Ceglédi Szakképzési Centrum Közgazdasági és Informatikai Technikum

Szoftverfejlesztő- és tesztelő (5 0613 12 03)

ZÁRÓDOLGOZAT

Hotpodtato – Amerikai football projekt

|  |  |
| --- | --- |
| Szabó Dániel | Készítette: |
| Témavezető | Varga Norbert |
|  | Monori Ferenc Dániel |

Cegléd, 2024

Tartalom

[1. Miért választottuk ezt a témát? 3](#_Toc164032237)

[2. A projekthez használt technológiák alkalmazások 3](#_Toc164032238)

[3. HTML leírása: 3](#_Toc164032239)

[4. CSS leírása és értelmezése: 4](#_Toc164032240)

[5. JavaScript leírása: 5](#_Toc164032241)

[6. A C# leírása: 6](#_Toc164032242)

[7. MySQL - relációs adatbázis 6](#_Toc164032243)

[8. XAMPP 7](#_Toc164032244)

[9. A weboldal bemutatása: 7](#_Toc164032245)

[9.1 Navigációs panel 7](#_Toc164032246)

[9.2 Főoldal 9](#_Toc164032247)

[9.3 Statisztikai oldalak 12](#_Toc164032248)

[9.3 Fantasy Ligák oldal 19](#_Toc164032249)

[9.4 Hírek oldal 22](#_Toc164032250)

[9.4.1 Háttere: 22](#_Toc164032251)

[9.5 Belépési felület 24](#_Toc164032252)

[10. Adatbázis felépítése 25](#_Toc164032253)

[11. API: 27](#_Toc164032254)

[11.1 API kérések: 27](#_Toc164032255)

[11.2 GET kérés: 28](#_Toc164032256)

[11.3 POST kérés: 29](#_Toc164032257)

[12. Kliens oldali programozás: 30](#_Toc164032258)

[12.1 Hírek oldal: 30](#_Toc164032259)

[12.2 Blogbejegyzések oldala: 32](#_Toc164032260)

[12.2.1 Blogbejegyzések megjelenítésének megvalósítása: 32](#_Toc164032261)

[13. Autentikáció - Hitelesítés 34](#_Toc164032262)

[14. Asztali alkalmazás 35](#_Toc164032263)

[15. Hibák 41](#_Toc164032264)

[16. Fejlesztési lehetőségek 42](#_Toc164032265)

[Felhasznált irodalmak 43](#_Toc164032266)

# Miért választottuk ezt a témát?

Ezt a projektet a Varga Norbert ötlete alapján készítettük, a cél az volt létrehozzunk Amerikai foci témájú Fantasy focis weboldalt, ahol a Fantasy football játékosok információkat kaphatnak az aktuális eredményekről.

Emellett szerettük volna, hogy az aktuális Amerikai football-al kapcsolatos összes fontosabb hírek, illetve történés is egy helyen elérhető legyen a felhasználók számára. Terveink szerint az oldal a későbbiek során élesben is beüzemelésre kerül majd.

# A projekthez használt technológiák alkalmazások

A weboldalt és a hozzátartozó asztali alkalmazást HTML, CSS, JavaScript és C# programnyelveken írtuk meg.

A weboldal programkód kódrészének elkészítéséhez a Microsoft Visual Studio Code-ot használtuk. Ennek a kódszerkesztési, és a terminal funkcióit vettük igénybe, azon belül is a node, bash, és powershell terminál típusokat. Ez a Git és a Node.js támogatás volt szükséges.

Az asztali alkalmazást szintén a Microsoft által üzemeltetett Visual Studio alkalmazásban készítettük el. A weboldal létrehozásához továbbá használtuk a Bootstrap, React, és Node.js keretrendszereket.

A projekt adatbázisának létrehozásához és teszteléséhez a XAMPP nevű szoftvercsomagot használtuk, amely tartalmazza az Apache HTTP szervert és a MySQL adatbázis-kezelő rendszert.

Fontos információ, hogy a költséghatékonysági okok miatt jelen projektünk is XAMPP és Node.js használatának segítségével futtatható.

# HTML leírása:

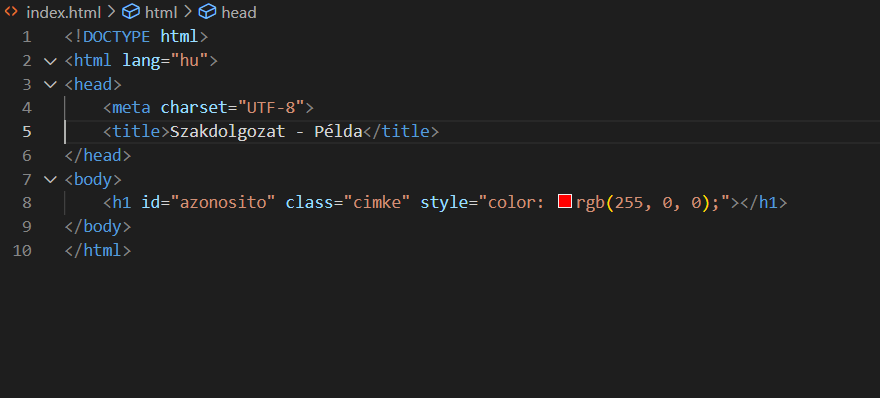
A HTML egy program-leíró nyelv ami teljes nevén annyit takar hogy Hyper Text Markup Language magyarul (hiperszöveges jelölőnyelv). Ezt a nyelvet weboldalak készítéséhez szokták használni.

A HTML felépítése kapcsán három fő részről beszélhetünk:

1. Dokumentum típus (<!DOCTYPE html>)
2. <head></head> tagek, ami a HTML dokumentációs tételeit jeleníti meg
3. <body></body> tagek amely a megjelenítendő adatokat tartalmazza.

Az *1.kép*en képen egy alap HTML dokumentumot lehet látni, aminek az első sora a dokumentum típusát adja meg, majd HTML tagen belül a dokumentum nyelve (”hu” vagyis magyar) van megadva.

A harmadik és hatodik sor között (head tagek) található: a negyedik sorban a karakterkódolás, ami esetünkben UTF-8, az ötödik sorban pedig oldal címe <title> tagek között, ami megadja, hogy mi legyen a böngésző ablak fülén megjelenő megnevezése az adott oldalnak.

**

*1.kép*

A hetedik és kilencedik sor között (body tagek) helyezkedik el: a HTML oldal tartalma, amit a böngésző jelenít meg. Ez esetben a <h1> tag, amiben egy szöveget lenne lehetőségünk elhelyezni.

A <h1> tag-nél látható egy id („azonosito” névvel) ami arra szolgál, hogy ennek az azonosítónak a megadásával tudjunk hivatkozni erre az elemre, ha szeretnénk azt a későbbiekben módosítani, vagy manipulálni.

Emellett megtalálható az elemnél egy class is (”cimke” névvel) ami egy osztályazonosító, és azt a célt szolgáltja, hogy előre definiált, egy osztályba rendezett (esetünkben ”cimke”) tulajdonságokkal vagy értékekkel tudjuk felruházni az adott <h1> tagben lévő elemet.

Továbbá megtalálható egy style attribútum is, aminek segítségével a HTML kódban tudjuk az adott elem stílusjegyeit állítani, jelen esetben ez az style attribútum a <h1> tagen belül lévő elem színét állítja pirosra.

# CSS leírása és értelmezése:

A CSS(Cascading Style Sheets) egy lépcsőzetes stílusleíró nyelv. Ennek használatával rengeteg lehetőség nyílik, hogy személyre szabhassuk egy weboldal stílusjegyeit.

Jellemzően a CSS stílus beállításokat egy külön .css file-ben tároljuk (bár ahogy a fenti példában látszik van lehetőség a HTML kódban való elhelyezésre is).

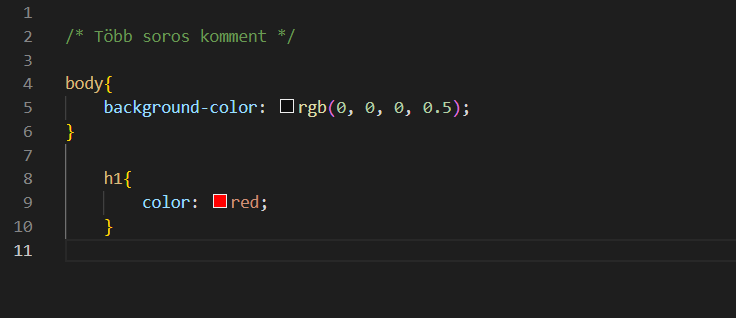
Emiatt a .css file-ban lévő beállítások érvényre juttatásához szükséges a stílus beállításokat tartalmazó .css file-t a HTML dokumentum számára hivatkozással elérhetővé tenni.

Ezt a HTML oldal <head> tag-jei között szükséges megtenni a *2.kép*en látható módon:



*2.kép*

Egy .css file a *3.kép*en látható módon épül fel:



*3.kép*

A fent látható kód második sorában egy komment látható, ami nem befolyásolja a beállításokat, csak a kód írójának egyfajta megjegyzése. Ez sok esetben nagyon hasznos, mert segíthet eligazodni egy akár több ezer soros fájlban.

A CSS fájlok szerkezete úgy épül fel, hogy megadjuk a stílusjegyekkel ellátni kívánt elem nevét, majd kapcsos zárójelek közé kerülnek a kívánt stílus beállítások.

A képen látható kód 4. és 10. sor között body tag-en belül a hátteret RGB opcióval feketére állítja egy 0.5 -es opacitivással(áttetszőség) ami egy szürke árnyalatot ad majd vissza, emellett a minden h1 -es taget betűszínét pirosra állítja.

# JavaScript leírása:

A JavaScript (rövidítve: JS) egy objektumorientált programozási nyelv amelyet webhez használunk. Egy prototípus-alapú scriptnyelv. A JS jellemző fájlkiterjesztése a (*.js*). A JavaScript futtatható böngészőn kívül is ami például lehet a node.js.

Amikor a böngésző weblapot tölt be, amelyben programkódokat talál akkor átadja azt a beépített értelmezőnek amely végrehajtja az utasításokat. Az ilyen programok megfelelőek a weblap manipulálásához. A weblap programozásnál kétféle programozási „módszert” tartunk számon.

1. Kliensoldali programozás
2. Szerveroldali programozás

Jelenesetben a JavaScript által nyújtott lehetőségek alapján, kliensoldali és szerveroldali programozást is használunk a projektben.

# A C# leírása:

A C# ejtsd(szíshárp) egy .NET rendszerhez írt objektumorientált programozási nyelv. A 2002 -ben megjelent Visual Studio fejlesztési környezetével jelent meg a nyelv is. Elődjének a C++ nyelvet lehet tekinteni.

A programozóknak, akik C és C++ -ban programoztak alkalmazásokat, a nyelv hosszú időciklusa hátrány volt. Így hát ki kellett fejleszteni egy gördülékenyebb nyelvet, ez lett C# amely kényelmesebb és gyorsabb.

A nyelv történetéhez tartozik egy botrány-legenda is, miszerint a Microsoftnak nem volt engedélye az operációsrendszer Sun – Microsystemstől származó Java függvényeivel kiegészíteni a nyelvet, így a Javát el kellett távolítani projektből. – a Javát továbbra is lehetett külön rendszerként telepíteni.

A C# és .NET keretrendszer alapja a Common Language Infrastructure(CLI).

# MySQL - relációs adatbázis

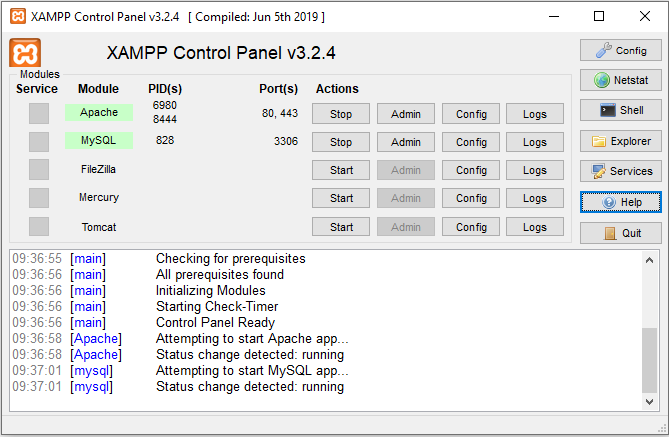
Relációs adatbázisnak nevezzük a relációs adatmodell elvén létrehozott adatok összességét. Reláció alatt azt értjük, hogy kapcsolati viszony jön létre az adott adatbázisban.

A MySQL egy többfelhasználós, többszálú, SQL-alapú relációs adatbázis-kezelő szerver.

A webalkalmazások futása alábbi séma szerint zajlik.

* A programkódokat a webszerver tárolja.
* Az kapcsolódás után a webszerverhez lekéri a programsorokat.
* A webszerver felveszi a kapcsolatot az alkalmazásszerverrel.
* Átadja a kódot amit végrehajt.
* A végrehajtás eredménye az lesz hogy a webalkalmazás – amely tartalmazza a űrlapokat, beviteli mezőket, hivatkozásokat és további programkódokat – biztosítja a felhasználó és az alkalmazás közötti interakciót.
* Az alkalmazásszerver átadja a weblapot a webszervernek.
* A webszerver elküldi a felhasználónak a weboldalt.
* Ha a felhasználó műveletet vagy adatokat gépel be a böngésző ismét a webszerverhez fordul és a programkód indítását kéri.
* A webszerver újra az alkalmazásszerverhez fordul és a lefutás után keletkező weboldalt küldi vissza a böngészőnek.

# XAMPP



*4.kép*

A *4.kép*en látható XAMPP egy nyílt forráskódú platformfüggetlen szoftvercsomag, amelyen tudunk Apache szervert futtatni.

Az Apache szerveren felül tudunk MySQL-t vagyis adatbázis kezelőt és FTP (file transfer protocolt) szervert futtatni.

Viszont nekünk csak az első kettőre volt szükségünk a projekt megvalósításához.

Hivatalosan a XAMPP tervezői az eszközt egy fejlesztőrendszernek szánták, amellyel a web-tervezők és programozók internetes kapcsolat nélkül fejleszthetik és tesztelhetik alkalmazásaikat. Ennek érdekében több fontos biztonsági funkció alapértelmezésben ki van kapcsolva a csomagban, ennek ellenére a XAMPP szoftvert valódi webes szolgáltatóként is használják.[[1]](#footnote-1)

# A weboldal bemutatása:

## 9.1 Navigációs panel



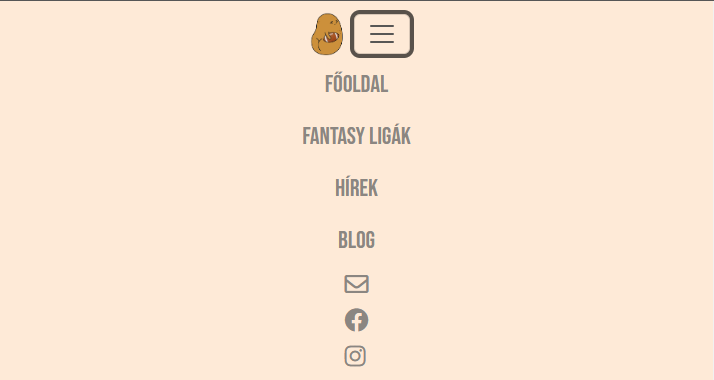
*5.kép*

Az *5.kép*en navigációs panel látható. Ez egységes megjelenéssel, és funkcionalitással megtalálható minden html oldal tetején. Szerepe, hogy a minden főmenüpont ugyanolyan módon legyen elérhető bármely html oldalon vagyunk aktuálisan.

Emellett bal oldalt tartalmazza az oldal logo-ját amire kattintva mindig a főoldalra juthatunk vissza.

Jobb oldalon lévő ikonok e-mail, illetve közösségi média eléréseket tartalmaznak. Ezekre az ikonokra kattintva az oldal későbbi üzemeltetőjének e-mail elérése mellett a Hot Podtato közösségi média eléréséit találjuk.

A navigációs panel reszponzívan működik, és adott képernyő méret alatt menü struktúrát vált a jobb láthatóság miatt, ez a 6.képen látható.



*6.kép*

Megvalósítását tekintve egy HTML <nav> tag került használatra, ami felruháztunk egy Boostrap-ból származó osztállyal, ami biztosítja, hogy adott méret alatt a <nav> tagben lévő elemek átrendeződjenek, és egy gomb segítségével legyenek kinyitatóak, illetve összecsukhatóak.

Ezek a *7.kép*en láthatóak HTML kód formájában.



*7.kép*

## 9.2 Főoldal

A főoldalon található a négy fő statisztikai kategória választó panel. Ez a kategória választó panel négy képpel került megvalósításra.

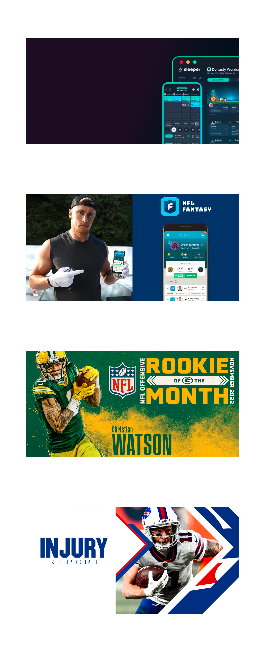
A képekre rámozgatva az egér mutatóját megjelenik az adott statisztikai kategória neve is.

Így van lehetőség tovább navigálni a Dynasty rangsor, a Redraft rangsor, a Rookie rangsor, és az Injury riport oldalakra ahogyan ez a *8.kép*en látható.



*8.kép*

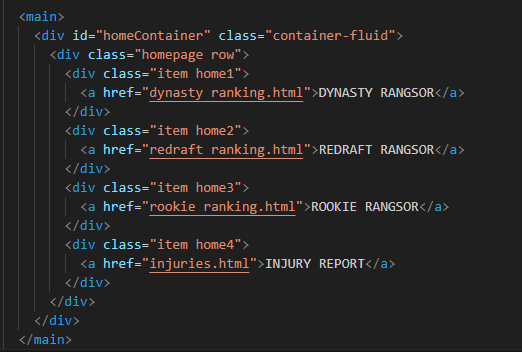
Az oldalon megjelenített tartalmak szintén reszponzív módon működnek, és a felbontás megváltozásával optimalizálódik a megjelenés is, ez a *9.kép*en megtekinthető.



*9.kép*

A megvalósítást tekintve a főoldalon egy div HTML elemben helyeztünk el egy újabb div elemet, ezen belül szintén div elemekben egy-egy hivatkozást helyeztünk el ami az adott html oldalra mutat, ahova szeretnénk navigálni.

Ezeket a div elemeket CSS stílusjegyekkel és Bootstrap osztályokkal láttuk el a kívánt nézet elérésének érdekében, ez a *10.kép*en látható.



*10.kép*

A CSS file-ban az adott div-ekben lévő elemekre vonatkozóan többek között a következő stílus beállításokat adtuk meg:

* az elem mérete
* a többi elemtől lévő belső távolsága
* az elemek elhelyezkedése
* a rendelkezésre álló terület kitöltésének mértéke
* az adott elemben megjelenítendő kép
* a megjelenített kép magasság/szélesség aránya
* a képen megjelenített kategória név megjelenítésére vonatkozó beállítások.

Ezek a *11.kép*en láthatóak:



*11.kép*

## 9.3 Statisztikai oldalak

A weboldalon statisztikai oldalak gyűjtőnévvel illetem azokat a HTML oldalakat, amik kinézetüket és megvalósításukat tekintve egyzenek, csak az a megjelenített adatokban mutatkozik eltérés.

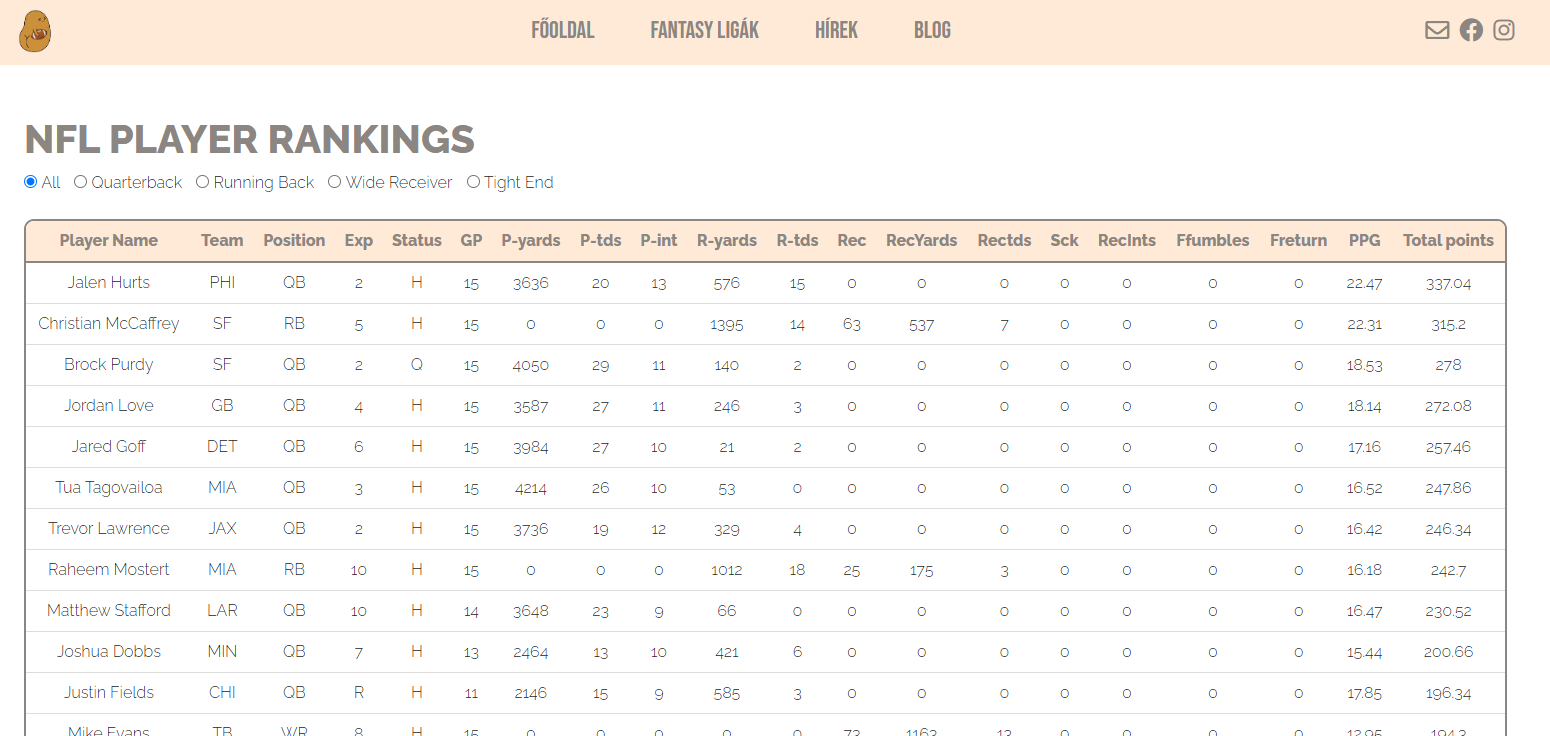
Ezek az oldalak a Dynasty rangsor, a Redraft rangsor, a Rookie rangsor, és az Injury riport oldalak. Ezeket az oldalakat a főoldalról navigálva érhetjük el.

Ezen az oldalon a Fantasy football-ban elérhető football játékosok adatait és eredményeit találhatjuk meg.

Mivel több fajta Fantasy football liga indítható el és ezek szabályrendszereiket tekintve különböznek, ezért statisztikai tekintetben is meg kell különböztetnünk ezeket a ligáékat.

Emiatt került kialakításra a Dynasty rangsor, a Redraft rangsor, a Rookie rangsor oldal külön-külön. Ehhez társul még egy Injury riport oldal is, ami minden ligára érvényes.

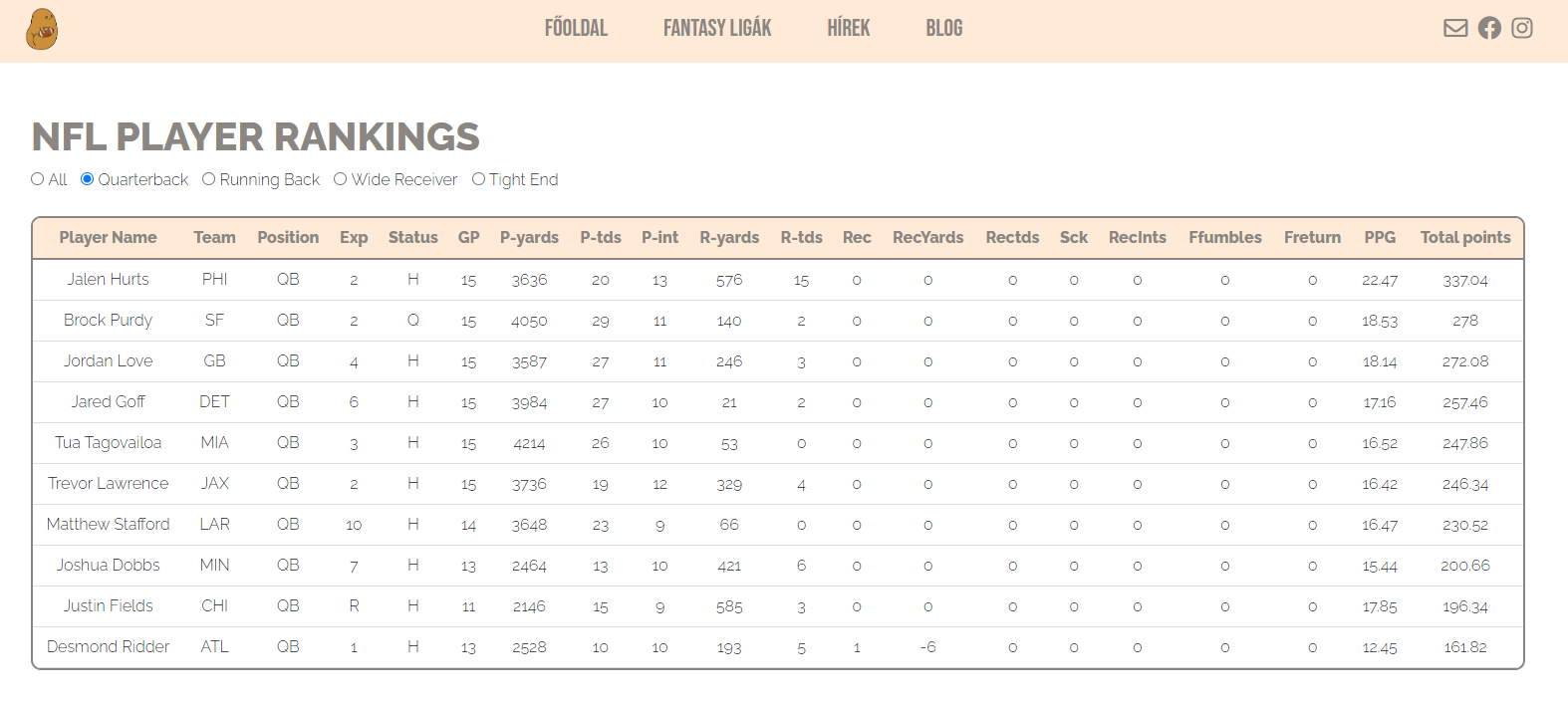
A statisztikai oldalak *12.kép*en látható módon kerülnek megjelenítésre.



*12.kép*

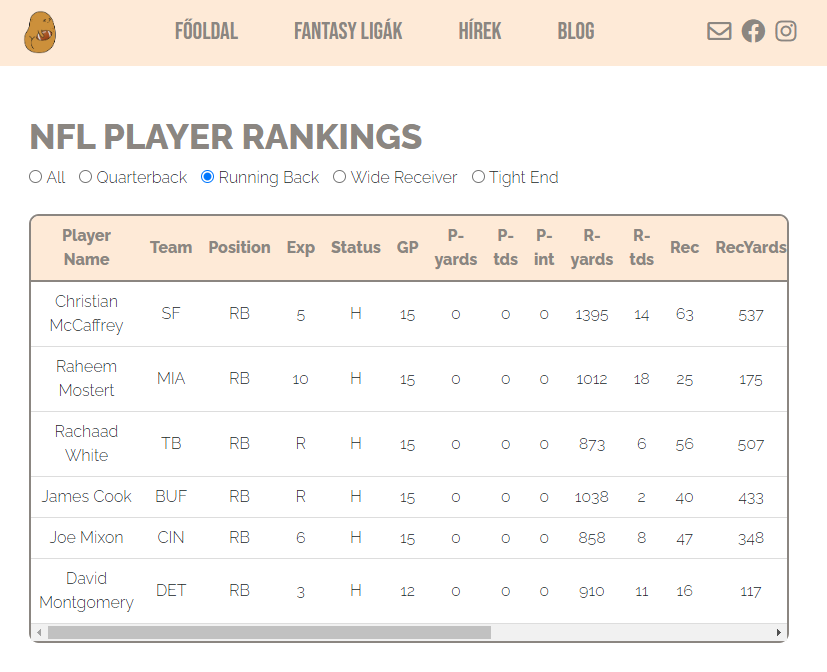
A statisztikai oldalakon egy táblázatban jelenítjük meg az amerikai football játékosok statisztikáit.

Ehhez tartozik egy tartalom szűrő, amivel lehetővé tesszük azt, hogy az összes, vagy csak az adott poszton szereplő játkosok statisztikái kerüljenek megjelenítésre, ahogy ez a *13.kép*en látható.



*13.kép*

Az oldal szintén reszponzív, és a táblázatok megjelenítését adott felbontás alatt úgy alakítja át, hogy az alsó sorba bekerül egy görgetősáv amivel a táblázat azon részei is megtekinthetőek amik a felbontás miatt nem megjeleníthetőek, ez a *14.kép*en látható.



*14.kép*

A statisztikai oldalak működésüket tekintve úgy épülnek fel, hogy egy üres HTML div elembe kerül javascript segítségével elkészítve belehelyezésre a táblázat, illetve a táblázatba az adatokbázisból API kapcsolat útján kinyert adatok.

Az adatok megjelenítésének szűrése szintén javascript segítségével valósul meg, olyan módon, hogy a HTML scriptben div-ekbe építve radio gombok kerültek elhelyezésre, amik kattintásra egy javascript függvényt futtatnak le.

A HTML oldal felépítése a *15. kép*en látható.



*15.kép*

A megjelenítést vezérlő javascript az oldal betöltése után azonnal lefut, azon belül is az AdatokLekereseAdatbazisbol nevű függvény, ami létrehoz egy üres adatok nevű tömböt, majd fetch függvény segítségével kikéri a megfelelő adatokat az adatbázisból.

Az adatbázisból való adatok visszaadását a server.js-ben get metódus segítségével kezeljük, de ami a *16.kép*en látható.



*16.kép*

Az adatbázis kapcsolatot felépítjük a kapcsolat változóban egy kapcsolat nevű változóban név nélküli nyílfüggvénnyel, createConnection használatával, megadjuk az elérést, felhasználót, jelszót, és az adatbázist amit szeretnénk elérni.

Ezután egy get metóduson belül kezeljük, hogy mit történtjen, ha egy adott kérés beérkezik a server felé (esetünkben a lépen ez a: ’/dynasty’ kérés).

Jelen ha megérkezik az adott kérés a server felé, akkor válaszként az adatbázis players nevű táblájából adjuk vissza az összes adatot, ahol a leaguetype Dynasty.

Kezeljük, hogy hiba esetén mi érkezzen vissza, illetve megadjunk, hogy válaszként küldje ki az eredményét a lekérdezésnek.

Ezután a kapott válaszban megtalálható adatokon egy foreach függvény segítségével végig lépdel, és az adatokat feltölti a korábban létrehozott adatok tömbbe. Ezután meghívja az AdatokBetolteseTablaba függvény az adatok paraméterrel.

Ez a *17.kép*en látható.



*17.kép*

A *18.kép*en látható, hogy ezután lefut az AdatokBetolteseTablaba függvény ami első lépésként document.GetElmentById segítségével kapcsolatot hoz létre a ”tablaDiv” nevű elemmel, ami a korábban már említett üres div elem a HTML oldalon.

Ezután document.createElement segítségével egy table elemet hoz létre, majd meghívja a *19.kép*en látható FejlecKeszites nevű függvényt a paraméterként a table elemmel, ami a táblázatunk fejlécének elkészítéséért felel.

Tovább futva elindít egy for ciklust, végig megy az adatok tömb elemein, minden lefutásnál készít egy sort, és a *20.kép*en látható CellatASorba függvény segítségével elhelyezi ebbe a sorba a megfelelő oszlopokat.

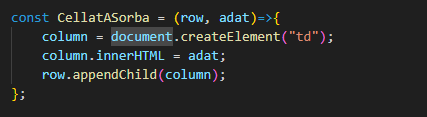
Ezután appendChild segítségével hozzáfűzi a table elemhez ezt.



*18.kép*



*19.kép*



*20.kép*

A FejlecKeszites nevű függvény paraméterként a table elemmel createElement segítségével készít egy sort, amely sorba a CellatASorba függvény segítségével oszloponként beleteszi a paraméterként megadott szövegeket, majd appendChild segítségével hozzáfűzi a table elemhez ezt, ezzel kialakítva a táblázat fejlécét.

A CellatASorba nevű függvény createElement segítségével létrehoz egy oszlopot majd ebben az oszlopban elhelyezi innerHTML segítségével a paraméterként megadott (adat) szöveget.

Ezután appendChild segítségével hozzáfűzi a paraméterként megadott (row) sorhoz, az oszlopot annak tartalmával.

Az adatok lekérése, és megjelenítése a fentiek szerint az oldal betöltésekor lefutó javascript AdatokLekereseAdatbazisbol függvényével történik meg.

Ezután a megjelenítés szűrése úgy történik. hogy a megfelelő radio gombra való kattintáskor lefut egy újabb függvény, ami ismét lekéri adatbázisból az adatok, és újra rajzolja a megjelenítést.

Ilyen esetben biztonsági okok miatt, illetve annak érdekében, hogy törekedjünk a kevesebb adatforgalmazásra, már az adatbázisból való kikérésnél is csak a számunkra releváns adatokat kapjuk vissza.

A *21.kép*en látható, hogy a Quaterback nevű radiogomb megnyomására lefutó positionQb nevű függvény miként épül fel.



*21.kép*

A működése ugyanazon az elven alapszik, mint amit fentebb már kifejtettem az összes adat kikérése kapcsán.

A különbség abban van, hogy a fetch függvénnyel egy másik kérést indítunk, amire egy másik választ is kapunk a servertől, és ebben a válaszban már csak az általunk kért adatok szerepelnek nem pedig az összes adat, így már csak ez kerül újra rajzolásra és megjelenítésre.

## Fantasy Ligák oldal

A weboldal fantasy ligák oldalán a fantasy footballt játszó emberek eredményeinek, és csapataik statisztikáinak megjelenítése a cél.

Az oldalon a megbízói kérésnek megfelelve négy liga kerül megjelenítésre. Jelen állás szerint ez a négy liga nem is fog változni a jövőben sem.

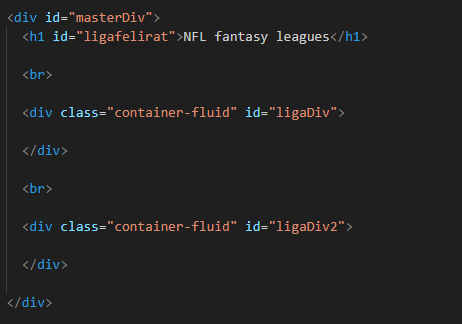
Megjelenésében ez az oldal is illeszkedik a weboldal témavilágába ez látható a *22.kép*en, illetve hasonlóképp reszponzívan működik.



*22.kép*

Az adatok megjelenítése itt is javascript segítségével történik. Hasonlóképp az előzőkhez itt is egy üres div elemek kerülnek a HTML oldalba, és ezekben a div elemekbe kerül behelyezésre a táblázat, és az adatok.

A HTML oldal a *23.kép*en látható módon épül fel.



*23.kép*

A megjelenítést vezérlő javascript az oldal betöltése után azonnal lefut, azon belül is az AdatokLekereseAdatbazisbol1,-2,-3,-4 nevű függvények.

Mivel itt megbízói kérésre a négy táblának teljesen el kell különülnie egymástól, hasonló módon mint az előzőekben, de itt négy külön függvénnyel kérjük ki az adatokat, ez látható a *24.kép*en.

Az AdatokLekereseAdatbazisbol1,-2,-3,-4 nevű függvények működése megegyezik az előzőleg adatbázis lekérésre használt függvényekével, különbség abban van, hogy a fetch függvénnyel egy másik kérést indítunk, amire egy másik választ is kapunk a servertől.

Ezután az AdatokBetolteseTablaba ill. AdatokBetolteseTablaba2 az függvény kerül meghívásra attól függően, hogy a ligaDiv vagy a ligaDiv2 azonosítójú div elembe kerülnek -e majd az adatok.

Az AdatokBetolteseTablaba ill. AdatokBetolteseTablaba2 függvények futásakor első lépésként document.getElementById segítségével kapcsolatot hoznak létre a a ligaDiv vagy a ligaDiv2 azonosítójú div elemmel.

Majd document.createElement segítségével egy új div elemet hoznak létre amit ellátnak egy „leagueDiv” azonosítóval és egy Bootstrap osztállyal ami a tábla reszponzivitását biztosítja.

Következő lépésként egy újabb div elemet hoznak létre ezt ellátják egy „leaguefl” azonosítóval majd appendChild segítségével csatolják az előzőleg létrehozott div elemhez.

Ezután egy sortörést hoznak létre createElement segítségével majd appendChild segítségével csatolják ez az előzőleg létrehozott div elemhez.

Majd a „leaguefl” azonosítójú elemben innerHTML segítségével elhelyezik az adott liga nevét, és appendChild segítségével csatlakoztatják a „leagueDiv” azonosítójú elemhez.

A következő lépésben a függvény createElement segítségével létrehoz két táblát egyet a fejlécnek, és egyet az adatoknak, és ezekhez azonosítót is rendel „flLigaTabla” és „ligaTabla” névvel.

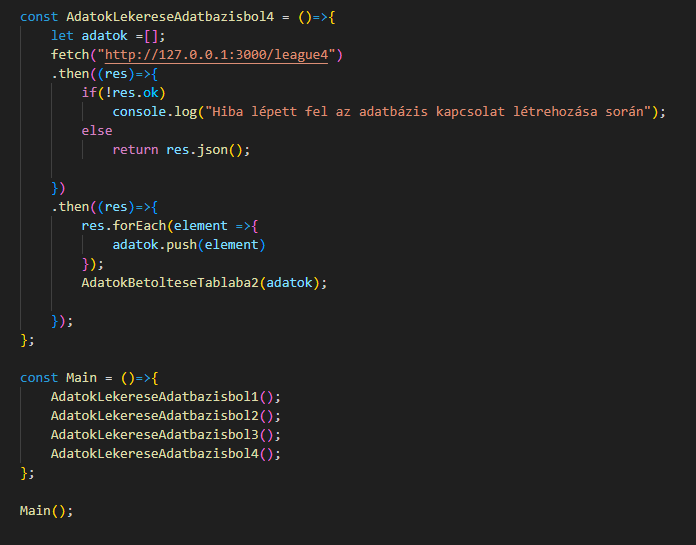
Ezután ismét egy div létrehozása következik, amit „leaguefl2” azonosítóval lát el majd csatlakoztatja appendChild segítségével az előzőleg létrehozott div elemhez.

Majd ismét egy div létrehozása következik, amit „leaguetblDiv” azonosítóval lát el majd csatlakoztatja appendChild segítségével a „leagueDiv” azonosítójú div elemhez.

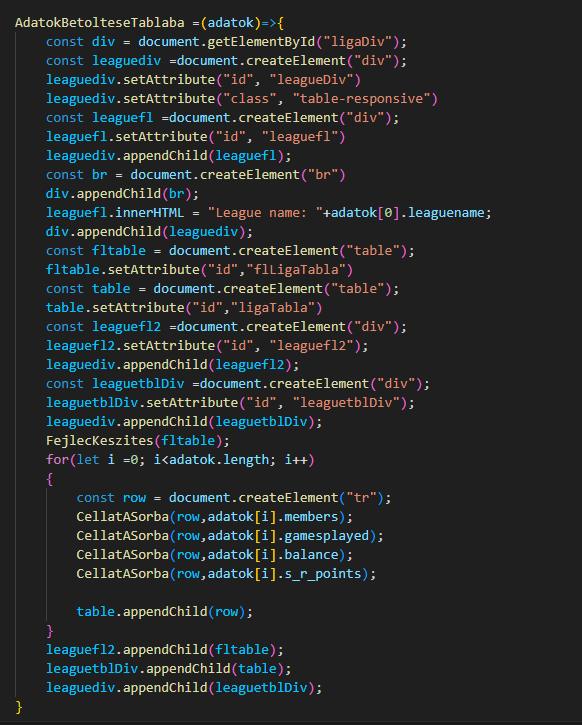
Ezután a már előzőleg ismertetett FejlecKeszites függvény kerül meghívásra paraméterként megadva az „flLigaTabla” azonosítójú table elemmel, ami lefutva elkészíti az aktuális tábla fejlécét.

Következő lépésben egy for ciklus segítségével végig megyünk az előzőleg AdatokLekereseAdatbazisbol1,-2,-3,-4 nevű függvények által feltöltött adatok nevű tömbön, minden lefutásnál createElement segítségével egy új sort hozunk létre, majd ezekbe a sorokban a már előzőleg ismertetett CellatASorba nevű függvény segítségével elhelyezzük a megfelelő oszlopokat a megfelel adattartalommal.

Ezután pedig appendChild segítségével a sorokat (row) az aktuális táblázathoz, az „flLigaTabla” azonosítójú elemet a „leaguefl2” azonosítójú elemhez, a „ligaTabla” azonosítójú elemet a „leaguetblDiv” azonosítójú elemhez, és végül a „leaguetblDiv” elemet a „leagueDiv” azonosítójú elemhez csatoljuk, ez a *25.kép*en létható.



*24.kép*



*25.kép*

## 9.4 Hírek oldal

## 9.4.1 Háttere:

A HTML része úgy lett kiépítve hogy *a .blog-page* -en belül egy *.container* majd egy *.row* osztályazonosítókba lettek pakolva, azután egy *.blog-post* class-ba kerül a tartalom.

A stílus amely a style.css -állományban van elmentve ami a *26.kép*en látható módon néz ki

.blog-page {

    color: rgba(0, 0, 0);

    padding: 30px;

}

.blog-post {

    background-color: rgba(254, 234, 215, 1);

    min-height: auto;

    margin: 20px auto;

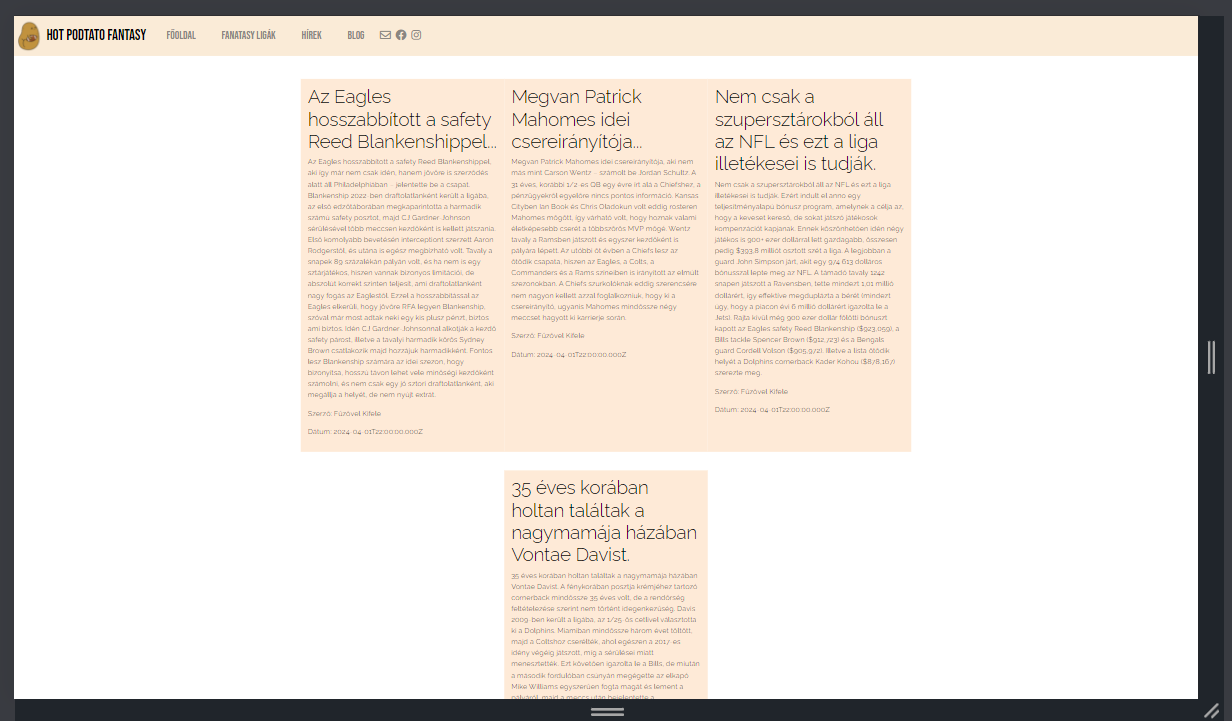
    padding: 15px;

    display: inline-block;

}

*26.kép*

Teljes mértékben reszponzív és Mobile L tehát 425px széles képernyőn egymás alatt és középre igazítva jelenik meg a blogbejegyzés kártyái. Az *27.kép*en képen tablet képernyőméret feletti felbontás megjelenítése látható.



*27.kép*

A képen szöveg, képernyőkép, dokumentum, Betűtípus látható

Automatikusan generált leírás

*28.kép*

A *28.kép*en, megjelenített tartalom egy mobilnézet felbontásban látható. – A kártyák szövege egy Lorem Ipsum általános szöveggel van kitöltve az átláthatóság miatt.

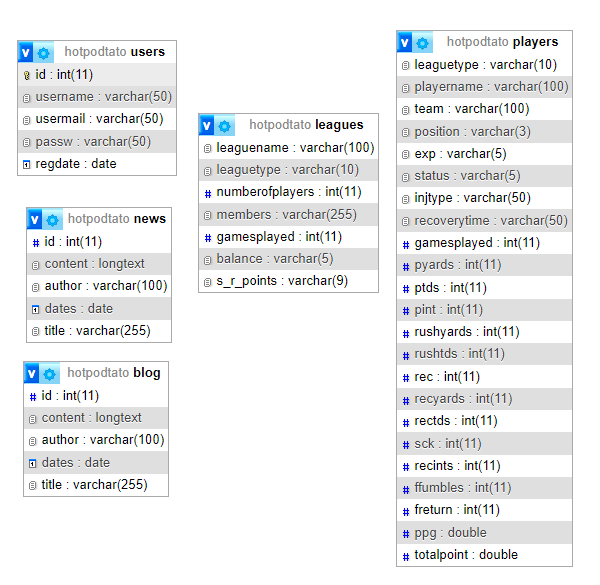
*A Lorem Ipsum amit már az 1500-as évek óta standard szövegrészletként szolgált az iparban; mikor egy ismeretlen nyomdász összeállította a betűkészletét és egy példa-könyvet vagy szöveget nyomott papírra, ezt használta.*

# 9.5 Belépési felület

A bejelentkezési felület részei:

1. H2-es címsor.
2. Vonalas elválasztó (HR)
3. Email label
4. Email input mező + placeholderrel kiegészítve ami az [email@examples.com] jeleníti meg
5. Jelszó label
6. Password(jelszó) input
7. Checkbox jelölőnégyzet és szöveg hozzá
8. Bejelentkezés button(gomb)

# Adatbázis felépítése



*29.kép*

A *29.kép*en hotpodtato látható adatbázisban öt tábla található.

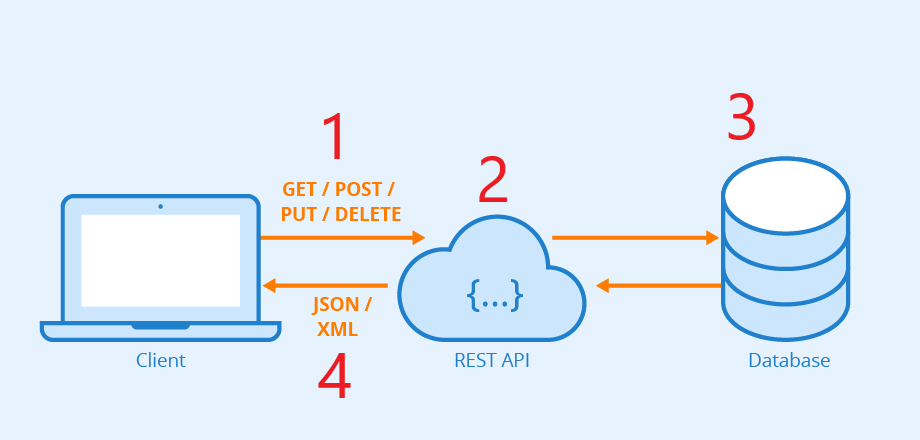
* 1. users – Autentikációhoz szükséges
  + id – INT típusú
  + username – VARCHAR típusú és 50 karakter fér bele
  + usermail – VARCHAR típusú és 50 karakter fér bele
  + passw – VARCHAR típusú és 50 karakter fér bele
  + regdate – DATE típusú
* 2. news – Hírek tartalma
  + id – INT típusú
  + content – Longtext (4,294,967,295 karakter fér bele) – Tartalom törzse
  + author – VARCHAR(100) szerző neve kerül bele
  + dates – DÁTUM(DATE) típusú
  + title – VARCHAR(255 karakter) típusú a bejegyzés címeinek
* 3. blog – Blog tartalma / 3. Hírek tartalma
  + id – INT típusú
  + content – Longtext (4,294,967,295 karakter fér bele) – Tartalom törzse
  + author – VARCHAR(100) szerző neve kerül bele
  + dates – DÁTUM(DATE) típusú
  + title – VARCHAR(255 karakter) típusú a bejegyzés címeinek
* 4. leagues – A fantasy ligák oldal adatait tartalmazza
  + leaguename – VARCHAR(100) a liga nevét tartalmazza
  + leaguetype – VARCHAR(10) a liga típusát tartalmazza
  + memberofplayers – INT a résztvevők számát tartalmazza
  + members – VARCHAR(255) a résztvevők nevét tartalmazza
  + gamesplayed – INT a lejátszott fordulók számát tartalmazza
  + balance - VARCHAR(5) a győzelem-vereség mutatót tartalmazza
  + s\_r\_points - VARCHAR(9) összes szerzett és kapott pontokat tartalmazza
* 5. players – Az amerikai football játékosok statisztikáit tartalmazza
  + leaguetype – VARCHAR(10) a liga típusát tartalmazza
  + leaguename – VARCHAR(100) a liga nevét tartalmazza
  + team – VARCHAR(100) a játékos csapatának nevét tartalmazza
  + position – VARCHAR(3) a játékos pozícióját tartalmazza
  + exp – VARCHAR(5) a játékos ligában eltöltött éveinek számát tartalmazza
  + status – VARCHAR(5) a játékos egészségügyi státuszát tartalmazza
  + injtype – VARCHAR(50) a játékos sérülésének típusát tartalmazza
  + recoverytime – VARCHAR(50) a játékos felépülési idejét tartalmazza
  + gamesplayed – INT a játékos által lejátszott mérkőzések számát tartalmazza
  + pyards INT – Amerikai football statisztika
  + ptds INT – Amerikai football statisztika
  + pint INT – Amerikai football statisztika
  + rushyards INT – Amerikai football statisztika
  + rushtds INT – Amerikai football statisztika
  + rec INT – Amerikai football statisztika
  + recyards INT – Amerikai football statisztika
  + rectds INT – Amerikai football statisztika
  + sck INT – Amerikai football statisztika
  + recints INT – Amerikai football statisztika
  + ffumbles INT – Amerikai football statisztika
  + freturn INT – Amerikai football statisztika
  + ppg DOUBLE – A mérkőzésenként áltagosan elért pontot tartalmazza
  + totalpoint DOUBLE – Az összesen elért pontot tartalmazza

A táblák külön tartalmazzák a működéshez szükséges adatokat, így a táblák összekapcsolására nem volt szükség a működéshez.

Mindemellett a táblák közötti kapcsolatok létrehozása tudná optimalizálni az adatbázis működését ezért erre a megoldásra jövőbeli fejlesztési lehetőségként tekintünk.

# API:

Az API rövidítése: Application Programming Interface. Jelentése: olyan alkalmazás programozási interfész vagy felület, ahol szoftverkomponenseket, lehet összekapcsolni. Biztosítja az adatok megjelenítését, vagyis lekérését, módosítását, illetve funkciókat lehet lehívni velük. A 4. ábrán az API kérések folyamatát lehet látni. Jól látható hogy a kérést elküldi a felhasználó, ezt követően az API értelmezi a kérést és az adatbázishoz fordul, majd ha sikeres a kérés akkor visszaadja az adatokat a kérő kliensnek.



*30.kép. Forrás: https://testerlab.io/blog/hogyan-m%C5%B1k%C3%B6dik-egy-rest-api/*

# 11.1 API kérések:

Mielőtt az API kérésekbe belemennénk ismertetni szeretném az API „inicializálását” ami így néz ki:

const fs = require('fs');

const express = require('express');

const app = express();

const bodyParser = require('body-parser');

const cors = require('cors');

app.use(cors());

// ADATBÁZIS LEKÉRDEZÉS

const mysql = require('mysql');

const kapcsolat = () => {

    return mysql.createConnection({

        host: 'localhost',

        user: 'root',

        password: '',

        database: 'hotpodtato'

    });

};

Az első és 5. sor között az API kellékei sorolódnak fel. Az első a fájl-system. A második, express keretrendszer. A harmadik sor az express kiszervezése egy változóba. A bodyParser egy modul az express csomagból amivel JSON, puffer, karakterláncok és URL adatokat tudunk „elemezni”. A cors pedig szintén egy modulcsomag.

Ezután következik az adatbázis(továbbiakban: SQL) inicializálás ami a következőképpen épül fel.

|  |  |  |
| --- | --- | --- |
| 13. sor - | megnyitja a kapcsolatot ami egy függvény, jelen esetben. |  |
| 14. sor - | return-el visszaadjuk és elindítjuk a „createConnectionnal” beépített függvénnyel a sql-kapcsolatot |  |
| 15. sor - | megadjuk az mySQL hosztot ami jelen pillanatban a localhost(helyi szerveren történik) |  |
| 16. sor - | megadjuk a SQL felhasználónevet ami a „root” a root felhasználó alapértelmezett esetben teljeskörű jogosultságokat ad. |  |
| 17. sor - | jelszó ami a root felhasználóhoz tartozik azért üres mert a rootnak alapértelmezetten nincs beállítva jelszó localhoston. |  |
| 18. sor - | végül megadjuk az adatbázis nevét ami a projektnév alapján „hotpodtato” |  |
| 19. és 18. sor - | lezárjuk a függvényeket |  |

## 11.2 GET kérés:

A GET-kérés felel az adatok lekérdezéséért. Az alábbi képen két GET kérés látható az első egy általános, a második egy fejlettebb kérés ami annyival különbözik az elsőtől hogy az SQL-ben „id” alapú lekérés történik.

app.get('/news/', (req,res)=>{

    const newNews = req.body;

    const connection = kapcsolat();

    connection.connect();

    connection.query('SELECT \* FROM news', (error, result, field) =>{

        if(error){

            res.send({error: "Hiba lépett fel a lekérés során."});

        }

        else{

            res.setHeader('Access-Control-Allow-Origin','\*');

            res.send(result);

        }

    });

    connection.end();

})

app.get('/news/:id', (req,res)=>{

    const connection = kapcsolat();

    connection.connect();

    connection.query('SELECT \* FROM news WHERE id=' + req.params.id, (error, result, field) =>{

        if(error){

            res.send({error: "Hiba lépett fel a lekérés során."});

        }

        else{

            res.setHeader('Access-Control-Allow-Origin','\*');

            res.send(result);

        }

    });

    connection.end();

})

Részeletessebben a GET kérésekről a fenti példákból. Az „app” az expresst hívja meg és ez után következik a beépített kérés függvénye ami jelen esetben a

„ *.get* ” ezt követően az API-linkje a következők pedig a nyílfüggvényben tartalmazza a requestet és a res-t mint választ. – Amennyiben beírjuk a böngészőben az API-szerver futtatása után a localhost+link tagját pl.: (*http://localhost:3000/news/*) ez esetben látnunk kell a helyi szerveren futó SQL-ből kinyert adatokat.

## 11.3 POST kérés:

app.post('/blog-post',bodyParser.json(),(req, res)=>{

    const newBlog = {

        id : req.body.id,

        content: req.body.content,

        author: req.body.author,

        dates: req.body.dates,

        title: req.body.title

    }

    const connection = kapcsolat();

    connection.connect();

    connection.query(`INSERT INTO blog(id,content,author,dates,title) VALUES (${newBlog.id},"${newBlog.content}", "${newBlog.author}","${newBlog.dates}","${newBlog.title}" );`, (error,result,field)=>{

        if(error){

            res.send({"error": "Hiba lépett fel a lekérés során.",

            "error2":error.message});

        }

        else{

            res.setHeader('Access-Control-Allow-Origin','\*');

            res.send(newBlog);

        }

    });

    connection.end();

})

Ez a PUT kérés annyiban tér el a GET kéréstől hogy egy objektumot hozunk létre és abba rakjuk a nekünk szükséges adatokat amiket a kérés testének részeivel tesszük egyenlővé. Emellett az SQL lekérdezés szintén módosul a beszúrás érdekében INSERT INTO-ra, majd az objektum testének paramétereivel.

# Kliens oldali programozás:

## 12.1 Hírek oldal:

Először is kimentettem az API URL-jét egy konstans változóba. Hogy később tudjak rá hivatkozni.

const apiUrl = 'http://localhost:3000/news';

Ezt követően egy aszinkron függvényt definiáltam és két lépésből áll. A fetch-el HTTP kérést indít az API URL-je felé, majd ha megérkezik a válasz vagyis a response, átalakítja JSON formátummá. Az visszatérési érték az adatok objektuma, ha bármilyen hiba történik a catch fogja lekezelni és egy hibaüzenetet ír ki a böngésző konzoljára.

Említésre méltó hogy "async/await" szintaktikát használja ami modern módszer és megkönnyíti és könnyebben kezelhető vagy olvasható a kód.

A blogbejegyzés megjelenítését szintén egy aszinkron függvénnyel oldottam meg.

Definiáltam két konstans változóban a HTML azonosítót és az előző aszinkron függvényt.

Majd elágazásban raktam, és feltételnek megadtam ha a megjelenítendő bejegyzések száma nagyobb mint nulla – tehát minimum egy bejegyzés található akkor készítsen egy html DIV elemet majd ezt egészítse ki egy „.blog-post” osztályazonosítóval és további Bootstrap keretrendszer osztályokkal.

Ezután ebben a HTML tagbe feltöltöttem innerHTML-el az SQL-ből szedett adatokat és így megkaptunk egy blogbejegyzést. Mivel ez egy forEach kezelte, ezt addig csinálja ameddig az adatbázisban bejegyzéseket talál és kilistázza az összes bejegyzést.

const postElement = document.createElement('div');

postElement.setAttribute("class","blog-post col-lg-4 col-md-4 col-sm-4");

          postElement.innerHTML = `

            <h1>${post.title}</h1>

            <p>${post.content}</p>

            <p>Szerző: ${post.author}</p>

            <p>Dátum: ${post.dates}</p>

          `;

## 12.2 Blogbejegyzések oldala:

|  |  |
| --- | --- |
| A Blog oldalát React(ReactJS) keretrendszerben készítettem el, a design vagyis kinézete nem különbözik a Hírek oldaltól mert ugyanazokat a CSS osztályazonosítókat használtam. – Így kinézetre nem különbözik, mégis teljesen más megközelítést alkalmaztam. – A React egy ingyenes és nyílt forráskódú Front-end keretendszer, főként felhasználói felületek létrehozására használják. A React előnye hogy csak az oldal megváltozott részeit jeleníti meg, míg a változatlan részekkel nem foglalkozik. A kódok komponensekre oszlik és azok entitásából áll. További előnye hogy a komponenseket többször is felhasználhatjuk.  A kivitelezés a ReactJS adta lehetőségével a tartalmakat egy „root” divben jeleníti meg. A „public” mappában találhatók az oldalhoz kapcsolódó állományok. Míg az „src” mappában vannak a fejlesztéshez szükséges fájlok. Az App.js-ben dolgozunk a fejlesztések folyamán, itt importáljuk a CSS és JS-eket és a komponenseket is. A rendezettség érdekében Én a komponenst egy „Components” mappába helyeztem BlogPosts.JS néven. | A képen szöveg, képernyőkép, szoftver, Multimédiás szoftver látható  Automatikusan generált leírás |

## 12.2.1 Blogbejegyzések megjelenítésének megvalósítása:

Első lépésként beimportáltam a használni kívánt „modult”.

import React, { useState, useEffect } from 'react';

Delaráltam egy konstans változóként egy BlogPosts nyílfüggvénnyel hivatkoztam az API URL-jére fetch függvénnyel ami jelen esetben:

fetch('http://localhost:3000/blog/')

Amennyiben hiba történne a lekérés során a catch függvény a böngésző konzolára kiír egy hibaüzenetet.

Végül a függvény return() részében visszaadatom a blogbejegyzés tartalmát esztétikusan formázva.

  return (

    <div className='blog-page'>

      <div className='container'>

        <div className='row'>

          {data.map(item => (

            <div className='blog-post col-lg-4 col-md-4 col-sm-4' key={item.id}>

              <h1>{item.title}</h1>

              <p>{item.content}</p>

              <p>Szerző: {item.author}</p>

              <p>Dátum: {item.dates}</p>

            </div>

          ))}

        </div>

      </div>

    </div>

  )

A fenti kódban a „map” függvénnyel végig megyek a tömb elemein és egy kulcsot rendelek a divhez. Majd mint a Hírek oldalnál, itt is hivatkozok a formázás keretein belül az esztétikus megjelenésre.

Utolsó lépésként ezt a BlogPosts komponenst meghívva az App.js állományban; a Visual Studio Code terminálon futtatom a ***npm run build***parancsot és a rendszer kigenerál egy build mappában minden fájlt ami szükséges a megjelenítéshez. Ezt a build mappát és benne lévő állományait áthelyeztem az oldal többi részéhez. Majd a menüben hivatkoztunk rá. Így lényegében elkészült a Blog oldala.

# Autentikáció - Hitelesítés

A képen szöveg, képernyőkép, tervezés látható

Automatikusan generált leírás

A belépési felülethez tartozó JS állományban deklaráltam egy üres függvényt és azon belül végeztem el a renderelést. Létrehoztam egy konstans változót loginForm néven és lekértem az azonosítóját (ID). Ezután erre a konstansra hivatkozva addEventListener függvénnyel elindítottam egy aszinkron metódust. Deklaráltam egy usermail és password változókat amikkel kikértem a Formban lévő username és password ID értékét „ .value ” -kkal.

      const usermail = document.getElementById('username').value;

      const password = document.getElementById('password').value;

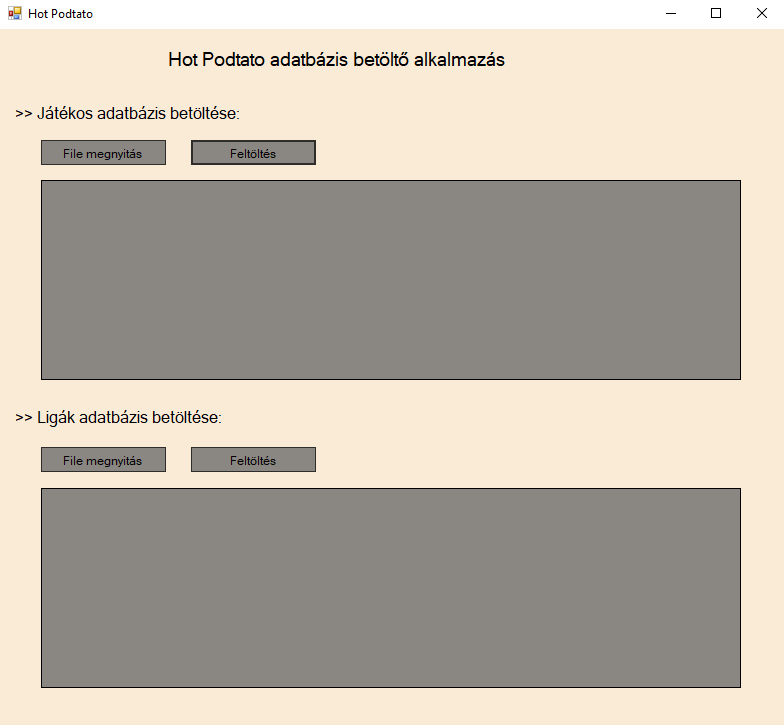
A try{ } részben hivatkozok az API-ra, majd response body-jára ami JSON-ra alakítja a tulajdonságokat és így hozzáfér az adatbázis „usermail” és „passw” mezőire. Ezután egy elágazás következik amennyiben a válasz rendben van átirányít a megadott paraméterre. Különben hibaüzenetet dob fel a felhasználónak.

# Asztali alkalmazás

A weboldalhoz egy asztali alkalmazás is készült C# programnyelvvel. Az alkalmazás célja, hogy API kapcsolat útján egy fájlból beolvasva az adatok megfelelően rendszerezésre kerüljenek.

Fontos szempont volt az is, hogy az adatbázisba való feltöltés előtt az adatok láthatóak, ellenőrizhetőek, és szükség esetén szerkeszthetőek legyenek.

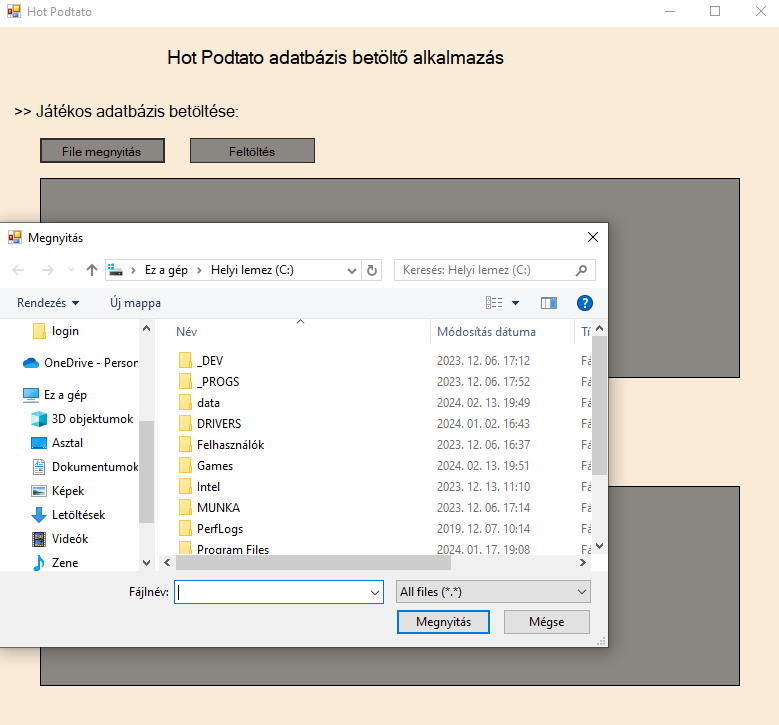
Az alkalmazás illeszkedik a webes felületen is használt témavilágba, ez a *41.kép*en látható



*41.kép*

Az alkalmazás alapvetően két adatbázis tábla feltöltését teszi lehetővé elkülönítve, a players tábláét, és a leagues tábláét.

A File megnyitás gomb megnyomásával betallózhatjuk a saját gépünkről a file-t, amiben a feltöltendő adatok vannak. Ez a *42.kép*en látható

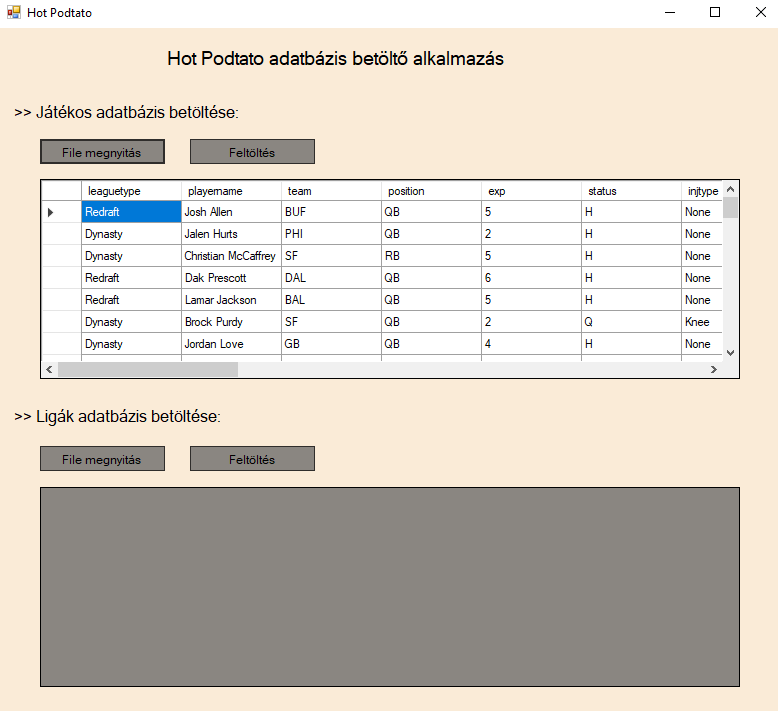


*42.kép*

A betöltendő file-nál az az elvárás, hogy az adatok vesszővel elválasztóvá, soronként rendezve legyenek benne.

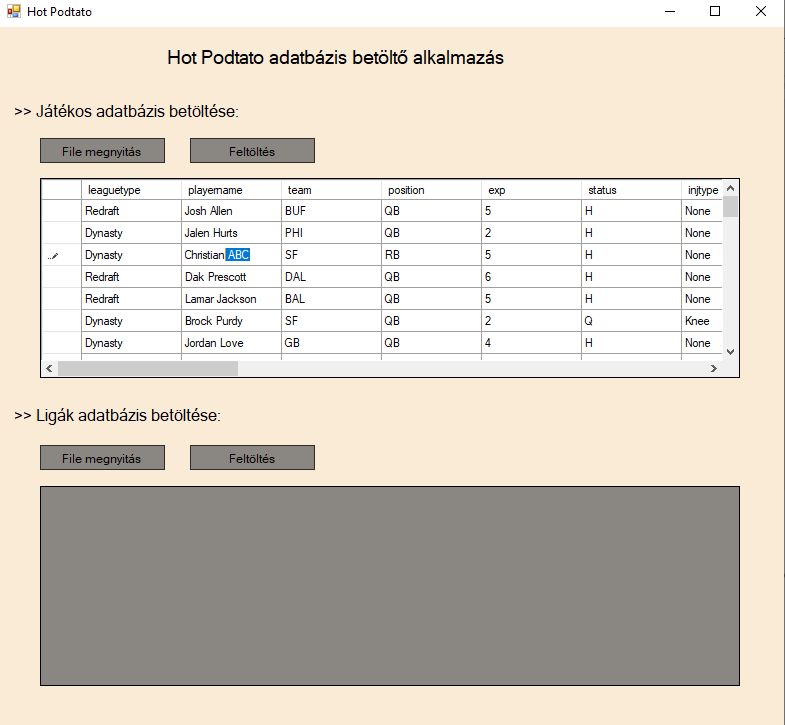
Ha a file-t betallóztok, akkor a megfelelő folyamatok lefuttatásával (amit lentebb mutatok be), egy DataGridView elembe kinyeri a file-ból a megfelelő adatokat ahhoz, hogy egy fejlécet alakítson ki a betöltendő táblázatnak.

A folyamat következő lépéseként pedig az adatok is elhelyezi a táblázat megfelelő sorában, és oszlopában, ez a *43.kép*en látható.



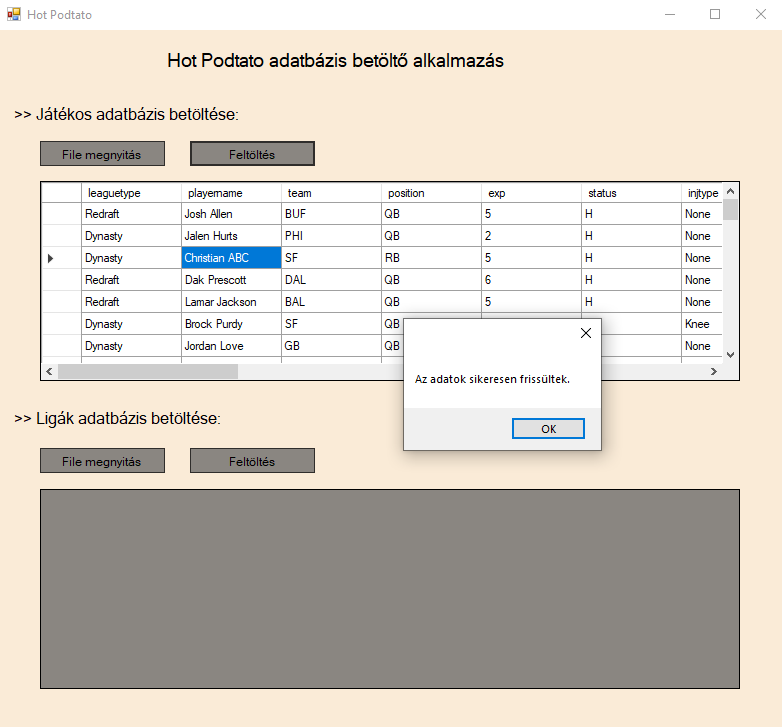
*43.kép*

Az adatok az egyes oszlopokban a fejlécre való kattintással sorrendbe rendezhetőek, illetve az adott „cellába” való kattintással szerkeszthetőek az adatbázisba való feltöltés előtt, ezt a *44.kép*en látható.



*44.kép*

Ezután az adatok Feltöltés gomb megnyomásával küldhetőek el az adatbázis táblába. A felküldés sikerességéről minden esetben kapunk visszajelző üzenetet egy alert ablakban, ez látható a *45.kép*en



*45.kép*

A működés úgy lett kialakítva, hogy button\_1\_Click\_1 eseményre (ami a Játékos adatbázis betöltése rész File megnyitás gombjának eseményvezérlője) OpenFileDialog segítségével megnyitjuk a párbeszédpanelt, ahol be tudjuk tallózni a file-t, ami az adatokat tartalmazza.

A file beolvasást StreamReader segítségével elvégezzük, és a file első sorának adatait vessző mentén feldarabolva átadjuk egy oszlnév nevű tömbnek.

Ezután a FeljecJatekos függvényt meghívjuk, és paraméterben átadjuk az imént létrehozott oszlnév tömböt annak adataival. A FeljecJatekos nevű függvény ezek alapján elkészíti a DataGridView-ben megjelenített táblázat fejlécét (a FeljecJatekos függvény pontos működését lentebb fejtem ki).

Ezután indítunk egy while ciklus azzal a megkötéssel, hogy addig fusson amíg van kiolvasható adat.

A kiolvasott adatokat vessző mentén szétdarabolja, majd az adat darabokat egy darab nevű tömbbe tölti fel.

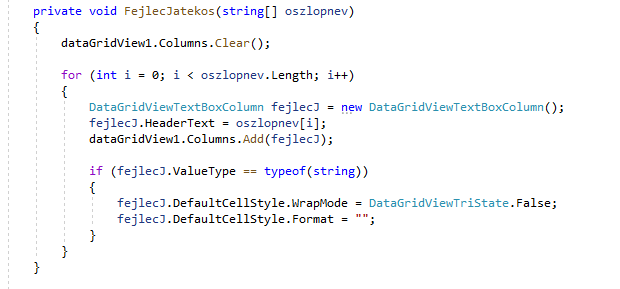
Ezután eltávolításra kerülnek egy for ciklus segítségével a megjelenítendő adatból az ”” karakterek.

Végül az adat darabokat a DataGridView-ben megjelenített táblázat megfelelő sorában elhelyezi, majd a hibakezeléssel zárul a folyamat, ez a *46.kép*en látható.



*46.kép*

A FeljecJatekos függvény úgy működik, hogy paraméterben megkapja az oszlopneveket, és ezen egy for ciklus segítségével végig megy, és a tömb elemeit egymást után elhelyezi az aktuális oszlop HeaderText-ében, ez a *47.kép*en látható.



Következő lépésként, ha az adatokat ellenőrizte a felhasználó akkor a Feltöltés gombbal továbbíthatja is azt az adatbázisba. Ez a Feltöltés gomb úgy működik, hogy megnyomásra elindít egy button2\_Click eseményvezérlőt.

A folyamat első lépéseként egy connectionString változóban megadjuk az adatbázis eléréséhez szükséges adatokat, majd egy torles nevű változóban az adott adatbázis tábla kiürítéséhez szükséges parancsot adjuk meg.

Alapvetően UPDATE adatbázis művelettel szerettük volna megoldani ez a feladatot, de megbízói igény volt, hogy a feltöltés előtt mindig törlésre kerüljön az aktuális tartalma a táblának.

Ezután létrehozzuk az adatbázis kapcsolatot, majd kiadjuk a szükséges törlésre vonatkozó parancsot az adatbázisnak.

Következő lépésként egy foreach függvény segítségével bejárjuk a DataGridView-ben található adatokat, egy beszuras nevű változóban meghatározzuk hova, és mit szeretnénk feltölteni, majd insertCommand segítségével a beszúrjuk a konkrét adatokat a DataGridView-ből. Legvégül pedig a hibakezelés található meg a kódban.

Ez látható a *47.kép*en és a *48.kép*en.



*47.kép*



*48.kép*

# 15. Hibák

Idő hiányában lekezelhető lehetett volna még a szerveroldali részben az a szituáció hogy nem meghatározott URL-ek beírása esetén a szerver 404-es hibát dobjon fel a felhasználó számára, esetleg egy olyan állományra hivatkozzon ami kiírja hogy „404 HIBA, Az oldal nem található…” esetleg „A szerver visszautasította a kérést!”.

Az autentikációnál kezelhető lett volna hogy még biztonságosabb lehessen az átirányítás. - A hitelesítésnél a felhasználó számára az átirányítás állománya nem látható mert változóban lett elmentve így lényegében biztonságosabb, de nem tökéletes.

# 16. Fejlesztési lehetőségek

Regisztráció lehetőség: Mivel prototípus oldalt készítettünk és csak adminisztrátorok számára biztosítottunk lehetőséget az adatok feltöltésére, így nem tartottuk lényegesnek. A helyes koncepció ennek lebonyolítására az lenne hogy az API-ban egy POST metódust írni ami az SQL parancsával INSERT INTO -val az értéket beszúrja az adatbázis táblájába ami jelen esetben a „users” tábla. Ezt követően kódolni lehetne a következőt…

Adminisztrátori jogosultság: Ha azt szeretnénk hogy az egyszerű felhasználók között kiemelkedjenek az adminisztrátorok akik kezelhetnek adatokat a szerveren és táblázatokban, hírekben vagy blogok között akkor a következőket úgy oldanám meg hogy: Az SQL táblákat módosítottam volna egy új mezővel ami BOOLEAN típusú lenne, ennek ugye két értéke lehet TRUE(igaz) vagy FALSE(hamis). Ezt követően az API-ban vagyis a szerver állományában(server.js) kéréssel kompenzálnám. Ha a bejelentkezést követően a szervertől azt a választ kapja hogy az adatbázis mezőben TRUE érték van akkor admin, ha nem akkor csak szimplán felhasználó. – Az adatbázisban csak adminisztrátorok adatai szerepelnek, így csak ők tudnak jelenleg bejelentkezni. További hozzáfűzés:

Az autentikáció résznél az átirányítást szerveren is lekezelhető lehetett volna. Ennek megvalósítására több mód is van amiből kiemelnék egy megoldást. Ha az SQL mezőben TRUE -ra állítódik az érték akkor elérhető lesz számára az állomány. Amennyiben FALSE érték van akkor az hitelesítés meghiúsult és a bejelentkező adatokat újra bekéri. Továbbá kliensoldalon is meg kell írni a kódot.

További jogosultságok: Elképzelhető hogy egy oldalon a biztonságos használathoz szükségessé válik hogy szerepköröket definiáljunk. Pl.: Felhasználó, Moderátorok, Adminisztrátor, Vezető-adminisztrátor, Rendszergazda. Ezeknek a szerepköröknek az adatbázisban adnám meg a jogosultságok szintjét. Pl.: 5555(Rendszergazda) / 4444(Vezető-adminisztrátor) / 3333(Adminisztrátor) / 2222(Moderátor) / 1111(Felhasználó). – Értelemszerűen minél nagyobb a mező attribútuma annál több mindent csinálhat az oldalon.

Olvassa tovább linkek: A bejegyzések alján megjelenítenék egyszerű ahref hivatkozásokat amelyek átnavigálnak a teljes bejegyzésre.

Kivonatok: A blogbejegyzésekhez és hírekhez létrehoznék kivonatokat(rövid tartalom ami csak a kártyákon látszik de a megnyitást követően a teljes tartalom jelenik meg, ezeket EXCERPT-nek nevezzük szakzsargonban) a megjelenítés során. – Így az adatforgalmak használatát csökkenthetjük. Ezt szintén adatbázisban lehetne lekezelni. Itt szintén egy új mezőt hoznék létre az adatbázisban ami mondjuk a VARCHAR maximális 255-karaktereit használni fel.

Képek: További kreativitásként díszíteni lehetne a bejegyzéseket képekkel amelyeket a tárhelyen(szerveren) tárolnék és ezeket kliensoldali programozásban valósítanék meg.

Előző-tovább navigálás: A teljes bejegyzéseket úgynevezett Paginationnal látnék el. Ezek az „Előző oldal” és „Következő oldal” szövegű hivatkozásokat jelenítenék meg. Erre főként akkor lehet szükség ha már sok bejegyzést kell megjeleníteni.

# Felhasznált irodalmak

<http://compalg.inf.elte.hu/~tony/Informatikai-Konyvtar/09-Programozas%20C-sharp%20nyelven/Programozas-Csharp-nyelven-Konyv.pdf>

<https://hu.wikipedia.org/wiki/C_Sharp>

<https://hu.wikipedia.org/wiki/HTML>

<https://mek.oszk.hu/14000/14068/pdf/14068.pdf>

<https://hu.wikipedia.org/wiki/Rel%C3%A1ci%C3%B3s_adatb%C3%A1zis>

<https://hu.wikipedia.org/wiki/MySQL>

<https://hu.wikipedia.org/wiki/XAMPP>

1. Forrás: https://hu.wikipedia.org/wiki/XAMPP [↑](#footnote-ref-1)