. ábra A hangok el vannak rejtve

A zongora, mint minden hangszer, a billentyűzet segítségével szólaltatható meg (egér kattintásra az ábrán nem reagál). A billentyűkiosztást angol billentyűzet mintájára (ezért nincs például az ’í’ gomb kihasználva), a valós zongorához megfeleltethetően állítottam be. Egy ábra egy oktávot fed le (12 egymást követő hang). Mivel összezavaró lehet, hogy a billentyűzet és a zongora gombjait is billentyűknek nevezzük, illetve, hogy a zenei hangokat is az ábécé betűivel jelöljük, ezért amikor a zongora billentyűi által megszólaltatott hangokra hivatkozok, nagybetűket fogok használni (’C’), a billentyűzeten lenyomott gombra hivatkozva pedig kis betűt (’y’).

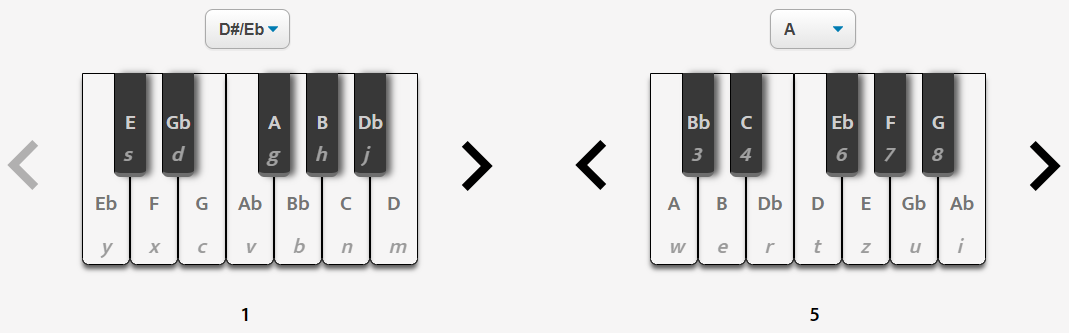
Kezdjük a bal oldali ábrával. Ahogy az app is jelzi, a billentyűzeten lévő legalsó betűsoron ’y’-tól ’m’-ig tartó intervallum szólaltatja meg a fehér billentyűket, a felette lévő betűsoron pedig az ’s, d, g, h, j’ gombok a fekete billentyűket. Ennek az az oka, hogy a valóságban is így játszanánk a hangszeren. A fehér billentyűkön játsszuk az egész hangot, a feketéken pedig a félhangokat. Mivel egy kromatikus skálán (ami minden hangot tartalmaz) a 12 hangból 7 egész-, és 5 félhang, így ki kell maradnia minden ismétlődő szakaszon kettő fekete billentyűnek, ilyenkor két fehér billentyű van egymás mellett. Ha egy valós zongorán egy ’C’ hangot megszólaltató billentyűre tekintünk, a következő ’C’ hangig azt fogjuk látni, mint most az ábrán. A ’C’ hang után következik a ’C’ és a ’D’ közötti félhang, a ’C#’ vagy más jelöléssel ’Db’, amit egy fekete billentyű szólaltat meg, ami az első két fehér között helyezkedik el. A feketék mindig rövidebbek, mint a fehérek, így a valóságban egy kicsit feljebb kell nyúlni az ujjainkkal, hogy elérjük őket. Éppen ezért van egy sor eltérés a billentyűzeten köztük, így reálisabb érzést ad játszani rajta. És ha megnézzük például az ’E’ és ’F’ hangokat, amik között nincs félhang (tehát nem létezik ’E#’ vagy ’Fb’), ott ki kell maradnia a fekete billentyűnek, ezért a billentyűzeten is kimarad az ’f’ gomb.

Természetesen ugyan így működik a másik kezünkre eső zongora is, azzal az eltéréssel, hogy ott a ’w’-től ’i’-ig tartó intervallumot használjuk a fehér, a ’3, 4, 6, 7, 8’ gombokat pedig a fehér billentyűkhöz.

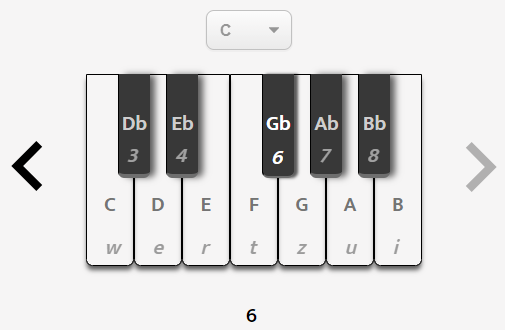
A leütött billentyűket minden esetben animáció jelzi, ha a segítség is be van kapcsolva, akkor a lenyomott gombok és hangok is kitűnnek.

3. ábra Animáció az Eb, G és Gb hangokon

Mindkét ábra jobb- és bal oldalán van egy nyíl. Ezeket megnyomva tudjuk a kívánt oktávot beállítani, azaz tulajdonképpen 12 hanggal „elcsúsztatni” a kezünket a nyíl irányába. Az aktuális oktáv számát az ábrák alatti szám jelzi. A zongora hanganyaga 6 oktávot fed le, ha valamelyik szélsőértékig „kinyújtjuk a kezünket”, akkor az adott nyilat nem nyomhatjuk meg többször. Bár a billentyűkiosztás fix, mindként billentyű-csoportot tudjuk a hanganyag teljes terjedelmében mozgatni, így nem kell ragaszkodnunk ahhoz, hogy a bal kezünk játssza a mélyebb hangokat, a jobb pedig a magasabbakat (mint általában a valóságban).

Végül az ábrák feletti selectek is a hangokat manipulálják. Ez a funkció kivételesen máshogy működik, mint a valóságban. Alapértelmezetten a zongora ’C3’ és ’C4’ hangjait veszi kezdőhangoknak (azaz az ’y’ és a ’w’ hangjai). A lenyíló lista felsorolja a létező hangokat, ezekből egyet kiválasztva pedig ahhoz igazítja a kezdőhangot, tehát tulajdonképpen amikor ’C’-ről ’D’-re váltunk, olyankor a 2 hang közötti távolsággal (ami ez esetben 2) eltolódik az összes hang. Ez egy egyszerű megoldás hangnem váltásra, gitáron is például csak a kezünket csúsztatjuk a nyakon hangnemváltáskor, és nem a fogásokon változtatunk. A valóságban viszont a zongora esetében ez kicsit bonyolultabb, a félhangok miatt. Mivel oktávonként két fekete billentyű kimarad, ezért ahogy eltoljuk a kezünket, a fogáson is kell változtatnunk, mert már más pozícióban lesznek a billentyűk. Habár megoldható lenne a billentyűk repozícionálása, úgy vélem mégis csak kézen fekvőbb ez a megoldás, eleve nem egyszerű megszokni a billentyűkiosztást, nehezítene a dolgon, ha a fekete billentyűk gombjai még változnának is minden egyes hangnem váltáskor. Természetesen az ábrára égetett hangok minden esetben igazodnak a beállított kezdőhanghoz. Ez a megoldás talán azoknak furcsa lehet, akik a valós zongorához vannak hozzá szokva, de akkor is egyszerűbb. Például énekesként belőni a saját hangterjedelműnkhöz passzoló hangnemet sokkal egyszerűbb egy kattintással.

4. ábra Személyre szabott beállítások a zongorán

Egy ’C3’-as kezdőhang esetén ’B3’-ig tart a hangok terjedelme (a hang melletti szám az oktávot jelöli). Ha ezt eltoljuk úgy, hogy ’D3’ legyen a kezdőhang, már kilépünk a 3. oktávból, ekkora az utolsó két hang már ’C4’ és ’Db4’ lesznek. Megfigyelhető, hogy amikor az ábrák melletti nyilakkal a 6. (legmagasabb) oktávra váltunk, a kezdőhang kiválasztása sem elérhető már, és ha előtte más volt beállítva, mint ’C’, vissza fog állni ’C’-re. Ennek egyszerűen annyi az oka, hogy a hanganyag hat oktávot fed le, azaz amikor a 6. oktávon más kezdőhangra váltanánk, a ’B6’ utáni hangok már nem szólalnának meg, így nem is lenne értelme a hangnemváltásnak.

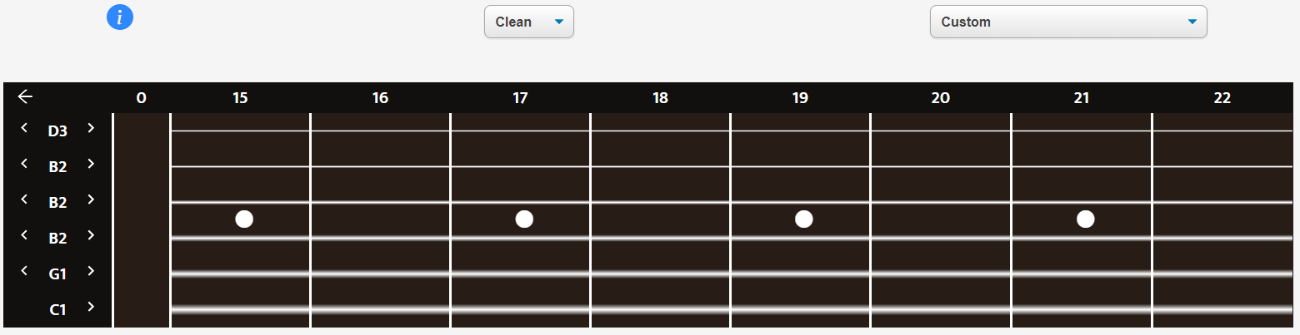
5. ábra A 6. oktávban a select le van tiltva

### Gitár

A gitár használata talán egy picit bonyolultabb, eleve nem lehetséges olyan jól implementálni, mint egy billentyűs hangszert, mivel egy átlagos gitáron 6 húr van, ezért lehetetlen a billentyűzeten minden húrt elkülöníteni. Az alkalmazás nincs felkészítve több húros gitárokra.

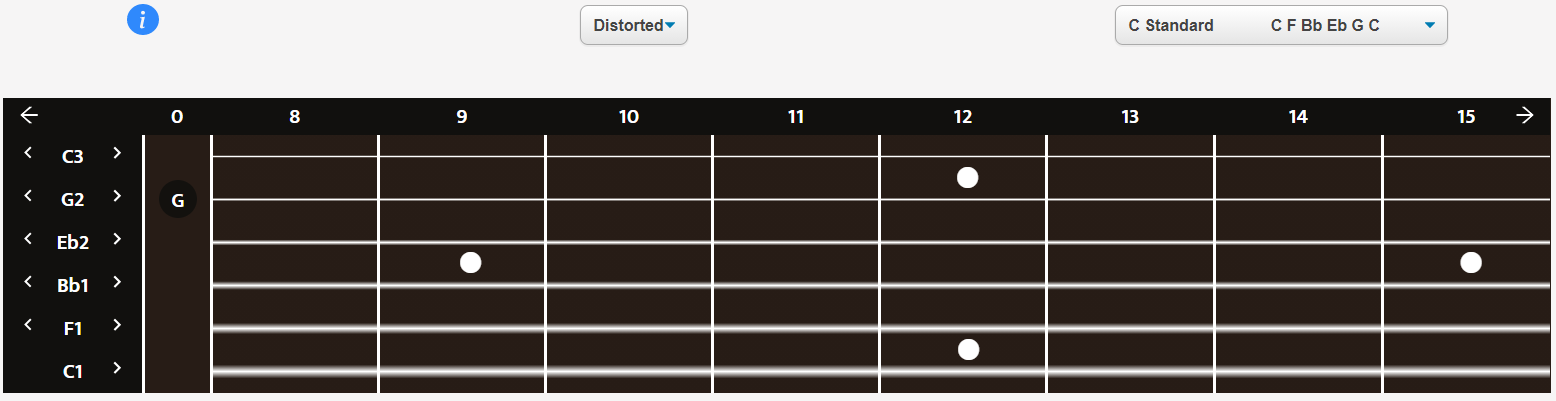
Az ábrán nagyon zavaró lenne minden hangot egyszerre megjeleníteni, a billentyűzet használata pedig nem bonyolult, így mindig csak a lejátszott hang jelenik meg. Egy info ikon ad egy tömör leírást, miszerint az alsó négy húr a billentyűzet négy „használható” során szólaltatható meg, a két legmagasabb húrnak viszont már nincs helye, így a ’shift’ billentyű lenyomva tartásával érhetőek el a középső két húr gombjaival. Az ábra jól szemlélteti a húrok vastagságával, hogy melyik-melyik, bár ebben az alkalmazásban ettől el lehet rugaszkodni.

Az info ikon melletti select egy olyan opció, ami a zongoránál nem volt még elérhető. A gitár hangszínét lehet állítani, jelenleg tiszta és torz hangzást állíthatunk be.

A hangolás realisztikus, azonban talán kicsit bonyolultabb, mint a zongoránál. Kezdetnek a legegyszerűbb a jobb oldali select használata, ahol előre beállított, gyakran használt hangolások közül választhatunk tetszőlegesen. A legelterjedtebb az ’E Standard’ hangolás, ez az alapértelmezett, amikor az oldalt megnyitjuk. Ha ezt megváltoztatjuk, az ábra is jelzi. A nyaktól balra minden húrhoz egy egyedi hangoló felületet is találunk, így kedvünkre tetszőleges hangolásokat is beállíthatunk (minden húr teljes terjedelemben hangolható, ami igaz nem túl reális, de nem látom értelmét ezt lekorlátozni), ha a presetek között nem találtuk meg amire szükségünk van. A kis nyilakkal fél hangonként tudjuk „tekerni” a húrt. Ha valamit elállítunk, a hangoláshoz tartozó select a ’Custom’ szót fogja kiírni.

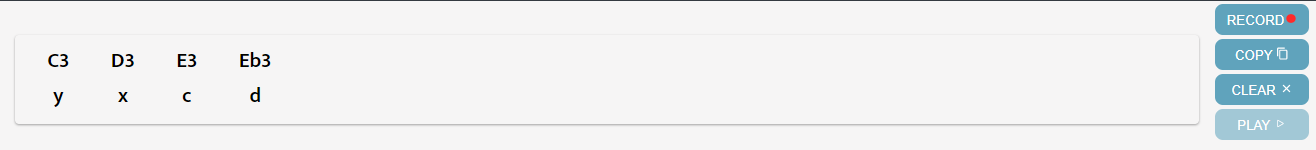
6. ábra Személyre szabott beállítások gitáron

Egy 6 húros gitár kevesebb hangot tartalmaz, mint egy zongora, így az ehhez tartozó hanganyag csak 4 oktávot fed le (igaz 2 féle hangszínben). A legmélyebb hang itt is a ’C1’, így, ha valamelyik húrra (optimális esetben a legalsóra) ezt állítjuk be, a nyíl eltűnik, nem tudjuk mélyebbre hangolni. Az ’E3’ a legmagasabb hang, amire egy húr hangolható. Ez talán kevésnek tűnhet, de ne felejtsük el, hogy míg a zongorán egy-egy oktávot (12 hang) lehetett játszani, egy gitár nyakán legalább 21 bund (szakaszok a gitár nyakán, tulajdonképpen minden bund más hangon szólal meg) van. Ez a gitár most 22 bundos, tehát a ’D5’ a legmagasabb hang.

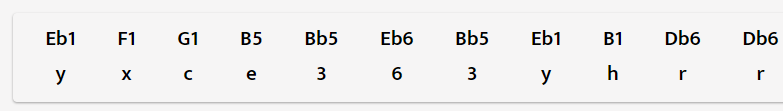
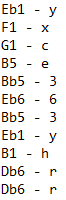
A nyak felett a bundok számozása látható, illetve két szélén egy-egy nyíl. Sajnos a billentyűzet nem épp a legalkalmasabb gitározásra, se a 6 húr, se a 22 bund nem fér el, a nyaknak mindig csak egy adott részét tudjuk kihasználni. Minden bund felett látszik annak sorszáma. Egy rögzített 0-s szakasz az első, ami az üres húr megpengetését jelenti (azaz úgy pengetjük a húrt, hogy nem fogunk le rajta semmit). A gitárok nyakán általában pöttyök jeleznek egyes bundokat, bár jelen esetben még számozás is van, a segítő pöttyök itt is megjelennek. A 3. 5. 7. 9. 15. 17. 19. 21. bundon egy pötty, a 12. bundon 2 pötty szokott lenni (a 12. az új oktáv első hangja, tehát ugyan az a hang egy oktávval feljebb, amit üresen pengetve hallanánk). A szélső nyilak mozgatása a nyak pozícionálására alkalmas. Ezt tulajdonképpen úgy képzeljük el, hogy a nyak melyik részén fogunk le hangot az ujjainkkal. A nyakon ahogy „mozgatjuk a kezünket” jobbra-balra, úgy változik a bundok számozása, illetve a pöttyözés is ennek megfelelően követi a kezünket. A rögzített üres húr nem változik ennek hatására, hiszen a valóságban is gyakran pengetünk üres húrt, akárhol is van a kezünk, mivel ahhoz nem kell lefogni semmit, maximum elengedni. Egyszerre 8 bundot tudunk kihasználni, a bal oldali nyíl nyilván 0 alá nem enged pozícionálni, a jobb oldali pedig 22-nél tűnik el. Egy húron több hang nem szólhat egy időben, ezért ilyenkor az adott húron a legutóbb játszott hang elnémul, ahogy a valóságban is. A lehetőségek tehát kicsit szűkebbek, de mivel a gitáron egy hangot több pozícióban is le lehet fogni, így a meglelő hangolással és nyakpozicionálással egész jól használható.

7. ábra Üres húr pengetés gitáron

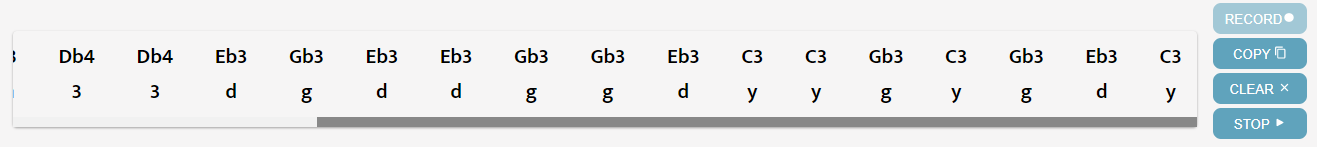
### Logger

A Logger egy olyan feature, ami minden hangszernél ugyan úgy működik. Ez a hangszerek felett található alapvetően üres sáv és a hozzá tartozó gombokat jelenti. Azért hívom Loggernek, mert elsőszámú feladata a leütött hangok logolása (megjelenítése), de ennél jóval többet tud.

8. ábra Felvétel készítés folyamatban

A sávtól jobbra található 4 gomb, amikkel kontrollálhatjuk. A ’RECORD’ gomb lenyomásával elindítunk egy felvételt, ekkor minden leütött hang meg fog jelenni a sávon, majd újra nyomásával leállítjuk a felvételt. Amíg a felvétel tart, a gombon egy kis animáció jelzi ezt. A sávon hang-billentyű párok fognak megjelenni, azaz a felső sor a lejátszott hang, az alsó pedig a lenyomott billentyűt jeleníti meg. A ’COPY’ gomb ezeket a párokat szövegként vágólapra másolja. Valójában ez csak egy alternatíva egy jobb megoldásra, amire később térek majd ki.

9. ábra A másolás ilyen formátumban adja vissza a hangokat

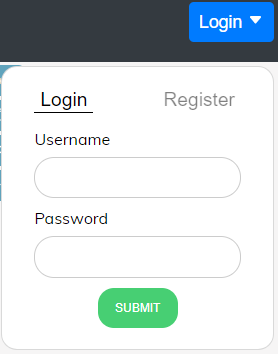
A ’CLEAR’ gomb kitöröl mindent, amit felvettünk. Előfordulhat viszont, hogy a felvétel maga jó lett, csak egy rossz hangot véletlenül beleütöttünk. Ha tudjuk, melyik hangokat szeretnénk eltávolítani, erre is van lehetőség. A sávon az eltávolítani kívánt hang-billentyű párokra kattintva kifejezetten csak az az egy elem fog eltűnni a felvételből. A ’PLAY’ gomb visszajátssza a felvételt. Habár ez nincs megjelenítve, de a hangok közt eltelt időt is rögzíti az alkalmazás, így pontosan úgy fogjuk visszahallani a felvételt, ahogy rögzítettük. Lejátszás közben a gomb felirata ’STOP’-ra változik. Ezzel tudjuk leállítani a felvételt, ekkor visszaáll ’PLAY’-re, ami előröl fogja indítani a lejátszást.

10. ábra Lejátszás folyamatban

Természetesen a ’RECORD’ és a ’PLAY’ kiütik egymást, egy időben nem lenne értelme használni őket. Ha lejátszás közben nyomunk a ’CLEAR’ gombra, a lejátszás megáll. A lejátszás közben egyenként törölt hangok még meg fognak szólalni, legközelebb viszont már nem. Felvett hangok nélkül értelemszerűen csak a ’RECORD’ gomb fog működni, a többi nem csinál semmit. Sajnos a szerkesztői felület primitívségéből adódóan a többszöri felvételindítás nem támogatott, mert a két felvétel közötti szünet meghatározása kivitelezhetetlen. A felvétel leállítása után tehát nem tudunk újabb recordot indítani, amíg a sáv tartalmát ki nem ürítjük.

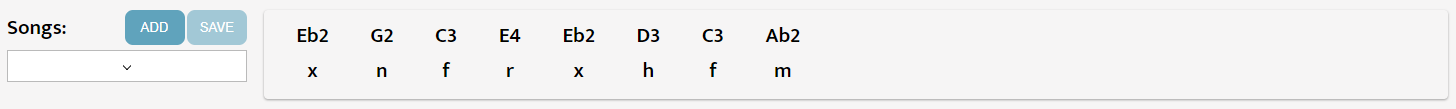
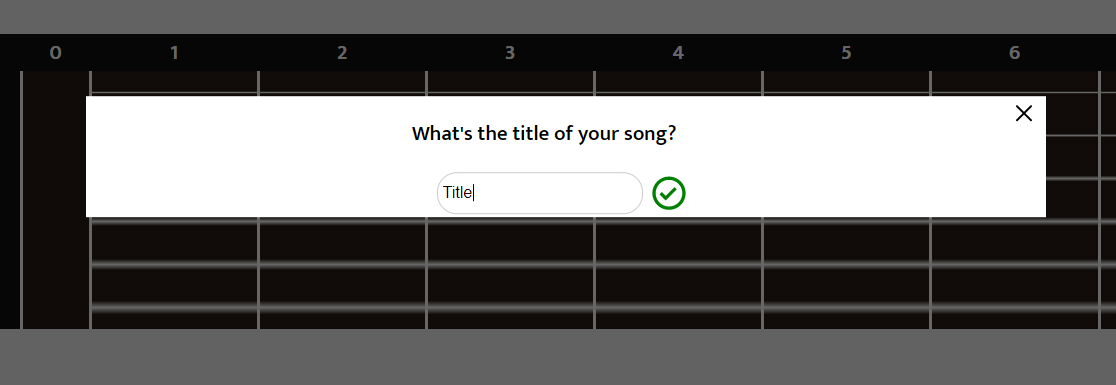
Bár funkcionalitásában megegyeznek, mégis minden hangszernek külön-külön, saját Loggere van. Tehát abban az esetben, amikor felveszünk zongorán valamit, majd gitárra váltunk, ismét üres lesz a Logger sáv. Az oldal újra töltéséig viszont minden hangszer meg fogja jegyezni a „saját számát”, tehát a hangszerek közötti váltogatással nem fognak eltűnni a felvett anyagok. A lejátszás és a felvétel egy másik hangszerre váltás esetében leáll.

### Felhasználó-kezelés

A fejlécen található Login gomb lenyomásával jön elő az autentikációs form. Itt lehet regisztrálni és belépni egy fiókba. A Register gombra kattintva lehet regisztrálni. Elég egy egyedi felhasználónév és egy jelszó a regisztrációhoz, az oldal nem fog személyes adatokat használni. Ha megadtunk egy legalább 5 karakter hosszú felhasználónév-jelszó párost, a submitra nyomva felugrik egy ablak, ami vagy a regisztráció sikerességéről tájékoztat, vagy arról, hogy már létezik egy fiók a megadott felhasználónévvel, ekkor újat kell megadni. Ha sikeres volt, akkor a Login gombra kattintva ugyan így be is lehet lépni. A bejelentkezett fiókot a program megjegyzi, tehát bezárás után legközelebb nem kell újra azonosítani.

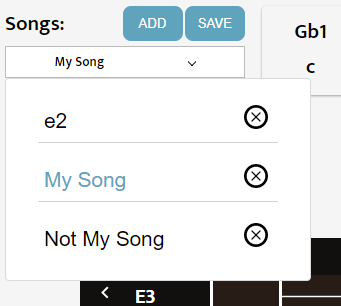
11. ábra Login form

A témabejelentőmtől itt egy kicsit elrugaszkodtam. Fejlesztés közben nem láttam már értelmét beállításokat elmenteni, mert csak pár kattintást spórolnánk minden használatkor. Helyette sokkal hasznosabb funkciónak tartom, hogy a felhasználó elmentheti a felvételeit, így inkább ehhez implementáltam egy toolt.

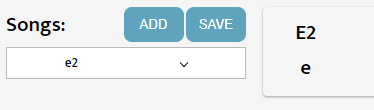
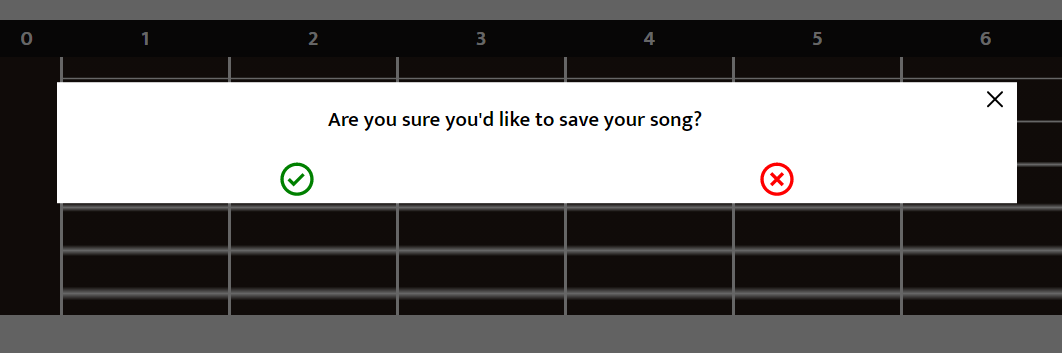
A belépés után a Login gomb Log outra vált, ezzel lehet majd kilépni. Egy új funkció jelenik meg a kezelőfelületen minden hangszernél, ez a Song Handler. Ennek segítségével van lehetőség egy felvett szám lementésére, így később bármikor el tudjuk érni és visszajátszani. Az ’ADD’ gomb lenyomásával tudunk új felvételt hozzáadni a számainkhoz. Ha nem készítettünk felvételt, nem lesz mit lementeni, így nem is fogja engedni a program. Ha nem üres a Logger, akkor már csak el kell nevezni a szerzeményt egy felugró ablakban. Az ilyen felugró ablakokat a jobb felső sarok ’X’ gombjával mindig bezárhatjuk. A pipára kattintva a számot hozzáadtuk a saját gyűjteményünkhöz, természetesen más felhasználó nem fog ehhez hozzáférni.

13. ábra Felugró ablak a szám elnevezéséhez

12. ábra A felvételt hozzá lehet adni a számaink közé

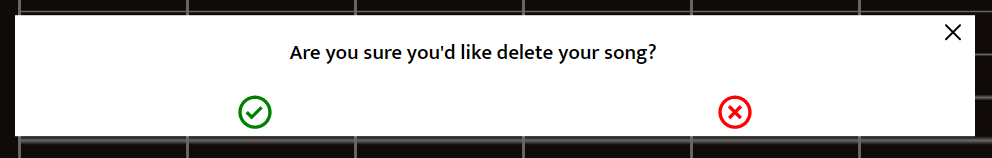
A hozzáadott szám címe meg fog jelenni ezután a lenyíló menüben, innen fogjuk tudni kiválasztani a számainkat, ha szeretnénk újra megnézni, lejátszani, vagy módosítani. Egy szám nevére kattintva a menüben a Logger betölti a hangokat. A szám innentől fogva amíg nem választunk mást helyette (vagy nem töröljük ki) az aktuálisan kiválasztottként lesz megjegyezve, ennek később lesz szerepe. Az aktuálisan kiválasztott számunkat a listát megnyitó gomb mutatni fogja. Egy számhoz mindig tartozik egy hangszer is, így ne lepődjünk meg, hogy egy gitárral felvett számot nem tudunk betölteni a zongora Loggerébe. Érdemes azt is tudni, hogy a hangolást viszont nem jegyzi meg a program, hogy ne állítódjon át a zenéink kiválasztásakor. A hang-billentyű párosból egyébként könnyű visszavezetni, milyen beállításokat használtunk.

14. ábra Dal lista

Egy szám betöltése után a Logger módosítható, ugyanúgy, mint eddig. A ’SAVE’ gomb lenyomásával tudjuk felülírni a már meglévő számunkat, itt jön elő az aktuális szám szerepe. Amit legutóbb választottunk, arra fog rámenteni a ’SAVE’. A gomb megnyomásával ismét felugrik egy ablak, hogy biztosan el akarjuk-e menteni a változtatásokat (kivéve persze ha nincs dal kiválasztva, olyankor nem működik a gomb). Mivel a jelenlegi verzió el fog veszni, ezért ilyenkor érdemes megnézni, hogy biztos jó számot választottunk-e ki.

16. ábra Felugró ablak a szám felülírásához

15. ábra A kiválasztott szám felülírható

Végül törlésre is van lehetőség, ha végleg el szeretnénk vetni egy lementett anyagot. A lenyíló listán minden cím mellett egy piros ’X’ gomb ad erre lehetőséget. Megint csak egy ablak fog felugrani, hogy megerősítsük, biztosan törölni akarunk-e, hiszen természetesen ezzel örökre elveszítjük az anyagot.

17. ábra Felugró ablak a szám törléséhez

1. **Fejlesztői dokumentáció**

Ebben a fejezetben részletesen kitérek minden megfogalmazott feladat megoldásának kivitelezésére, ismertetem az alkalmazás struktúráját, valamint a backend és a frontend különálló működését.

## Feladatok megfogalmazása, Tervezés

Egy webalkalmazásnak az összetettsége gyakran inkább a struktúrájából adódik, és nem az algoritmusok bonyolultságából. Előszőr is legtöbb esetben (most is) két külön alkalmazásról van szó, egy a háttérben történő folyamatokért felelős, a másik pedig a kezelőfelületért. Ezek kommunikációja eredményezi a tényleges applikáció megfelelő működését, de az összekötés már az utolsó lépés, először érdemes külön-külön megvizsgálni őket.

### Alapanyagok, Kódolás előtti teendők

Az app lényege a zenélés, így elengedhetetlen egy hanganyagbázis, azaz minden hangszer minden hangzásához legalább pár oktáv terjedelemnyi hanganyag, ami egész pontosan egy-egy hangból álló .mp3 fájlt jelent. Ezt a frontend fogja kezelni, később arra is kitérek, hogyan.

Egy apró megjegyzés: képeket nem használtam a projektben, a hangszerek ábrái is kóddal rajzolódnak ki.

### Verziókezelés

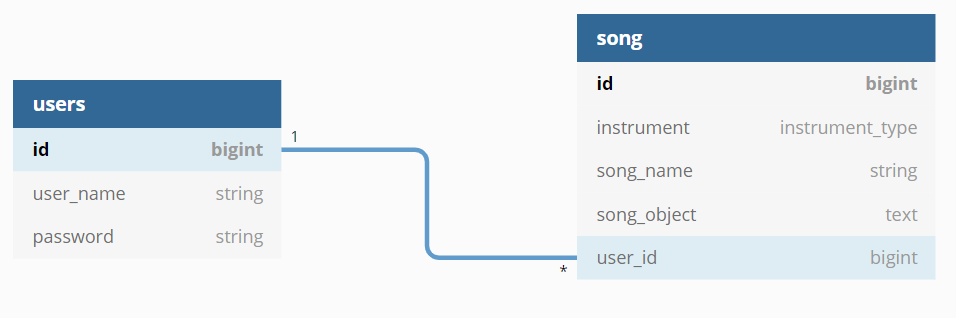
Mivel egyedül dolgozom a projekten és egy PCn, így remélhetőleg a verziókezelésnek nem lesz nagy szerepe, de a biztonság kedvéért mindenképp érdemes egy repositoryt létrehozni és rendszeresen commitolni, illetve pusholni. Egy számítógép meghibásodás, a kód elvesztésének esetén, vagy ha csak szükségem lenne egy régebbi verzió áttekintésére, életmentő lehet. Végül még ha nem is tenném a fejlesztést követhetővé, a deployoláshoz egyébként is szükség lesz egy repora. A verziókezeléshez gitet használok, és Githubon fog élni a repository, az alábbi linken:

<https://github.com/gyugyipeter/szakdoga>

### Backend

Érdemes a backendel ismerkedi először, mivel ez az, ami minden esetben önállóan is kell, hogy működjön. A frontend működéséhez, vagy legalábbis egyes funkciók működéséhez viszont elengedhetetlen a kettőjük közötti kapcsolat. Az alkalmazás funkciói többnyire frontendes megvalósítást igényelnek, így a backend kisebb.

A feladata a felhasználók kezelése és az adatbáziskezelés. Ehhez ugyebár szükséges egy adatbázis (a fejlesztés ideje alatt lokálisan), illetve (bár ez már nem a backend része, de vele fog közvetlenül kommunikálni) szükséges egy szerver, ami az applikáció elérését és az adatok hosszútávú tárolását biztosítja.

Az adatbázisnak két táblát kell kezelnie: egynek a felhasználókat kell számontartania, a másiknak pedig a dalaikat. Mivel kapcsolódnak egymáshoz, ezért relációs adatbázisra lesz szükség, erre a PostgreSQL megfelelő.

18. ábra Adatbázis modell

A teljes projekten szeretném a lehető legmodernebb technológiákat használni, ezért a backend megvalósításához Spring Bootot választottam. A Java-kezelő IDEk közül az IntelliJ IDEAt tartom a legalkalmasabbnak, így ebben a környezetben fejlesztem a backendet.

Nem gondolom, hogy túl kellene bonyolítani a Securityt ebben az alkalmazásban, mivel a felhasználóknak nem kell személyes adatokat megadnia, így ebbe csak minimális energiát ölök.

Végül a frontend-el való kommunikálás szokásos REST APIon keresztül fog történni.

### Frontend

A munka nagyrészét a frontend teszi ki. A kezelőfelület, annak designja és a logika nagy része itt kerül implementálásra.

Cél, hogy a kezelőfelület minél letisztultabb, minél egyszerűbben használható legyen. A designhoz nem használok külső libraryket, minden cssben lesz megírva.

A modern technológiát követve React Hooksot (azaz function based React-et) használok, ehhez VSCode-ot. Nyelvet tekintve JavaScriptet fogok használni.

A struktúra a fejlesztés közben sokat változhat, de egy általános terv nem árthat. Első lépésben, ami biztosan kelleni fog:

* Egy API Service, ami a backend kommunikációját fogadja
* Kezelni kell a hanganyagot, ehhez be kell olvasni a fájlokat
* Minden hangszert külön komponens-csoport alkosson, amik tartalmazzák a hangszer ábráját, a hangszeren való játék logikáját, és a hozzá tartozó beállításokat.
* Egy általános külső komponens a hangszerek számára, hogy a Loggert mindenhova be lehessen illeszteni.
* Egy sidebar a hangszerek közötti navigáláshoz
* Egy Header, itt történik az authorization egy dropdown formban
* A Logger kiegészítése autentikált módban, ehhez Modalok

### Deployment

Szükség lesz service-ekre az app hostolásához. Az adatbázishoz és a backendhez Herokut, a frontendhez pedig a Netlifyt fogok használni. Ezekhez kelleni fog egy-egy git repository.

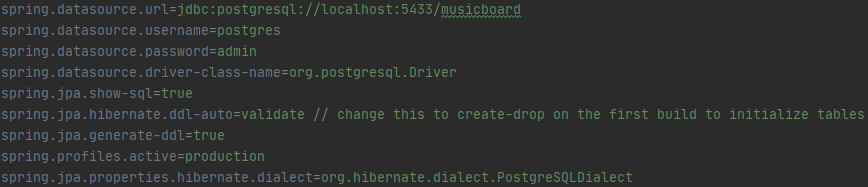
## Backend implementáció

Tekintsük át a Spring Boot projektet és az adatbázist.

### Projekt inicializálás, Dependencyk

Maven projektet választottam inicializáláskor, ami generál egy pom.xml fájlt, ebből kiolvasható minden tulajdonsága a projektnek - a nyelv Java 13, a Sping Boot verzió 2.2.5. A dependencyk érdekesebbek, ezekről röviden:

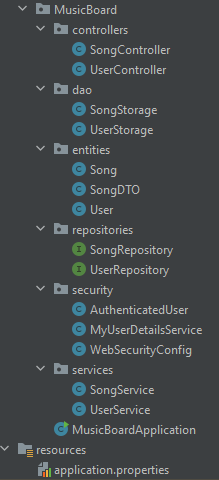
* Lombok – annotationöket ad a boilerplate kódok generálásához, emiatt nincsenek getterek, setterek és konstruktorok a kódban
* Spring Web – REST API controllerhez
* Spring Security – authenticationhöz
* Spring Data JPA – SQL adatbázis kezeléshez
* PostgreSQL Driver – a PostgreSQL adatbázishoz való csatlakozáshoz szükséges
* Spring Boot DevTools – a fejlesztést segíti

Bár az alkalmazás folyamatosan él Herokun, a fejlesztést érdemes lokálisan végezni. Ehhez vagy IDEből kell futtatni a Spring Boot appot, vagy commandlineból a ’mvn spring-boot:run’ paranccsal a projekt mappájában. Egy lokális postgresql is optimális a fejlesztéshez, ehhez viszont egy kis konfigurálás is szükséges a resources/application.properties fájlban, ehhez az alábbi példa segít:

19. ábra application.properties példa lokális adatbázis konfiguráláshoz

### Szerkezet

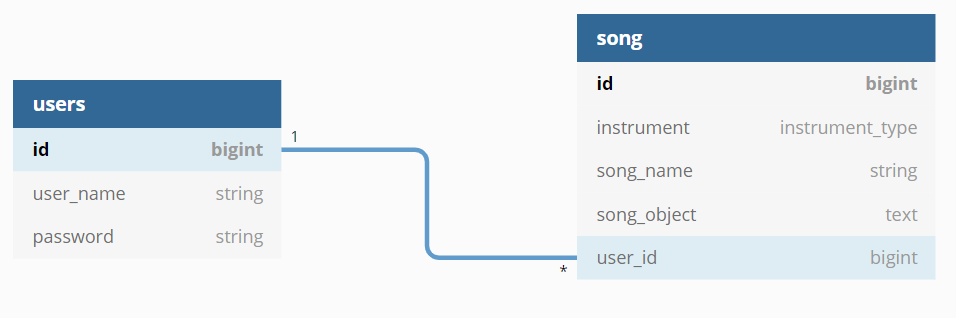
A szokásos Spring Boot package-szerkezet alkotja a projektet, azaz:



20. ábra Backend struktúra

* controllers – a database táblákhoz tartozó REST API controllerek, ezekben a classekben vannak meghatárzova az API endpointok, ez köt össze mindent
* dao – storagek a repository interfacek implementálása
* entities – Modellek és DTOk (data transfer objects), az adatbázis tábláinak tartalma (oszlopai) itt van meghatározva
* repositories – az adatbázis táblához tartozó interfacek
* security – Az authenticationhöz tartozó web- és API biztonsági konfigurációk
* services – Az adatbázis kommunikációjának logikája
* Ezeken kívül még egy Main class az app futtatásához és az application.properties fájl a fő konfigurációk meghatározásához, minden más már a fordítóhoz tartozik, vagy az IDE generálta.

### Adatbázis-kezelés, Service-ek

A Spring Boot legeltérőbb tulajdonsága más frameworköktől, hogy annotációkkal konfigurálunk mindent. Például a Spring Boot adatbázis felépítése és kezelése feltűnően kevesebb kódot igényel, mint más rendszerben. Kezdjük is az adatbázis inicializálásával. Az entities packageben vannak meghatározva a táblák és azok adattagjai. Egy model egy tábla, nézzük a felhasználókat tároló tábla osztályát.

21. ábra Adatbázis modell

Az annotationök meghatározzák a tábla nevét, boilerplate kódokat generálnak és mapelik a classt az adatbázis táblához. Ezek az osztályok implementálják a Serializable interfacet, mert nem tartalmaznak metódusokat, csak a táblák adattagjait. Először is kell egy ID természetesen, annotációkkal ezeket automatikusan generálja a Spring Boot. Egy userhez egy username és egy password tartozik, így ezeket az adattagokat kell definiálni a classben. Az annotationök meghatározzák, hogy ezek value-ja nem lehet null, illetve a username unique. A userekhez tartoznak számok, ezért kell a két tábla között egy one-to-many kapcsolatot létrehozni, ez a songs adattag.

A songs tábla tárolja a userek számait, ez hasonlóan épül fel. Itt is van egy ID és egy név. Az instrument csak egy enumból kaphat értéket, ami a frontenden létező hangszerek listája. A songObject egy dal hangjaihoz tartozó objecktek listája string formátumban. A frontend dolga ennek a parseolása, ott objectként lesz ez használható, de az adatbázisba stringként kell visszaküldenie. Mivel a varchar karaktertömb mérete limitált, ezért nem sok hangot tudna tárolni, de a @Lob annotációval már képes lesz nagy objecteket is tárolni. Végül a kapcsolat másik oldalát is meg kell határozni, mivel egy songhoz kell tartoznia egy usernek.

A dao (Data Access Object) package storage fájljai implementálják a repository interfaceket. A Spring Bootos repositoryk tartalmaznak kereső függvényeket, amiket implementálás nélkül lehet hívni a táblázatban való kereséshez adattagok alapján.

Egy adatbázisban alapvető funkció általában a hozzáadás, frissítés, törlés és adattag alapján keresés. Én viszont ezzel kicsit spóroltam, mivel a felhasználóknak nem kell semmilyen személyes adatot megadniuk. Így nincs is értelme a fiókjukat törölni vagy módosítani, ezért nem is adtam erre lehetőséget, tehát ezekhez a műveletekhez nincs implementáció. Ami a projekt szempontjából szükséges, az a felhasználó keresés és a hozzáadás, ezek az adatbázis műveletek lettek implementálva a UserStorage osztályban. A SongStorage-ban már szükséges az adatok törlése és módosítása, illetve a userekhez tartozó dalok keresése is. Így a dao fájlok tehát tartalmazzák az adatbázis műveleteket.

A services package a service fájlokat tartalmazza, amik a storage methodokat használva a műveletek logikáját tartalmazza. A UserService-ben van a register method, ami annyit jelent, hogy titkosítja a kapott jelszót és elmenti az adatbázisba (a titkosításról még később lesz szó). A SongService-en többnyire csak átmennek a methodok a SongStorageből, kivéve az addSong metódust. Az entities class ismertetésekor szándékosan nem említettem a SongDTO classt, ennek ugyanis itt van szerepe. A probléma abból adódott, hogy egy szám hozzáadásához az adatbázis a következő paramétereket várja: a dal címe, a hangszer típusa, a hangok (string object) és a felhasználó, akihez tartozik. Frontendről viszont nem szerettem volna elküldeni a usernek minden adatát (eleve tárolni sem), csak a userIDt. Erre megoldásként készítettem egy úgy nevezett DTO, azaz Data Transfer Object classt a számokhoz, ami ugyan azokat a propertyket tartalmazza, mint a Song entity (kivéve persze az ID), de a teljes user helyett csak egy userIDt. Ez arra volt jó, hogy így a service-ben az addSong methodban tudunk egy Song objectet építeni egy SongDTO objectből. Csak ki kell keresni a paraméterként kapott userID alapján a felhasználót az adatbázisból, majd a builder() segítségével felépíti az objektumot a SongDTO propertykkel és a megtalált userrel.

### Controller

Miután a szükséges metódusok mindenen végig mentek, a controller osztályokban az endpointok definiálásánál használjuk is őket. Az osztályokon lévő @Controller annotation biztosítja, hogy ezek a classek képesek legyenek requesteket kezelni, a @RequestMapping pedig mapeli a requesteket egy útvonalhoz. A @CrossOrigin engedélyezi a ’CORS’-ot (cross-origin resource sharing) minden metódusra az osztályon belül. Az endpointok mind ugyanúgy épülnek fel: az annotation meghatározza a request típusát, a metódus kérhet paramétert a linkben, vagy requestbodyt, majd megpróbálja végrehajtani a kérést, hiba esetén exceptiont dob.

Először nézzük át a UserControllert. Itt csak két végpontot használunk a belépéshez és a regisztráláshoz. Mint már említettem, egy elég primitív, mondhatni rendhagyó beléptetési rendszere van a programnak. A register megpróbálja elmenteni a kapott requestbodyt az adatbázisba és visszaküld egy üzenetet, ha sikerült. Ebben nincs semmi érdekes. Az autentikációt viszont tulajdonképpen kihagytam az applikációból, mert csak biztonsági szerepe lett volna. A login mindössze csak megnézi, hogy a kapott username-password párost tartalmazza-e az adatbázis (a jelszó ugye titkosítva került be az adatbázisba, de szerencsére a Spring Boot megoldja, hogy ezzel már ne kelljen foglalkozni), és visszaküldi a választ. Ennek előnye, hogy egyszerűbb a kód, hátrányát a security részben kifejtem. Ezeken kívül még írtam endpointokat a fejlesztés során, utólag viszont már nem használtam fel őket, de kommentként bent hagytam mindent, hátha később még jól jönnek. A SongController természetesen hasonlóan néz ki, csak ez a song táblával kommunikál. Minden végponton meghívódik a megfelelő service metódus. Az addNewSong várja a korábban említett DTO objectet.

### Security

A security package tartalmazza a biztonsági konfigurációkat. Mivel kihagytam az autentikációt, így az ehhez kapcsolódó classeknek nincs szerepük. Ez a hiány azt jelenti, hogy nem lehet biztonságosan kezelni a felhasználók hozzáférését más adataihoz. Persze egy átlagos felhasználónak ez nem tűnik fel, egy kicsit tapasztaltabb programozó viszont tulajdonképpen a böngésző editorában manuálisan meg tud változtatni adatokat, vagy egy REST API toolal. Emiatt bár nem a legkedvezőbb megoldás, de mint már kifejtettem, a felhasználó eleve nem ad meg személyes adatot. Más fiókjába való belépés pedig eleve nem lenne ilyen egyszerű, mivel a jelszavak titkosítva vannak tárolva az adatbázisban, tehát nem lehet visszakérni. A titkosítást egy nagyon egyszerű Springes import végzi, a BCryptPasswordEncoder. Ez egy BCrypt algoritmussal hasheli a jelszót, és az így kapott kódot igaz APIon le lehet kérni, de visszafejteni lehetetlen.

## Frontend implementáció

Tekintsük át a React projektet.

### Projekt inicializálás, Dependencyk

NodeJS alapú projektekben a package.json fájl tartalmazza a dependencyk listáját:

* axios – http kliens az API kommunikácihoz
* copy-to-clipboard – Egy library ahhoz, hogy egyszerűen lehessen vágólapra másolni egy stringet
* react-click-away-listener – egy eventet biztosít egy komponensből való kikattintáshoz
* react-howler – egy audio library a hangfájlok kezeléséhez
* react-icons – egy libary rengeteg ikonnal
* react-keyboard-event-handler – a keyboard eventekhez

Természetesen a frontend fejlesztést is lokálisan érdemes fejleszteni, a futtatáshoz elég a frontend mappában egy ’npm i’ és egy egy ’npm start’ parancs.

### Szerkezet

A gyökér mappa tartalmazza a node modulokat, a generált fájlokat és a package.jsont, ezekről nincs mit beszélni. Az src mappában van a kód, a program kiindulópontja az App.js. A sounds mappa tartalmazza a hangfájlokat, a domain mappa pedig segédfüggvényeket. A komponensek tartalmazzák a UI elementeket és a program logikáját, ezek a components mappában vannak. Ebben már csak a hangszerek vannak külön mappákba kiszervezve. A CSS fájlok többnyire komponensekkel vannak párban.

### Autentikáció, API Context

Az App.js a program kiindulópontja. Ez csak komponenseket renderel egymásba ágyazva. A külső komponensek a Contextek, amik olyan stateket és functionöket tartalmaznak, amiket a program bármely részéről el szeretnénk érni. Az AppContext vegyesen tartalmazza az app működéséhez szükséges adatokat, az ApiContext pedig a REST API kommunikációhoz szükséges endpoint hívásokat és userkezelést. Ezekbe van beágyazva a Header és a Body, így biztosítva azt, hogy mindenhonnan elérhető lesz minden a contextből, illetve a contextek egymással is tudnak kommunikálni.

Nézzük át mit tud az ApiContext. Az API fetchelést az axios library fogja végezni. A http konstans tartalmazza az alapkonfigurációt, erre már csak a hívás típusát kell meghatározni. A register esetén például egy ’POST’ methodról van szó, ami a megfelelő urlel és request bodyval már meg is hívható. Paraméterként természetesen kellenek az autentikációhoz szükséges adatok, ezek fogják alkotni a request bodyt. A hibákról minden esetben, illetve most épp a sikeresség visszajelzéseként is alerteket dob a program. A backendhez hasonlóan itt is try catch statementekkel tudjuk kezelni a hibákat.

A login is nagyon hasonlóan néz ki, csak itt el is kell tárolni valamilyen formában a bejelentkezett fiókról valamilyen információt. Az IsLoggedIn state használható minden olyan esetben, amikor a bejelentkezett állapottól függ valami. A userIDra is szükség lesz persze, viszont ezt state helyett inkább local storage-ba menti a program, így nem kell minden használatkor újra bejelentkeznie a felhasználónak. A logoutról a backendnek nem kell tudnia, így nincs szükség API hívásra.

A getAllSongsByUser egy rendhagyó method, itt Callback lett alkalmazva. Erre azért van szükség, hogy megakadályozzunk felesleges rendereket. Ez elsősorban egy Effectbe kerül bele, mivel a számok listáját automatikusan szeretnénk frissíteni minden render esetén, nem csak egy-egy gomb nyomásakor, mint máshol. Ezen felül frissíteni kell a listát amikor hozzáadunk, vagy törlünk az adatbázisból. Minden híváskor frissül songList state, ami tartalmazza az egyes dalokhoz szükséges adatokat. A getSong egy song ID alapján kéri el a szám Loggerbe illeszthető adatait ebből a stateből.

Az addSong hozzáadja a Logger tartalmát az adatbázishoz. Ehhez paraméterként a usernek meg kell adnia a címet persze, illetve az objectet is kéri a Loggerből. Azért nem elég közvetlen contextből elkérni a Logger tartalmát, mert annak formátuma array, nekünk pedig egy JSON object kell. Az objectet persze az array alapján készíti a program, de erről a SongHandler gondoskodik. Instrumentként az aktuálisan renderelt hangszert el tudjuk kérni a másik contextből, viszont a backend enumjában az instrument típusok nagybetűvel vannak megadva, ezért kell rá egy toUpperCase(). A userIDt már könnyedén ki tudjuk szedni a local storageból. Mivel a mentéssel már egy dallá válik a Logger tartalma, ezért ilyenkor a kiválasztott dalhoz tartozó stateket is felülírjuk.

Az updateSongban nem történik semmi különös, csak a Logger stringifyolt verzióját át kell dobni. A deleteSoing picit érdekesebb, mert a listáról bármikor bármelyik szám törölhető, nem kell kiválasztani a dalt, amit törölni szeretnénk. Ha viszont az épp kiválasztott dalt szeretnénk kitörölni, akkor frissíteni kell a stateket, hogy ne maradjanak bent a nem létező szám adatai.

Az első szemmel látható komponens, amit az App.js renderel, az a Header (Header.js). Itt hívódnak az autentikációhoz tartozó függvények az API contextből. Először kezelni kell a dropdownt, ehhez szükséges egy state, amit egyes eventek módosíthatnak és számontartja, hogy meg kell-e jeleníteni a dropdownt. Az egész egy ClickAwayListenerben van, aminek az onClickAway propjában tudjuk a stateket állítani. Erre most csak azért tértem ki külön, mert máshol is használtam ugyan ezt a megoldást. Egy conditional render határozza meg, hogy login vagy log out button legyen a Headerön, itt jön például jól az isLoggedIn state az API contextből.

Az autentikációhoz természetesen egy form kéri be az adatokat. A Login/Register opció a UIon nem változtat semmit, de a háttérben a submit gomb eventjét ők állítják. A form submitja fogja meghívni a megfelelő methodot a contextből. Az e.preventDefault() megakadályozza, hogy a linkbe bekerüljenek az adatok. Az inputok onChange eventje folyamatosan állítja a statetek, amiket a submitnál paraméternként átad a program a login/register függvénynek. Habár a backend nem vizsgálja ezt, de az inputokban meg van határozva, hogy minimum és maximum hány karakter hosszú lehet a felhasználónév és a jelszó.

### Navigálás

A Header után térjünk át a Body (Body.js) komponensre. Itt használjuk a Routert. Egy single page applicationről van szó, tehát túl sok navigációra nincs szükség. Az egyetlen rooting a hangszerek között történik, jelenleg tehát csak két path létezik, a …/piano és …/guitar. Ilyenkor viszont nem frissül az oldal, csak az aktuális hangszer komponense renderelődik ki. A navigáció egy sidebaron történik, ezt tartalmazza a komponens (ehhez most külön .css fájlban van a design). Ha a ’Link’ komponenssel körülveszünk egy UI elemet, azzal tulajdonképpen rákerül egy event, ami átnavigál. Ezután a ’Switch’-ben megmondjuk, hogy melyik path melyik hangszer komponensét renderelje. Ennek a formája …/’hangszer’, viszont az oldal megnyitásakor normál esetben ez csak …/ ezért erre is rákerült a zongora komponense. A hangszerek root komponense mindig ’hangszer\_neve’Body (pl. PianoBody) és minden esetben ugyan úgy néznek ki. Először is egy useEffect hívás, ami mindig végrehajtja a műveleteit, amikor az adott komponens renderelődik. Ezek csupán annyit tesznek, hogy ha a korábban használt hangszernél nem állítottuk le a felvételt, vagy épp legjátszottunk egy dalt, akkor ezeket megállítja a context statek settereivel. Ezután rendereli a Loggert és az aktuális hangszer ábrázolását, beállításait.

### Logger

A Logger egy univerzális tool az alkalmazásban, minden hangszernél használható, és ő nem tud arról, hogy épp melyik hangszerrel használjuk. Ahhoz képest, hogy milyen sok mindent tud, nincs túl sok kód a Logger.js fájlban. Ez főleg azért van, mert az itt használt műveletek nagyrésze a contextben van definiálva (AppContext.js), ezért is van olyan sok elem behúzva. Minden hangszernek külön Loggere van tehát. A Logger a logs statet tartalmazza. Ez egy olyan object, ami a hangszerekhez külön-külön vezet egy arrayt, amit a hangszerek keyeventjei tudnak feltölteni (vagy egy adatbázisból importált szám). Contextből mindig lekérhetjük az éppen használt hangszert, és így le is tudjuk kérni a megfelől arrayt. Ez a megoldás biztosítja egyébként például azt is, hogy ne vesszenek el a logok, amikor másik hangszerre navigálunk. A Logger sávja tehát egy egyszerű mapeléssel feltölti a logokat. A fájl ezután már csak a gombokat hozza létre, de minden onClick event az AppContext.js-ből hívja a megfelelő functiont, ezért nézzük most át azokat.

A record az isRecording boolean változó értékét változtatja. A keydown eventek mindig meghívják az addLog függvényt, ez viszont csak akkor csinál valamit, ha az isRecording true. Ha a hangszerhez tartozó (ezt a currentInstrument state segítségével határozza meg, ez nagyon gyakran jól fog jönni) logs üres array, akkor a startRecording statet az aktuális dátumra állítja. Ezután a logs currentInstrument arrayéhez hozzáad egy objectet, ami a logot tárolja, egy unique idt (később lesz hasznos) és egy timingot. A timing az aktuális dátumból vonja ki a startRecordingot, ami a két hang között eltelt időt fogja eredményezni. Mivel minden log tudja, mennyi idővel később szólal meg az első hanghoz képest, így a lejátszás megfelelően tudja tartani a szüneteket a hangok között.

A copyLogs csak egy npm package segítségével vágólapra másolja a logokat a megadott formában, a clearLogs kiüríti a log-arrayt.

A playLogs végig iterál a currentInstrument arrayén és meghívja a megfelelő playSound függvényt. a setTimeout a timingokkal beidőzíti a playSoundok lefutását. Mivel nem ’alszik’ a függvény, tehát nem valós időben tölti be a logokat, ahogy halljuk a legjátszást, ezért a globális timeIds arraybe pusholja a timeoutokat. A stopPlaying úgy állítja meg a lejátszást, hogy meghívja a clearTimeoutot a logokra, azaz a várakozó hangok megszűnnek, a timeIds array kiürül.

A removeLog a kattintott logot törli csak, ehhez kellett az log.id. Az arrayből filterezi a kiválasztott hang IDját. Egy log.idt azért az aktuális idő és a log.key alkotja, hogy biztosan egyedi legyen.

### Song Handler

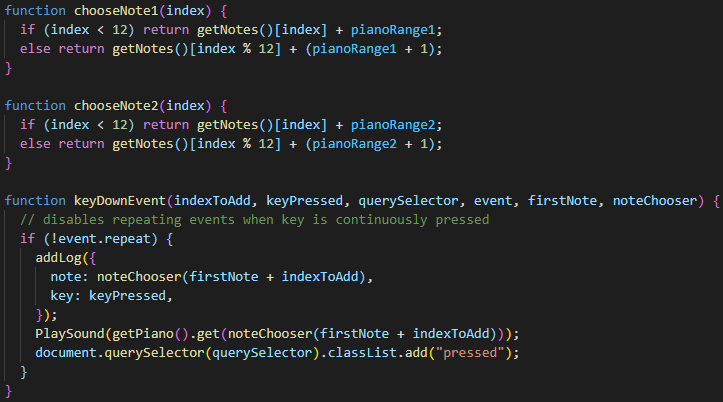
Bejelentkezett állapotban a Logger kiegészül a SongHandler (SongHandler.js) toolal. Ez használ Modalokat, amik szintén saját komponensek (Modal.js). Egy Modal egy show booleant kap propként, ami meghatározza, hogy meg kell-e jeleníteni, egy bezárást kezelő függvényt, és a tartalmát childrenként (ezt nem kell paraméterként átadni, a React tudja mi az a children). A SongHandlerben minden modalhoz tartozik két függvény, amik kezelik a megjelenést és a bezárást. Az addNewSong a hozzá tartozó modal formjából kinyeri az input valuet, a loggerből egy stringesített JSON objectet hoz létre, ezeket adja paraméterül az API hívásnak. Az overwriteSong is ugyan így működik.

A SongSelector az APIon lekért szám-listára mapel és SongListElementeket hoz létre az objectekből. Ez a számra való kattintásra az IDja alapján lekéri a számot APIon, az ’X’ ikonnal pedig törölteti.

### Hangfájlok importálása

A NoteFilePairs.js egy segédfájl, ami könnyen elérhetővé teszi a file-okat az app számára. Egy-egy hangzáshoz a /sounds mappa tartalmazza az audio fileokat. Ezekben az .mp3 kiterjesztésű file-okat külön arrayekbe tölti, majd egy map objectet készít, hogy könnyedén hozzá lehessen ezekhez férni. A value a fájl, annak nevéből pedig szerencsére substringel könnyen ki lehet szedni a hang nevét. Így tehát készül minden hangzáshoz egy olyan map object, aminek egy eleme például így néz ki: A3 - …A3.mp3. Ezeknek a mapeknek a getterei teszik elérhetővé a fájlokat. Ha például torz gitár hangzással szeretnénk az A3 hangot lejátszani, akkor azt egy getDistortedGuitar().get(A3) hívással könnyedén elérjük. Emellett a zenei hangok neveit egy ’notes’ arrayben tárolja, amit szintén getterrel el lehet kérni, ez sokszor jön jól.

### Piano

Egy hangszer bodyja mindig a Loggert és az adott hangszer komponensét tartalmazza. A hangszerek felépítése hasonló, de működésük eltérő, ezért most csak a zongoráról lesz szó első sorban, ami a PlayPiano (PlayPiano.js) komponensben valósul meg. A hangszerek billentyűzeten szólaltathatóak meg, ezért először is szükség van egy keyboard event handlerre, ez a HandleKeyPress. Egy switch kezeli minden billentyű eventjét. A library sajnos egy komponensnek csak egy típusú eventet enged használni, ezért kétszer kellett minden caset felvenni, egyet a keydown eventekre, egyet a keyup eventekre. A keydown a billentyű lenyomására lejátssza a hangot és ráteszi a ’pressed’ classt, ami animációt rak az ábrákra, majd a keyup event ezt a classt leveszi. A paraméterek a következőben befolyásolják a lejátszást: A ’keyPressed’-et akkor használja fel, ha az isRecording true, ekkora hozzáadja a logot a logshoz. A querySelector az animáció kezeléséhez szükséges. Az event megakadályozza, hogy egy gomb lenyomva tartása esetén folyamatosan bekerüljön a hang a logokba.

22. ábra Zongora key event handling

A firstNote contextből elérhető, ez egy select value-ja. A noteChooser egy function, ami a megfelelő hangot kéri el a zongora fileok mapjéből a paraméterül kapott index segítségével. Ez az index a firstNote és az indexToAdd összege, mivel a firstNote state azt határozza meg, hogy a zongora bal szélső (azaz legmélyebb) billentyűje milyen hangra van állítva, az indexToAdd pedig az ettől a billentyűtől való távolságot jelenti, azaz, hogy hány hanggal magasabb az elsőhöz képest. Például, ha a firstNote 0, akkor a korábban definiált notes tömb nulladik eleme, azaz a ’C’ hang lesz az ’y’ billentyűre binde-olva. Ha a ’c’ billentyűt megnyomjuk ez az ’y’ hangjától 4 távolságra van (y-s-x-d-c – C-Db-D-Eb-E), tehát a indexToAdd 4 kell, hogy legyen. A noteChooser function tehát a keresett hangot már megkapja, de a megfelelő oktávot még nem tudja. Ezt a ’pianoRange1’ és a ’pianoRange2’, contextben élő statek kezelik. Ez a függvény tehát ezzel a két információval már tudja returnölni a keresett map kulcsot. Egyetlen dolog, amit még kezelnie kell, hogy az index nem-e lépett át egy magasabb oktávba (tehát az index nem-e nagyobb, mint a notes tömb hossza), mert ilyenkor előröl kell indítani az indexelést, és egy oktávot hozzáadni. Ez például olyankor történik amikor a firstNote egy ’B’ hang és megnyomjuk az ’x’ billentyűt. Ekkor már eleve 11 a firstNote értéke, mert a ’B’ hang a legmagasabb egy oktávban, tehát ha az ’y’ után bármilyen gombot nyomunk, az már a következő oktávban kezdődő skála része. Azért szerepel néhány function és variable duplán, hogy külön lehessen kezelni a két ábrát, ahol szükséges.

A PlayPiano komponens tartalmazza tehát ezt a handlert. Utána rendereli a zongorához tartozó beállításokat, létrehozza a toggle-öket, rendereli a zongora két ábráját a hozzájuk tartozó oktávok statejeit állító nyilakkal (a legmagasabb értéknél a firstNote-okat azért állítják, hogy ne indexeljen túl a fileok map-jén nem létező hangokat keresve), végül az ábrákhoz tartozó oktávokat is megjeleníti.

Az előbb említett beállításokat a PianoSettings (PianoSettings.js) komponens tartalmazza. Ez csupán két selectet tartalmaz, amik a firstNote stateket kezelik. Két fontos dolgot emelnék ki ezekkel kapcsolatban. Az egyik hogy az onChange eventeken lévő blur() azért fontos, mert ha nem kerül le a fókusz a selectről, a billentyűkkel elállíthatódnak (pl. a ’c’ billentyű kiválasztaná a ’C’-t, mint option). A másik, hogy a react lokális futtatáskor egy warningot dob a böngészőben a ’selected’ használata miatt. Sajnos nem tudok erre megoldást, mivel szerintem eleve értelmetlen a warning. A ’selected’ ugyanis nem default value-t jelent egy selecten, ahogy azt a warning állítja.

Végül az ábrák kirajzolását tartalmazó PianoRepresentation (PianoRepresentation.js) komponenst is nézzük át. Propsként megkapja, hogy melyik ábrát fogja kirajzolni, és a hozzá tartozó billentyűket. Tulajdonképpen csak fehér- és fekete billentyűket rajzol a megfelelő sorrendben, a hozzájuk tartozó hangokkal és billentyűkkel. Mivel a sorrend adott, a billentyűket csak elég sorban elkérni a propként kapott keyBindings tömbből. Az id property a querySelector miatt kell, az animációkhoz. A note esetében is segít a sorrend, mert mindig csak egyel nagyobb indexel kell lekérni a hangot a ’notes’-ból (itt is ügyelve a maradékos osztásra a túlindexelés miatt). A fekete-fehér billentyűk a JS kódban nem igazán különböznek, nyilván a cssnek van itt nagyobb szerepe. Még egy apró megjegyzés: Az egymás mellett lévő fekete-fehér mindig egy spanbe vannak rendezve, de ennek csak a design szempontjából van szerepe.

### Guitar

Mindenek előtt felhívnám a figyelmet arra, hogy sokban hasonlít a gitár kódja a zongoráéhoz, ezért csak az eltérő dolgokat fogom részletezni.

A PlayGuitar (PlayGuitar.js) ugyan olyan switch-case-ekkel kezeli az eventket, mint a PlayPiano, csak az eventek mások. A hangfájlok lejátszása annyiban eltér a zongorától, hogy nem zenghet ki minden hang addig, amíg el nem hallgat. Egy gitáron egy húron egyszerre csak egy hang szólhat, ezért folyamatosan tárolni kell a legutóbb lejátszott hangot (azaz a Howl objectet) minden húron (contextben lastNotesOnString object), hogy meg lehessen állítani, amikor legközelebb használja a felhasználó ugyan azt a húrt. Emiatt a PlaySound helyett a gitárnak saját playere van, a PlayGuitarSounds. Ez annyiban más, mint az eredeti, hogy minden hang lejátszása előtt hív egy stop()-ot a húrhoz tartozó legutóbb játszott hangra, és az új Howl objectel felül írja azt. Ehhez viszont paraméterül kell kapnia a húr számát is. Szerencsére a billentyűk a beállításoktól függetlenül hozzárendelhetőek a húrokhoz, ezért a keyDownEvent a keyPressedből ezt meg tudja állítani.

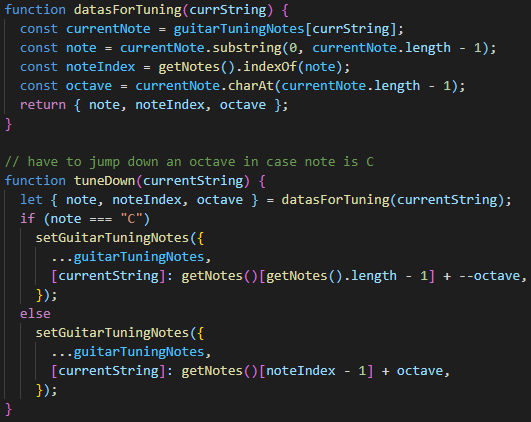
A gitárhoz tartozik egy segédfájl, ami a domain mappában található (GuitarHelper.js). Ennek a getStringNumber függvénye határozza meg a húr számát a lenyomott billentyű alapján. A chooseNote (zongoránál NoteChooser) is egy picit bonyolultabb, mint a zongoráé. Az indexnél nem csak az éppen megjelenített bundok számítanak, hanem a nyak csúsztatása is. A hang meghatározásához tehát az indexToAdd (bal szélső billentyűtől való távolság) hozzáadásán kívül még a nyak-csúsztatások számát is hozzá kell adni a firstNote indexéhez, akkor, ha nem üres húrt játszik a felhasználó, azaz nem a sor bal szélső billentyűjét üti le. Az oktáv meghatározása is kiegészül egy vizsgálattal, mivel az alkalmazás egy 22 bundos gitárt reprezentál, és ez két oktávnyi ugrást is lefedhet, ezért, ha a generált index 24-nél nagyobb, ami ugye két oktávnyi hangot jelent, akkor két oktávot kell hozzáadni. A keyPressek tehát kezelve vannak, ezután a komponens még a beállításokat rendereli és az ábrát (plusz a nyak tologatásáért szolgáló nyilakat, de ennek csak design szempontjából van szerepe).

23. ábra Gitár key event handling

A beállításokat a GuitarSettings (GuitarSettings.js) komponens tartalmazza. Először csak egy info ikont renderel, ami hoverelve megjelenít egy tooltipet, ami egy kis guide-ot tartalmaz a használathoz, de ebben nincs semmi érdekes. A gitárhoz több hangzás is tartozik, ezért az egyik select ezt kezeli. A kiválasztott hangzásokat egy contextben lévő state kezeli. A következő select előre beégetett, gyakori hangolások közül enged választani. A hangolások lekéréshez ismét a GuitarHelper nyújt segítséget, egy szimpla getter a hangolás neve alapján returnöli a húrokhoz tartozó hangokat, és ezeket beállítja a húrokra. A már említett warning itt is előjön a selected használata miatt az utolsó, hidden optionön.

Végül az ábra kirajzolását is nézzük át, ez a GuitarNeck (GuitarNeck.js) komponensben van. A gitár nyaka egy táblázat. A headerStyle meghatározza a bund számozása alapján, hogy kell egy az adott pozícióra pötty a nyakra. Egy stringet returnöl, ami a táblázat headerének a classname-jét határozza meg. A táblázat bodyjának sorai egy-egy String (nem mint string típus persze, hanem mint gitárhúr) componentet renderel, amik propként megkapják az összes bund IDját. Ezeket azért kell számon tartani, hogy a billentyű leütésekor a megfelelő pozícióban jelenjenek meg a játszott hang.

Habár a hangoláshoz használandó első három td nem a húr része, de ahhoz tartozik. Egy nyíl onClick eventje meghívja a hangoló-függvényt (tuneDown és tuneUp). Itt megint fennáll a már jól megszokott probléma, a ’notes’ túlindexelése esetén az oktávot is növelni vagy csökkenteni kell. A probléma kezeléséhez szükséges adatok összegyűjtéséhez meghívja a datasForTuning segédfüggvényt. Ez returnöl egy objectet, ami a húr alapján magába foglalja a hozzá tartozó hangot, annak indexét, és az oktávot.

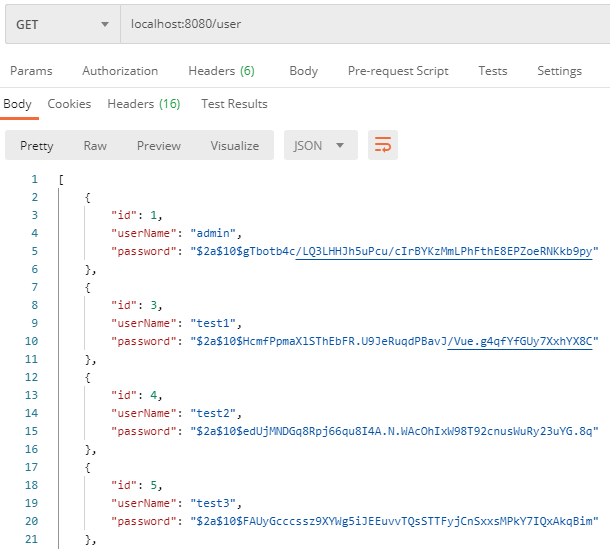


24. ábra Gitárhúr hangolás

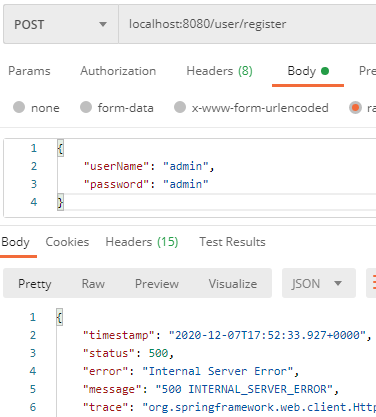
A hangolás után már csak a JSX elementek rajzolódnak ki, a kapott IDkat felhasználva. A displayIndex a lejátszott hangot adja vissza (megint csak a fix sorrendet kihasználva), ez viszont alapértelmezetten el van rejtve, csak a ’pressed’ classname-el látható, amit a gitár esetében is kezel a keyEventHandler.

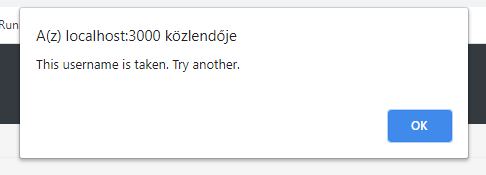
## Tesztelés

Az alkalmazáshoz nem tartoznak automatikus tesztek, ezért black-box techikával mutatok be use-case-eket. Egy kezelőfelülettel rendelkező alkalmazás szerintem akkor van jól felkészítve felhasználói hibákra, ha megjelennek grafikusan a hibák következményei (persze most nem bugokra gondolok, hanem kezelt hibákra). Ezért azt gondolom, hogy screen shotokkal sokkal jobban be lehet mutatni a teszteseteket, mint írott szöveggel, szóval többnyire Postman és böngészős screenshotokkal mutatom be a tesztek eredményeit.

****

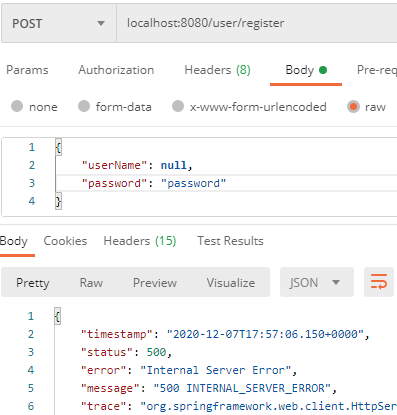
25. ábra GetAllUsers hívás, az alkalmazás nem használja, de a fejlesztés során hasznos

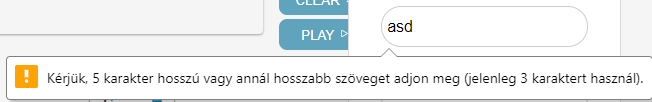
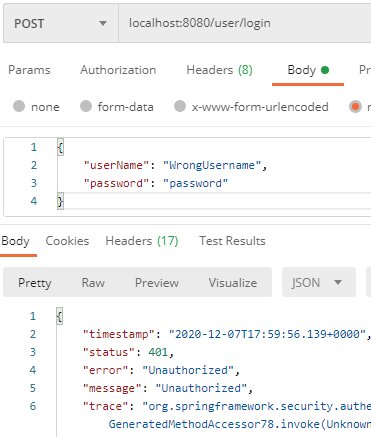
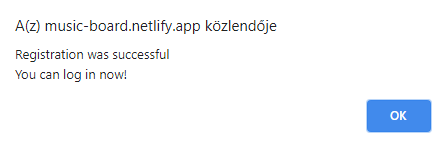
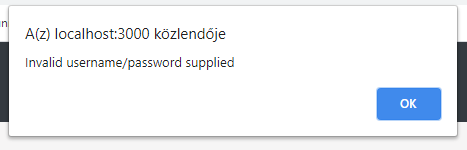
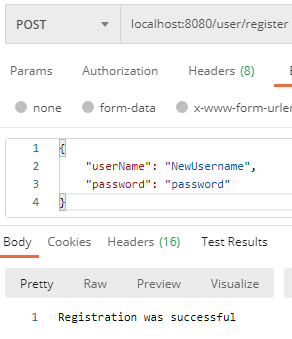
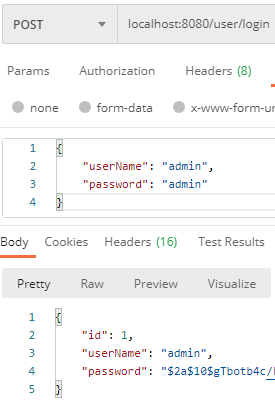
****



27. ábra Frontend alert ugyan ebben az esetben

26. ábra Már létező felhasználóval nem lehet újra regisztrálni

****



33. ábra Sikeres bejelentkezés visszaküldi a usert

30. ábra Sikeres regisztráció, üzenettel

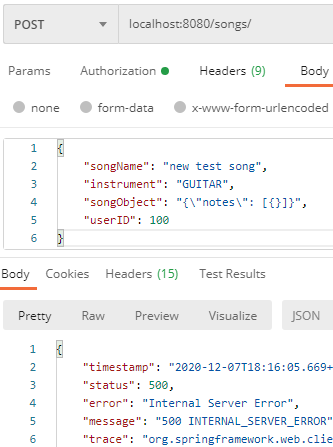
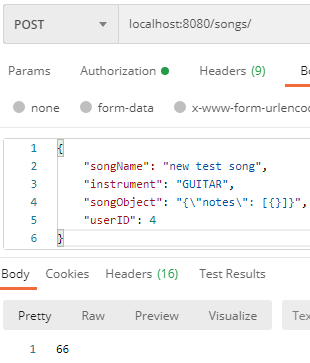
32. ábra Frontend hibás paraméterek való login próbálkozás esetén alertet dob

31. ábra Sikeres regisztrációt jelzi a frontend

34. ábra Hibás paraméterek esetén 401-es hiba

28. ábra A frontend nem enged regisztrálni, ha a szöveg hossza nem megfelelő

29. ábra Null value-val nem lehetséges a regisztráció

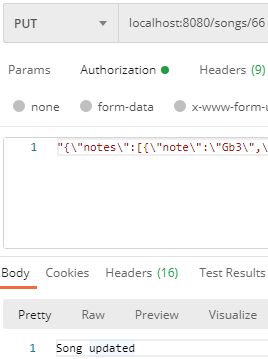
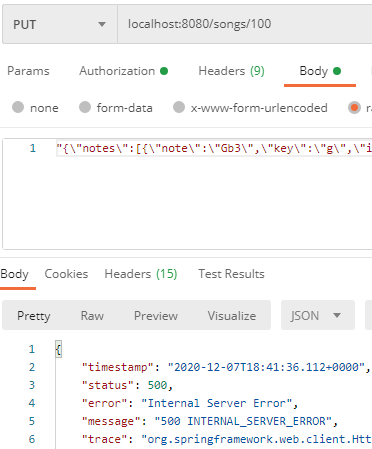
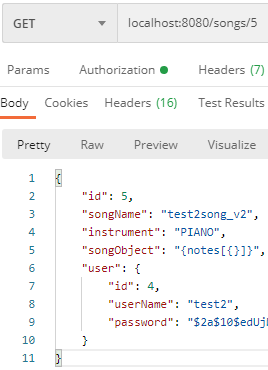
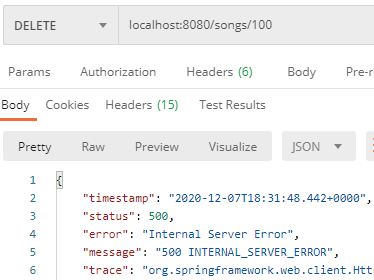


38. ábra A user dalai közé bekerült az újonnan hozzáadott szám

37. ábra Szám hozzáadása sikeres, visszaküldi az IDját

36. ábra Dal hozzáadása nem létező user IDval

35. ábra User dalainak lekérése ID alapján

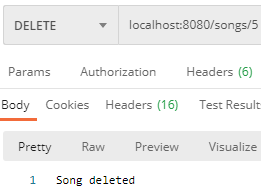
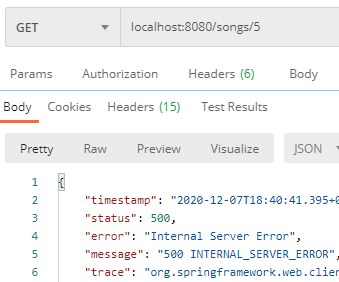


42. ábra Szám törlése nem létező user ID paraméterrel sikertelen

41. ábra Egy dal lekérése ID alapján

39. ábra Dal frissítése rossz user ID paraméterrel

40. ábra Dal frissítése sikeres



44. ábra A szám tényleg törlődött, az IDja már hibaüzenetet dob vissza

45. ábra A user dalainak listájáról is sikeresen eltűnt a szám törlés után

43. ábra Dal törlése sikeres

## Deployment

A backend Herokun él egy PostgreSQL adatbázissal, a frontend pedig Netlifyon. A Herokus deploy automatikusan generált egy git repositoryt, de a Netlifyhoz manuálisan hoztam létre egy új git repositoryt, ami csak az alkalmazás frontendjét tartalmazza. Ez az alábbi linken érhető el:

<https://github.com/gyugyipeter/music-board-frontend>

A deploy során akadt néhány probléma, ezért módosításokat kellett végeznem a projekten. A Spring Boot project application.properties file-ja tartalmazta az adatbázis-kapcsolódáshoz szükséges konfigurálást. A Heroku azonban a saját adatbázisát konfigurálás nélkül is magától hozzárendeli a projekthez, ezért ezeket eltávolítottam.

A frontendel sajnos több probléma is volt. Sok esetben a link egy 404-es site-ra irányított úgy, hogy elérhető volt az app. Ezért hozzáadtam a ’netlify.toml’ file-t a projekthez, ez redirectel az oldalra ilyen esetben.

A hang file-okat nagyon lassan töltötte be a lokális verzióhoz képest, ezért szükséges volt az összes audio file-t Howl objectté inicializálni a program elindulása után. Ezért az AppContext első lépésként végig iterál a file-okat tároló mapeken és meghívja rájuk a Howl konstruktorát. Most már csak pár másodpercet kell várni az oldal megnyitása után, onnantól tökéletesen használható.

Sajnos továbbra is fennálló probléma viszont, hogy lassú a kommunikáció a frontend és a backend között, az API requestek több időt vesznek igénybe a kelleténél.

# Konklúzió

A projekt elkészítése nagy kihívást jelentett számomra, mivel többnyire korábban általam nem ismert technológiákat használtam. Első sorban a célom az volt, hogy megtanuljam a legmodernebb, legelterjedtebb webtechnológiákat használni. Sok energiát kellett belefektetnem, de utólag nagyon megérte, rengeteget tanultam az applikáció elkészítés során.

Összességében a tervet teljesítettem, de a végére sokkal jobban is sikerült, mint ahogy a tervezéskor elképzeltem, és sok funkciót is hozzáadtam, amikre csak a fejlesztés során gondoltam.

Biztosan több ponton lehetett volna hatékonyabb a kód. Sajnos a tanulóprojektnek ez a hátránya, de így is többször visszaléptem egyet, refaktoráltam kódrészeket, újra gondoltam komponenseket stb., szóval jelenleg azt hiszem kihoztam a projektből, amit tudtam. Zenészként biztosan hasznát fogom venni az alkalmazásnak, programozóként pedig még rengeteget gyakorolhatok ezen a projekten.

# További fejlesztési lehetőségek

Szinte végtelen még a lehetőség ebben applikációban, nem is lenne értelme most konkrét feature-öket megfogalmaznom. Ha már csak arra gondolunk például, hogy még mennyi hangszert lehetne hozzáadni, egy-egy hangszerhez mennyi hangzást... A logger is rengeteg feature-el bővülhet még, lehetne okosabb a kezelőfelülete.

Első sorban a backendet lenne érdemes bővíteni, az autentikációt rendesen jwtokenekkel megcsinálni, a userek tárolható tulajdonságait bővíteni (pl. egyedi hangolások, billentyűkiosztások). A felvételkészítést újra gondolni, hogy lehessen egy felvételt folytatni, lehessen utólag az időzítéseken módosítani, lehessen metronomra felvételt készíteni, visszajátszáskor a tempón módosítani stb.

Tehát lényeg, hogy ezt semmiképp sem lehet egy kész alkalmazásnak nevezni, csak egy első verziónak.

# Forrásjegyzék

<https://dbdiagram.io/> - 2020.12.02.