



EÖTVÖS LORÁND TUDOMÁNYEGYETEM

INFORMATIKAI KAR

MÉDIA- ÉS OKTATÁSINFORMATIKAI

TANSZÉK

Mozihálózat üzemeltető webalkalmazás

Témavezető:

Bakonyi Viktória
egyetemi adjunktus

Szerző:

Barnák Péter
programtervező informatikus BSc

Budapest, 2022

Az eredeti szakdolgozati / diplomamunka témabejelentő helye.

Tartalomjegyzék

1. Bevezetés	3
2. Felhasználói dokumentáció	5
2.1. Rendszerkövetelmények	5
2.1.1. Szoftver	5
2.1.2. Hardver(fejlesztéshez használt)	5
2.2. Webes alkalmazás	5
2.2.1. Telepítés	5
2.2.2. Webes alkalmazás használata	6
2.3. Asztali alkalmazás	14
2.3.1. Telepítés	14
2.3.2. Asztali alkalmazás használata	14
2.3.3. Közös felület	15
2.3.4. "Admin" felület	16
2.3.5. További felületek	29
2.3.6. "Product Supervisor" tab	30
2.3.7. "Product Seller" tab	31
2.3.8. "Ticket Seller" tab	31
3. Fejlesztői dokumentáció	32
3.1. A szakdolgozatban használt technológiák	33
3.2. Fejlesztéshez használt eszközök	34
3.3. Webszolgáltatás (ASP.NET Core WebApi) bemutatása	35
3.3.1. Kontrollerek bemutatása	38
3.3.2. A modell bemutatása	54
3.3.3. Adatbázis leírása	58
3.4. Asztali alkalmazás bemutatása	66
3.4.1. Vezérlés	70

TARTALOMJEGYZÉK

3.4.2. Modell	70
3.4.3. Nézetmodell	71
3.4.4. Nézet	73
3.5. Webes alkalmazás bemutatása	75
3.5.1. Az alkalmazás UML diagramja	76
3.5.2. Az alkalmazás struktúrája	78
3.5.3. Az alkalmazás nézetei és komponensei	81
3.6. Tesztelés	82
4. Összegzés	87
4.0.1. Továbbfejlesztési lehetőségek	88
Irodalomjegyzék	89
Ábrajegyzék	91
Táblázatjegyzék	93

1. fejezet

Bevezetés

A mai világban egy nagyon gyakran választott kikapcsolódási forma a filmnézés. Ezt a tevékenységet az ember pedig szereti a barátaival, családjával végezni. A filmnézés egyik leggyakrabban választott módja a mozitermek látogatása. Igaz, hogy a Covid-19 vírus megjelenése és gyors terjedése miatt a mozik nagy része bezárásra kényszerült az egészségügyi korlátozások miatt (ezzel együtt a filmgyártás is visszaszorult), viszont ettől függetlenül a mozik látogatása mindig is nagy érdeklődéssel fog rendelkezni a társadalmunkban.

Én magam is hatalmas rajongója vagyok a filmeknek és nagyon gyakran járok a barátaimmal mozitermekbe. Ezért is szerettem volna egyszer egy moziteremben dolgozni. Erre pár évvel ezelőtt lehetőségem is adódott, hogy diákmunka keretében belül egy nagy mozihálózat alkalmazottja lehessék. Sikeresen be is kerülttem, és elkezdhettem a munkámat mint jegyértékesítő.

A munkám során egy a cég számára külön fejlesztett alkalmazást használtunk, mely segített nekünk a jegyek értékesítésében, illetve a büfé kínálat eladásában is. A program felülete közel sem volt felhasználóbarát, illetve működésében is voltak problémák. A jó pár hibája miatt a felhasználóknak nagyon oda kellett figyelni a használatakor, ami igen csak rontott a munkateljesítményen illetve a munkamorálon is. Ezért döntöttem úgy, hogy szakdolgozatom témája pont egy ilyen mozikezelő alkalmazás lesz, amely sok tekintetben kezelhetőbb mint az ott használt program.

Az alkalmazás egy Windows platformra lefejlesztett program, amely lehetővé teszi egy mozi teljes körű üzemetetését. Több felhasználó felületet is biztosít azért, hogy éppen ki milyen szerepkörrel rendelkezik, illetve jelentkezik be a rendszerbe. Az alkalmazás célja a lehető leggyorsabb és legkényelmesebb munkavégzés biztosítása.

tása. A mozirendszer kezelő alkalmazáshoz tartozik egy webalkalmazás is, amelyen keresztül a látogatók értesülhetnek az aktuális filmekről, illetve foglalhatnak helyet azokra.

2. fejezet

Felhasználói dokumentáció

2.1. Rendszerkövetelmények

2.1.1. Szoftver

- Operációs rendszer: Windows 10 vagy annál újabb, 64 bit
- Állományok kicsomagolása: Winrar [1]
- Telepítendő keretrendszer: .NET 6.0 [2] (ezt az alkalmazás könyvtára tartalmazza), illetve Node.js [3] legfrissebb verziója

2.1.2. Hardver(fejlesztéshez használt)

- Tárterület: 340 Mb
- Processzor: AMD Ryzen 7 2700
- RAM: 16 Gb
- GPU: NVIDIA GeForce GTX 1050Ti

2.2. Webes alkalmazás

2.2.1. Telepítés

A program indítása a *Node.js* futtatónkörnyezet feltelepítése után nyissunk egy új terminál ablakot. A terminálba írjuk be az alábbi parancsokat.

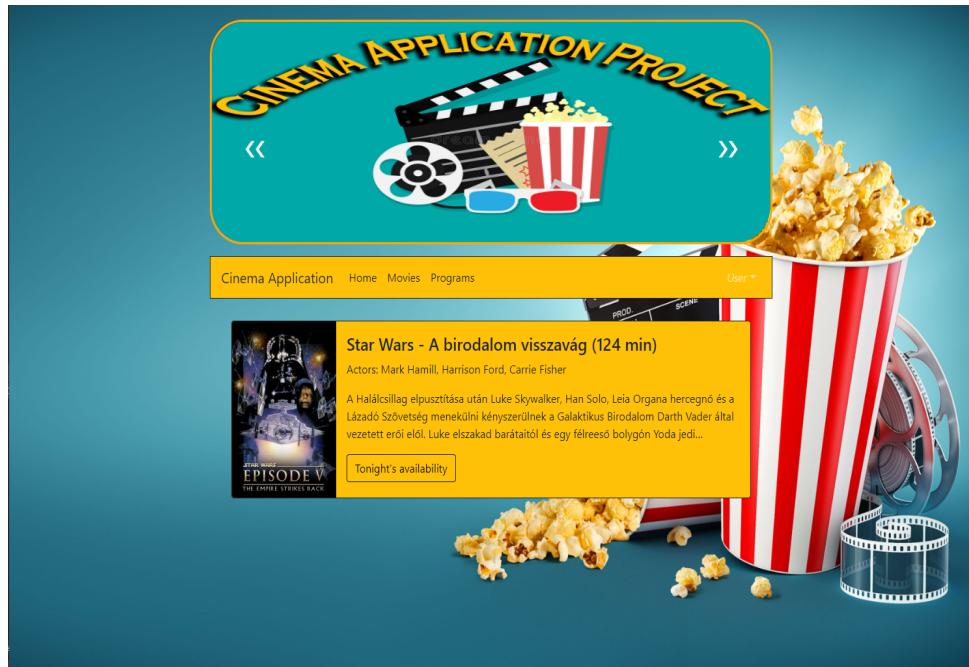
- `npm install`
- `npm run serve`

A második parancs végrehajtása hosszabb időt is igénybe vehet, ez rendszerfüggő. Ezt követően sikeresen elindult a weboldalunk. Ahhoz, hogy ezt elérjük nyissunk egy böngésző ablakot és a címsorba gépeljük be az alábbi webcímét: `http://localhost:8080`. Fontos megjegyezni, hogy ezzel az utasítással csak fejlesztői módban érhetjük el az alkalmazásunkat. Ebben a módban lehetőségünk van az alkalmazásunk tesztelésére, illetve ha módosítunk valamit annak a forráskódján, akkor az azonnal tesztelhető lesz a böngészőnk oldalán.

Ahhoz, hogy egy teljes optimalizált build-et készítsünk futtassuk le a `npm run build` utasítást, amellyel a `CinemaApplicationProjectWeb/dist` mapába előáll egy lefordított verziója az előbb fejlesztői módban futtatott alkalmazásunknak. Ez a lefordított verzió jelentősen gyorsabb és jobb teljesítményű, mint a fejlesztői. A build-elt verziójú alkalmazásunkat egy webtárhelyre felhelyezve (a tárhelyre is telepítve a `Node.js` futtatókörnyezetet) egy terminálból könnyen elindíthatjuk a `serve -s dist` parancssal. A parancs beütése után böngészőnkben a `http://localhost:3000` címen érjük el a futó alkalmazást. Természetesen ezt lokálisan a saját számítógépünkről is megtehetjük, de később, ha szeretnénk ezt élesben futtatni, célszerű valamilyen webtárhelyet (akár felhő alapú webtárhelyet) használnunk.

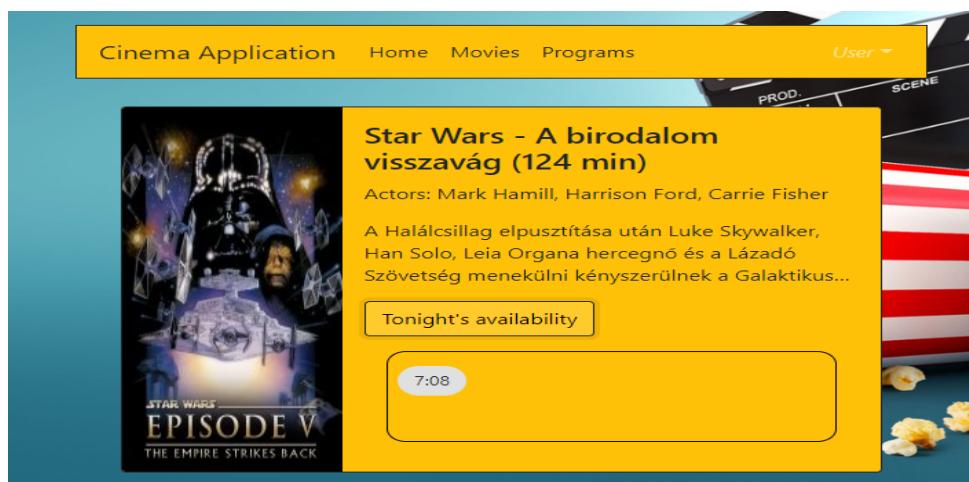
2.2.2. Webes alkalmazás használata

Az alkalmazásunk elindítása és böngészőn kereszttüli elérése után a 2.1. ábrán látható *Kezdőlapon* találhatjuk magunkat. A következőkben végigvesszük innen indulva, hogy milyen funkciói vannak a weboldalunknak és azokhoz, hogy tudunk elnavigálni.



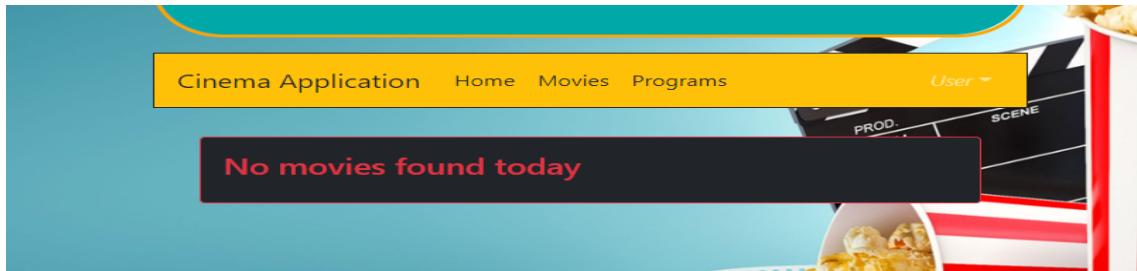
2.1. ábra. "Home" oldal

A *Home* oldalon az aktuális napi filmek tekinthetőek meg, illetve ezekhez tarozóan láthatjuk a műsor kezdésének időpontját is. A filmekről kapunk egy rövid ismertetőt, elolvashatjuk a benne szereplő színészeket, illetve a film címe mögött zárójelek között láthatjuk a film hosszát is. Az időpontok megjelenítésére kattintunk a *Tonight's availability* gombra. Ekkor az adott kártya alsó részében láthatóvá válnak a mai előadások időpontjai (2.2. ábra). Az előadások időpontjaira kattintva foglalhatunk helyet azokra. Egy későbbi bekezdésben ezen felületet is ismertetem.



2.2. ábra. Választott mozifilm, kinyitott előadások füllel

Ha nem találhatóak az adott napra már további előadások, akkor arról is értesülünk egy hibaüzenet formájában. (2.3. ábra)



2.3. ábra. Hibaüzenet, az előadások hiányáról

A megjelenített filmek címére kattintva elérhetők azok adatlapja is (2.4. ábra). Ezen az adatlapon már egy bővebb leírást olvashatunk a filmről, megjelenik a rendező neve, illetve egy listába felsorolva a filmhez tartozó színészek is. Ezenfelül beágyazott formában megtekinthetjük az aktuális film előzetesét is. A lap alján megjelenő szekcióban olvashatunk az adott filmről más felhasználók által készített véleményeket is. Lehetőségünk van saját véleményt is hozzáadni, ezt az *Add opinion* gombbal tehetjük meg.

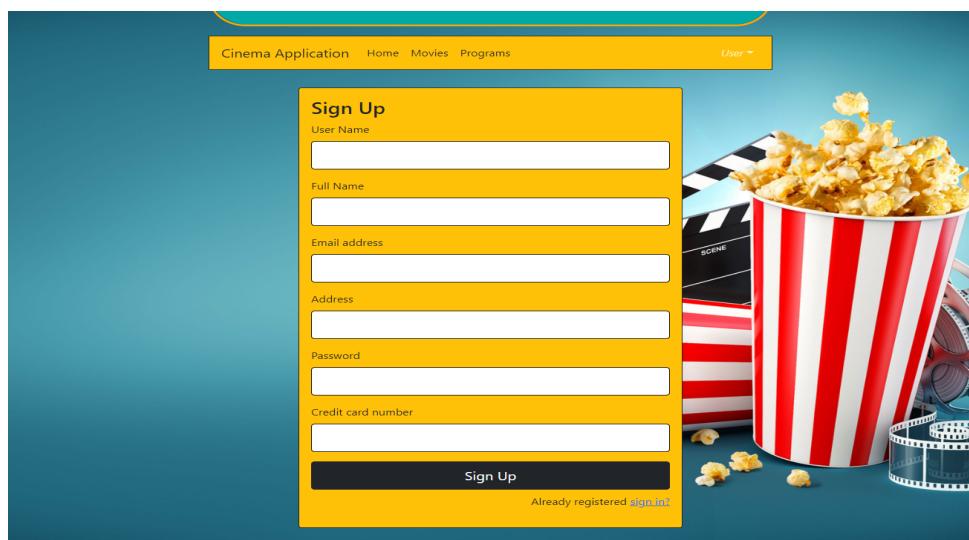
A screenshot of the 'Cinema Application' showing a detailed movie page for 'Star Wars - A birodalom visszavág' (Episode V). The page has a teal header with the application logo and a yellow main content area. The movie poster for 'Star Wars: Episode V - The Empire Strikes Back' is prominently displayed. Below the poster, the movie title and duration ('124 perc') are shown. The director ('George Lucas') and actors ('Mark Hamill, Harrison Ford, Carrie Fisher') are listed. A descriptive text about the plot follows. Below this, there's a video player for a 'MODERN TRAILER' with options to 'Watch on YouTube' or 'Watch later'. At the bottom of the page, there's a section for user reviews ('Add Opinion') with three entries from different users ('admin2', 'Anonymous', 'Anonymous') with their ratings and comments. The page is framed by a decorative border of popcorn and a movie clapperboard.

2.4. ábra. Teljes oldalas kép a filmek adatlapjáról, véleményekkel

Egy új vélemény létrehozásához be kell regisztrálnunk, majd bejelentkeznünk, így előbb nézzük meg ezek felületét. A menüsávunk jobb részén a *User* lenyíló menüre kattintva érhetjük el ezeket a felületeket (2.5. ábra). A *Regisztráció* felülete a 2.6. ábrán látható. A regisztráció során különböző adatokat kell megadnunk magunkról. Ezek megadása kötelező, tehát egyik beviteli mező sem maradhat üresen. Ha a regisztráció sikeres, ismét a *Kezdőlapon* találjuk magunkat. Ezt követően bejelentkezzünk be a *Login* gombra kattintva.

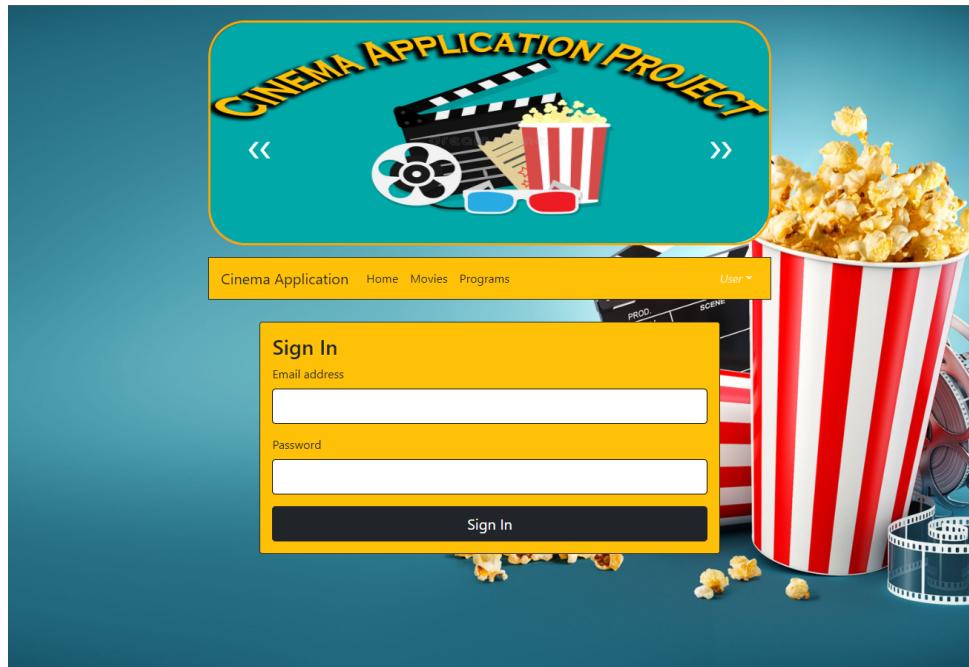


2.5. ábra. Menüsáv a lenyíló User menüvel



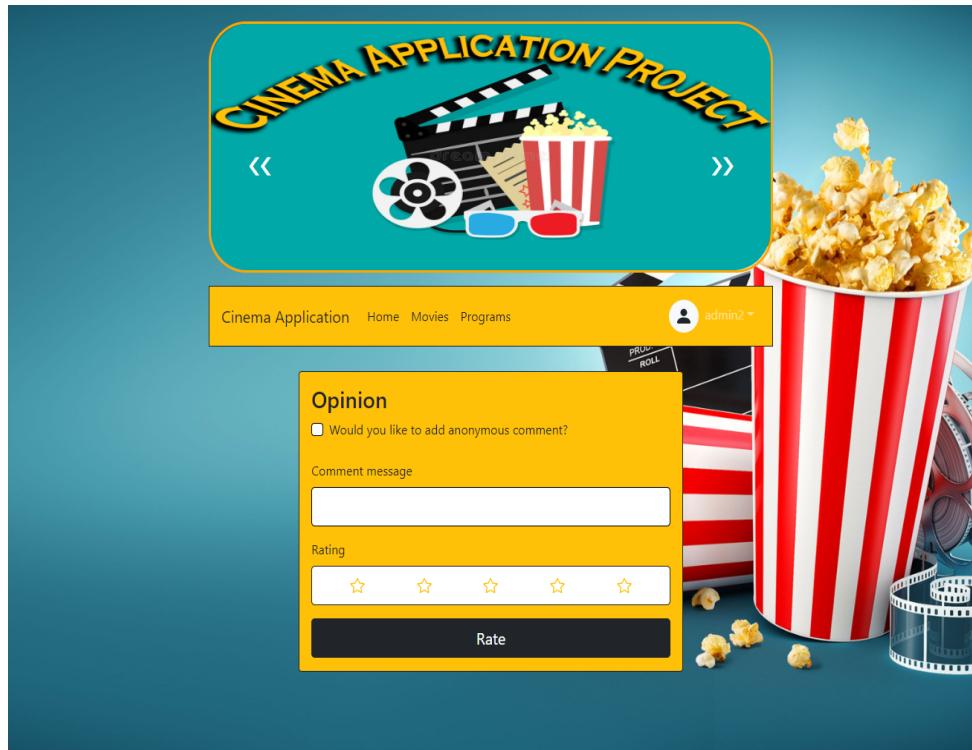
2.6. ábra. Regisztrációs felület

A bejelentkezési felületen két adat megadása szükséges a sikeres bejelentkezéshez (2.7. ábra). Először meg kell adnunk a felhasználónévünket, majd az ehhez tartozó jelszót. Ha bármilyen hibát vétünk, arról az oldal tájékoztat minket. Sikeres bejelentkezés esetén ismét a főoldalon találjuk magunkat.



2.7. ábra. Bejelentkezési felület

A regisztráció, illetve bejelentkezés után már hozzá is adhatjuk a saját véleményünket egy filmhez. Ehhez ki kell választanunk ismét a filmet, majd az *Add opinion* gombra kattintva már meg is jelenik számunkra az ehhez szükséges felület (2.8. ábra). Ezen a felületen beállíthatjuk, hogy szeretnénk-e anonim módon megosztani a véleményünket. Ilyenkor a véleményeket megjelenítő listába nem lesz kiírva a nevünk. Ezt követően adunk kell véleményt és egy értékelést a filmről. Az értékelést a csillagok segítségével tehetjük, ahányadik csillagot választjuk ki, akkora pontszámot kap tőlünk a film.



2.8. ábra. Vélemény hozzáadása

A következő felület a *Movies* oldal (2.9. ábra). Ezen az oldalon láthatjuk a mozi által játszott filmeket. Nem csak az aktuálisokat, hanem minden. Az oldal különlegessége, amiben eltér például a *Home* oldaltól, hogy itt a mindenkorai filmek között tudunk szűrni kategóriánként vagy pedig filmcím részlet alapján. Egyszerre csak egy szűrési feltétel alkalmazható. Ezen az oldalon is lehetőségünk van filmek adatainak megtekintésére, csakúgy mint a *Home* oldalon.

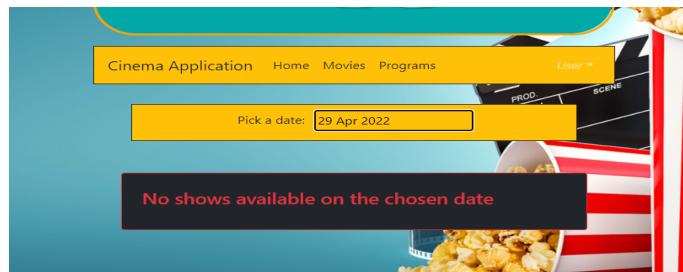


2.9. ábra. "Movies" oldal

Az utolsó hátralévő nagyobb felület a *Programs* oldal (2.10. ábra). Ezen az oldalon tekinthetjük meg a kiválasztott dátumhoz tartozó előadásokat filmek szerint egybe gyűjtve. Lehetőségünk van dátumot is választani, ekkor a választott dátumhoz tartozó filmek listáját láthatjuk megjelenítve. Ha esetleg egy olyan dátumot választunk amelyen nincs előadás arról hibaüzenet formájában tájékoztat minket a rendszer (2.11. ábra).

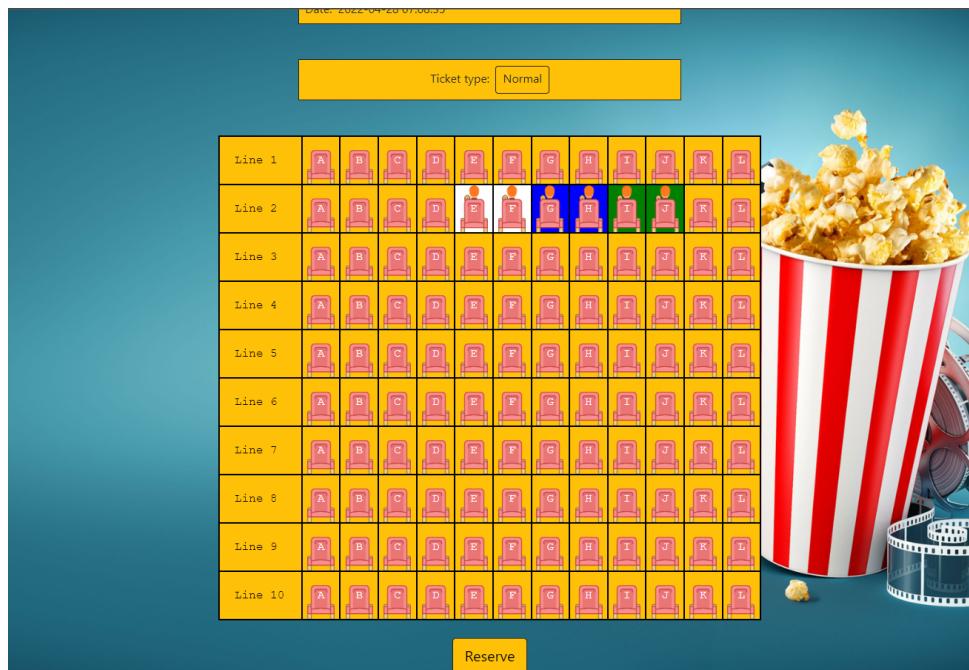


2.10. ábra. "Programs" oldal



2.11. ábra. Hibaüzenet, ha nincs előadás a választott napra

Egy általunk megfelelő időpontra kattintva megjelenik előttünk a *Reserve* oldal (2.12. ábra). Ezen a felületen láthatunk pár adatot a foglalásról, például, hogy pontosan melyik filmről is van szó, illetve, hogy melyik terembe szól a foglalásunk. A terem méretét egy NxM-es gombrács ábrázolja. A gombrács felső részén egy le-gördülő menü található, amelyben van lehetőségünk 3 jegytípus közül választani (diák, normál, nyugdíjas). Ha kiválasztottuk az általunk foglalni kívánt hely típusát, már kiválaszthatjuk a megfelelő helyeket a mezőn. Miután kiválasztottuk a számunkra tetsző helyeket, vélegesíthetjük a foglalásunkat az oldal alján található *Reserve* gombbal. Fontos viszont megjegyezni, hogy a foglalás vélegesítéséhez be kell jelentkeznünk. A *Login* felület elérését az előzőekben már bemutattam.



2.12. ábra. Reserve felület, kiválasztott helyekkel

2.3. Asztali alkalmazás

2.3.1. Telepítés

Az asztali alkalmazás esetében telepítésre nincs szükségünk. Az alkalmazást egyszerűen elindíthatjuk ha a CinemaApplicationProject/CinemaApplicationProject/Desktop/build mapában található CinemaApplicationProject.Desktop.exe-t futtatjuk. minden az alkalmazás futásához szükséges fájl megtalálható ebben a mappában, így az indításkor a rendszer automatikusan megtalálja azokat.

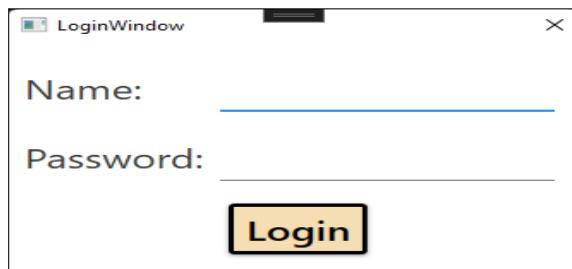
2.3.2. Asztali alkalmazás használata

Az alkalmazásnak több nézete is van, attól függően, hogy az adott felhasználó minden szerepkörrel rendelkezik. A szerepkörök határozzák meg, hogy az egyes felhasználók pontosan milyen feladatot látnak el a mozi működésén belül. Egy felhasználó lehet adminisztrátor, büfé vezető, pénztárvezető, pénztáros vagy büfés. Ezek az alap szerepkörök, melyek az alkalmazáson belül bővíthetők szükség szerint. A fő nézete az adminisztrátori felület, ezen, mint a neve is sugallja, minden funkció teljesen elérhető. Ezen felület bemutatásával szeretném ismertetni az asztali alkalmazás funkcióit.

A funkciók bemutatása végén pedig összegzésképp egy rövid leírásban ismertetem, hogy melyik szerepkörökhez mely funkciók tartoznak az adminisztrátori felületről.

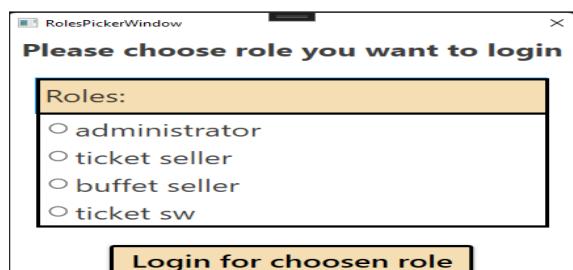
2.3.3. Közös felület

Ahhoz, hogy elérjünk bármilyen felületet az alkalmazáson belül, be kell jelentkeznünk. Ehhez egy kis ablakot kapunk, ahova be kell írnunk a felhasználónevünket és az ahhoz tartozó jelszót (2.13. ábra).



2.13. ábra. Bejelentkezésre szolgáló felület

Hibásan megadott adatok után a rendszer nem engedi bejelentkeztetni a felhasználót és erről egy hibaüzenetben értesíti is azt. Ha egy felhasználó több szerepkörrel is rendelkezik, akkor a sikeres bejelentkezést után egy választó felület jelenik meg, ahol ki tudja választani, hogy milyen szerepben szeretne belépni az alkalmazásba (2.14. ábra).



2.14. ábra. Szerepköröket kiválasztó ablak

A bejelentkezést követően minden felület felső részén található egy *Refresh lists* gomb, mellyel a felületeken megjelenő listákat frissíthetjük, azaz tölthetjük újra őket az adatbázisból, ezzel megkapva esetleges módosításokat melyeket mások végezhettek a rendszerben. Ez a gomb a 2.15. ábrán piros körrel megjelölve látható.

AdminMainWindow				
Cinema Movies Shows Users Ticket Selling Buffet Warehouse				
Rooms:			Tickets:	
Kilincs	12	10	Delete	Delete
Kristály	20	20	Delete	Delete
Kopár	6	6	Delete	Delete
			Add Room	Add Ticket
			normal	1200
			student	900
			retired	700
			Delete	Delete

2.15. ábra. "Refresh list" gomb

2.3.4. "Admin" felület

A felület eléréséhez adminisztrátori szerepkörrel kell rendelkeznünk, tehát ezt a felületet nem érheti el minden alkalmazott, csak a mozi vezetősége használhatja. Ezáltal mivel ez a vezetők által használt felület, ezért itt minden funkció elérhető, és használható.

Ha rendelkezünk a kívánt szerepkörrel, akkor bejelentkezés után már láthatjuk is ezt a felületet.

A felület több almenüvel rendelkezik. Ezekre az almenükre innentől kezdve *tab*-ként fogok utalni. minden ilyen *tab*-on egy-egy valamilyen mozival kapcsolatos tevékenység funkciói vannak elszeparálva.

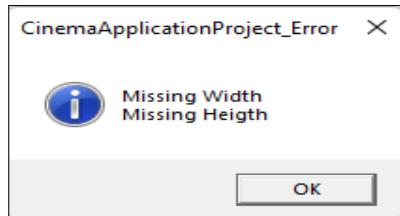
Listák általános leírása

Listák esetében lehetőségünk van elemek kiválasztására és ezek szerkesztésére. Ebben az esetben, ha kiválasztunk egy elemet, akkor a lista alatt látható eddig üres területen megjelenik egy *Details* nevű terület, a kiválasztott elem adataival. Ezek az adatok egy-egy beviteli mezőben láthatóak, ezáltal lehetővé téve a szerkesztést az egyes adatokon. Ha befejeztük a szerkesztést és készen állunk ezt menteni, kattintunk a *Details* területen található *Update ...* gombra, ahol a ... az adott elem típusát jelöli (ezt a jelölést az elkövetkezőkben is használni fogom).

Majdnem minden listához lehetőségünk van hozzáadni új elemet is. Ezt úgy tehetjük meg, hogy a lista neve mellett lévő *Add ...* gombra kattintunk, majd a lista alatt megjelenik a már bemutatott *Details* rész. Ebben az esetben viszont a *Details*-ben lévő beviteli mező értékei üresek, hiszen új elemet szeretnénk hozzáadni, tehát nekünk kell kitölteni azokat. Ha kitöltöttük minden mezőt, kattintsunk a *Update ...* gombra, ezzel menthetjük el az új elemet az adatbázisba. A változtatásainkat a listák azonnal követik, így azokat egyből láthatjuk is.

Listák esetében törölhetünk is elemeket. Ennek a folyamata nagyon egyszerű: elég egy általunk választott elem sorában a *Delete* gombra kattintanunk.

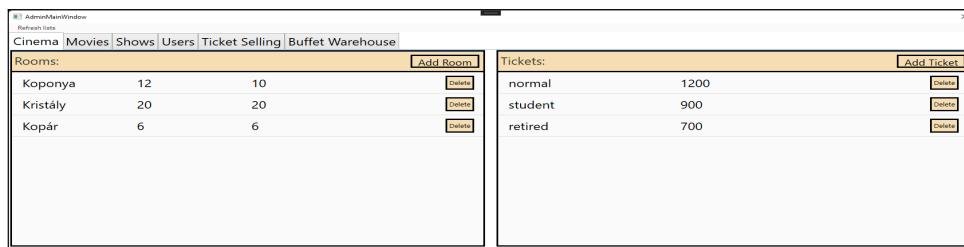
Listák adatainak szerkesztésénél, vagy új adat hozzáadásánál előfordulhatnak úgynevezett validációs hibák. Ezek a hibák abból következnek, hogy valamely mezőt elfelejtjük kitölteni, pedig ennek kitöltése kötelező lenne. Erről a program egy felugró ablakban értesít minket, erről látható egy példa a 2.16. ábrán.



2.16. ábra. Validációs hibaüzenet helytelenül kitöltött mezőkről

"Cinema" tab

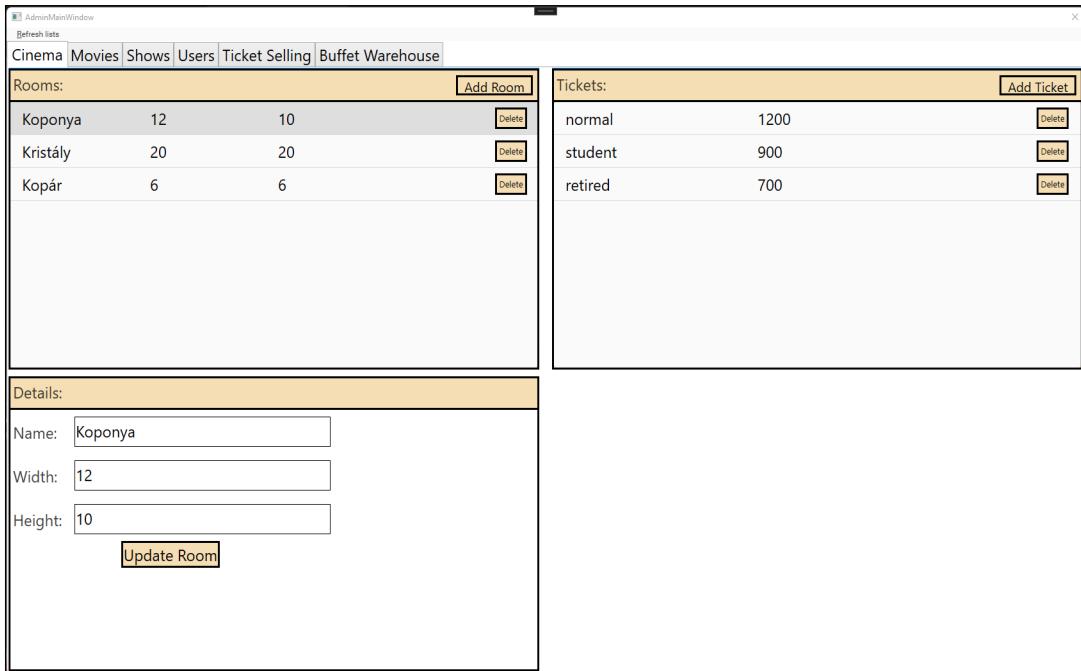
Ez az első *tab*, a bejelentkezés után a betöltést követően itt találjuk magunkat (2.17. ábra). Ezen az oldalon a mozi termeit szerkeszthetjük illetve a mozi jegytípusait tudjuk módosítani. A képernyő ezek alapján oszlik két részre. Baloldalon a termek, jobb oldalon pedig a jegyek listáját láthatjuk.



2.17. ábra. A mozi felület

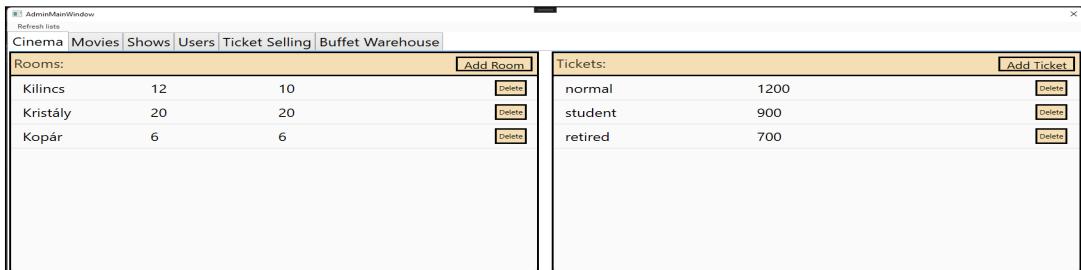
A "Listák általános leírása" részben ismertetett szerkesztési és hozzáadási opcióra itt ezen tab esetében szeretnék egy példát hozni, a könnyebb megértést segítve.

Vegyük a termek listáját. 2.17. ábrán látható termek közül kiválasztva az elsőt a megjelenik a 2.18. ábra *Details* része. Itt láthatjuk, hogy az egyes beviteli mezők (Name, Width, Height) értéket kaptak, melyeket szerkeszthetünk. Például írjuk át a nevet "Kilincs"-re, majd kattintsunk az *Update Room* gombra, a *Details* alsó részén.



2.18. ábra. A mozi felület elérhető szerkesztővel

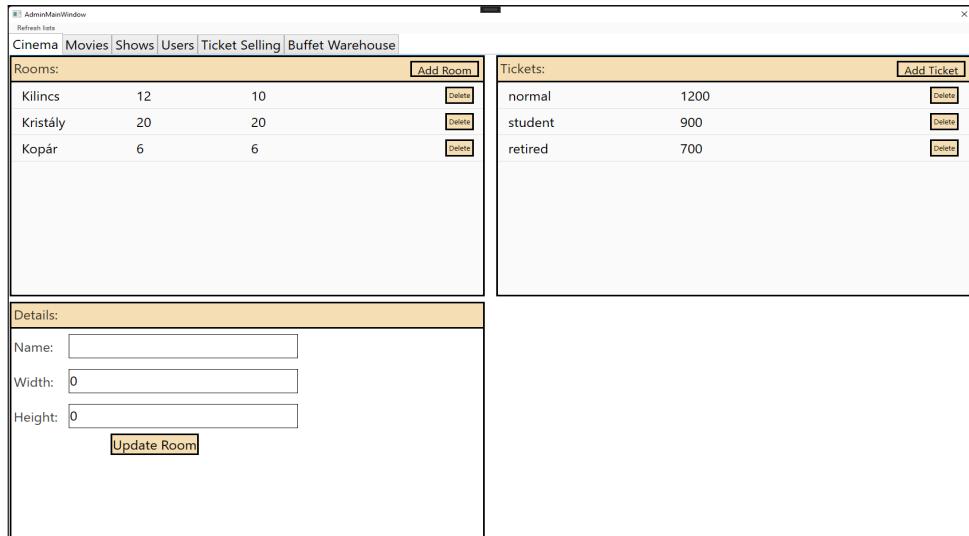
A módosítást már láthatjuk a listánkban (2.19. ábra).



2.19. ábra. A lefrissült terem lista

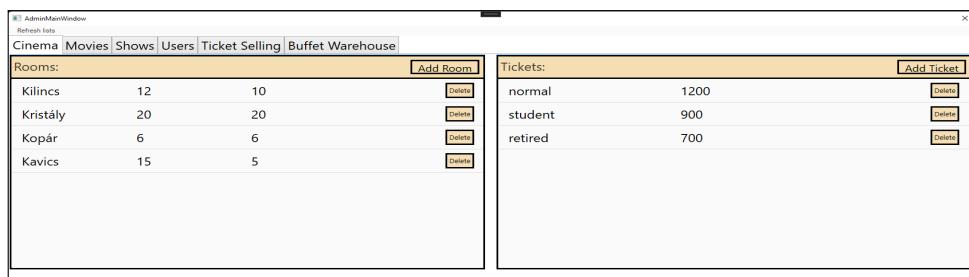
Nézzünk egy példát az új elem hozzáadására is. Kattintsunk a lista jobb felső sarkában található *Add Room* gombra. Ekkor a 2.20. ábrán látható módon, ismét elérhető válik a *Details* terület, viszont mint láthatjuk itt már üres értékekkel. Töltsük ki ezeket rendre a következő értékekkel: Kavics, 15,5. Ha kitöltöttük kattintsunk az *Update Room* gombra amely végre hajtja a módosításainkat.

2. Felhasználói dokumentáció



2.20. ábra. Új elem hozzáadása a "Details" szekcióban

A sikeres végrehajtás után már láthatjuk is az eredményét a listánkban (2.21. ábra).

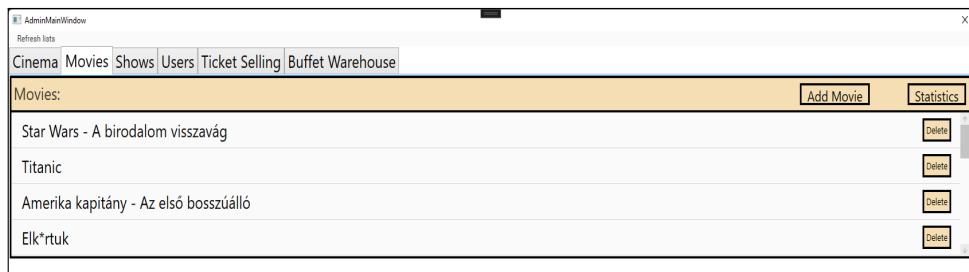


2.21. ábra. Sikeres hozzáadás eredménye

Természetesen az előbbiekben hozott példa a későbbiekben (például a jegyeknél) is érvényes, a működése ugyanaz, ezért arra nem térnék ki külön.

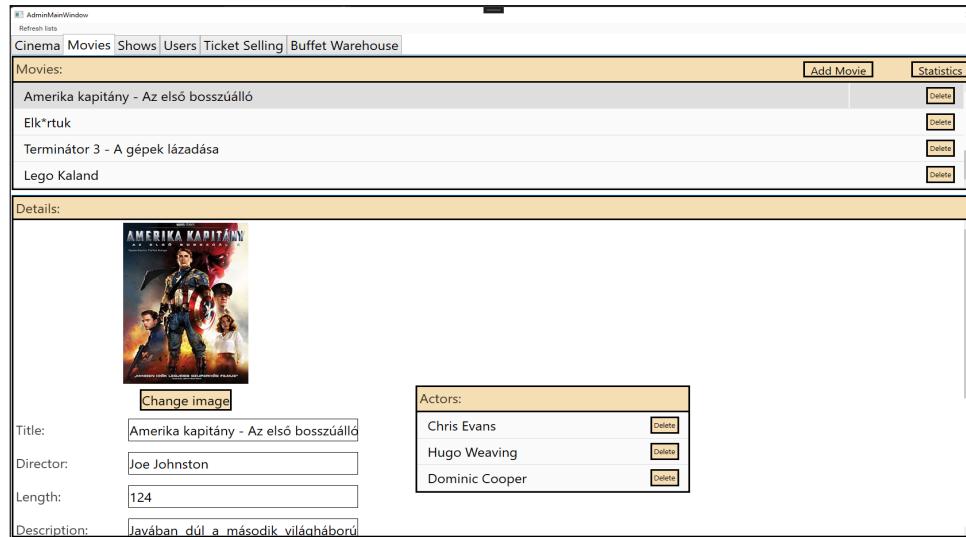
"Movies" tab

Ezen a felületen a mozi által játszott filmeket láthatjuk, szerkeszthetjük, illetve kérhetünk le azokról statisztikai adatokat (2.22. ábra).



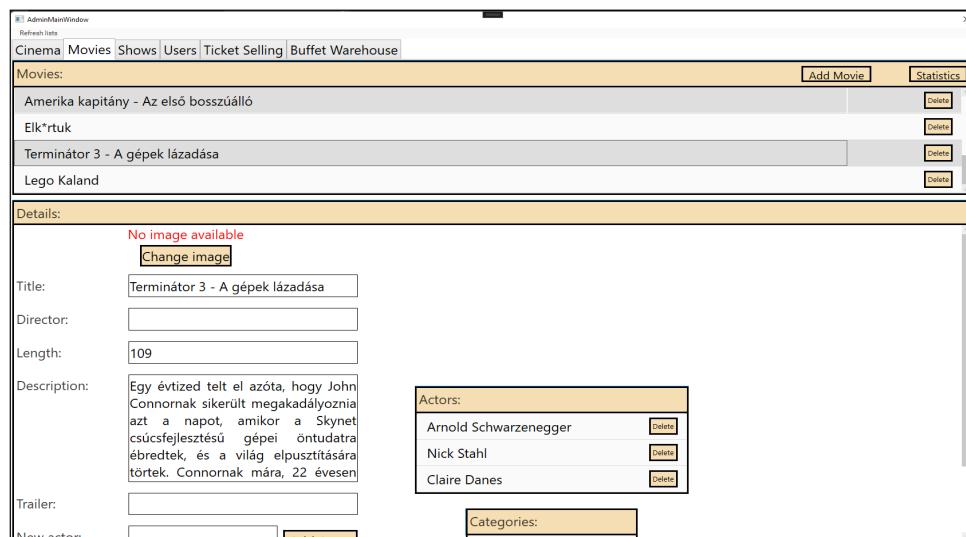
2.22. ábra. A filmek felület

A "Listák általános leírása" részben ismertetett szerkesztési és hozzáadási metódusok itt is ugyanazok, minden ugyanúgy megtehetünk. Viszont itt a szerkesztésnél/hozzáadásnál nem csak sima beviteli mezőink vannak, hanem mellettük található még két kibővített beviteli lehetőség is. Az első (ez a *Details* tetején található, 2.23. ábra) egy kép hozzáadási lehetőség.



2.23. ábra. Képpel rendelkező film

Ha az adott filmhez még nincs hozzáadva kép, akkor piros színnel a "No image available" szöveg olvasható, viszont ha van hozzárendelt kép, akkor pedig a kép kicsinyített verziója látható a felületen (2.24. ábra és 2.23. ábra).

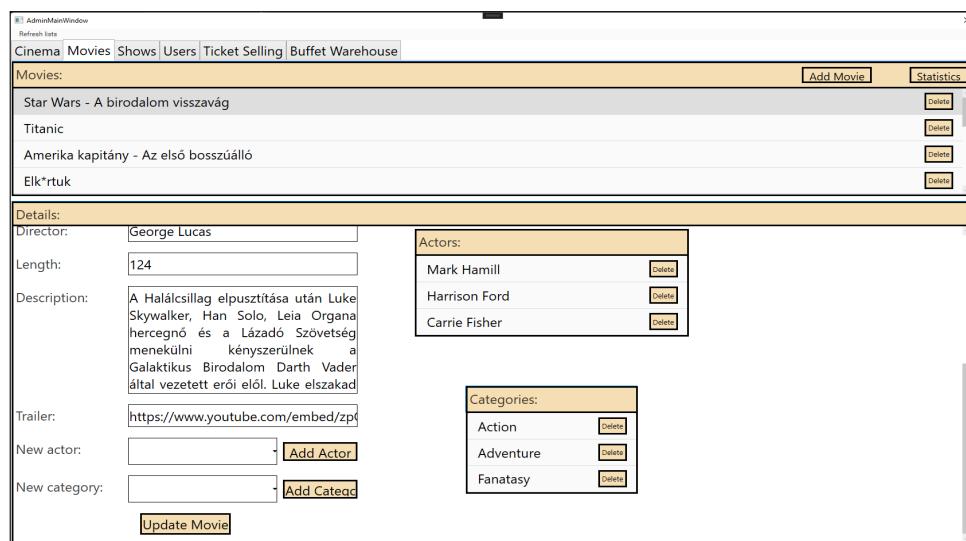


2.24. ábra. Film hiányos képpel

Lehetőségünk van módosítani ezt a képet, vagy pedig újat hozzáadni, ha a *Change image* gombra kattintunk. Ilyenkor megjelenik a fájl tallázására alkalmas

ablak, ahol a számítógépünkön tárolt képek között böngészhetünk, és tölthetjük fel a nekünk tetsző képet.

Filmek esetében lehetőségünk van azokat színészekhez illetve kategóriákhoz rendelni. Ezeket szintén a *Details* részben találjuk, annak is a jobb oldalára kerültek ezek a listák (2.25. ábra). Ezen listák esetében nem alkalmazhatók a "Listák általános leírása" részben megfogalmazott szerkesztési, és hozzáadási funkciók, viszont a törlés itt is ugyanúgy néz ki. A *Details* szekcióban található 2 db legördülő beviteli mező, az egyik *New actor* névvel, a másik pedig *New category* névvel, mellettük egy-egy *Add ...* gombbal. Ezen speciális beviteli mezők biztosítják új színészek vagy új kategóriák hozzáadását a filmekhez.



2.25. ábra. Egy film színészei és kategóriái az ezekhez tartozó beviteli mezőkkel

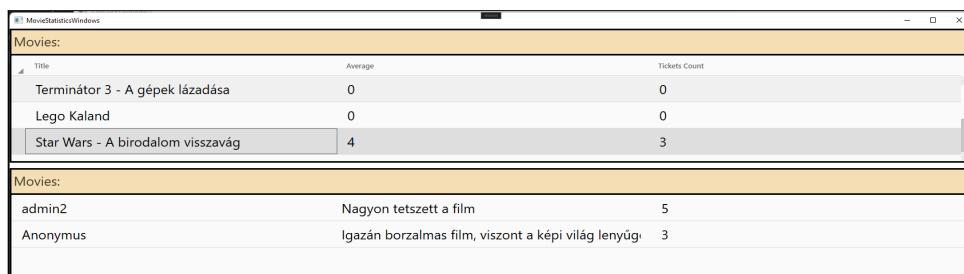
Használatuk a következőképpen működik. Vegyük például a színészeket. Ha bekattintunk a *New Actor* felirat mellett található beviteli mezőbe, akkor megjelenik a mező felett számunkra egy lista, színészek neveivel. Ez a lista minden színész nevét tartalmazza, aki valamelyik filmhez rendelve van a mozinkon belül. Ekkor lehetőségünk van választani közülük (sőt, ha elkezdjük gépelni, a lista folyamatosan szűrni fog a beírt szöveg szerint), és őt hozzáadni a kiválasztott filmünk színész listájához. A hozzáadást az *Add Actor* gombbal tehetjük meg.

Viszont, nem csak az adott listából tudunk választani. Lehetőségünk van olyan színészt is hozzáadni a listához, aki még nem szerepel az adatbázisban. Ekkor elegedő beírnunk a mezőbe a nevét, majd az *Add Actor* gombra kattintanunk és a rendszer hozzáadja a film színészeihez, illetve felveszi őt az adatbázisba is.

A *New category* címke mellett található legördülő beviteli mező is ugyanezen az elven működik. A későbbiekben még találkozni fogunk ezzel a speciális beviteli formátummal, így innentől az egyszerűség kedvéért *dropdown*-ként fogok rá hivatkozni.

A dropdown-ok használata után nem kell rákattintanunk az *Update Movie* gombra, ezt a frissítést a dropdownhoz tartozó *Add ...* gomb elvégzi.

Végezetül pedig tekintsük meg a filmekhez tartozó statisztikai adatokat. Ezt a funkciót a filmek lista felső részén lévő *Statistics* gombbal érhetjük el. Ez egy külön ablakot nyit, amelyet a 2.26. ábrán láthatunk. Ebben az ablakban filmekre lebontva láthatjuk, hogy melyik filmet hány pontra értékelték a nézők, illetve, hogy melyikre hány darab eladt jegy van eddig. Ha a kilistázott filmek közül kiválasztunk egyet, akkor egy részletes listában az adott filmhez tartozó véleményeket, szerző nevével és az általa leadott pontszámmal ellátva. Itt nincs lehetőségünk szerkesztésre, ez csak egy adatokat megjelenítő felület.



The screenshot shows two stacked windows. The top window is titled 'MovieStatisticsWindows' and contains a table with movie statistics. The columns are 'Title', 'Average', and 'Tickets Count'. The rows show data for three movies: 'Terminátor 3 - A gépek lázadása' (Average 0, Tickets Count 0), 'Lego Kaland' (Average 0, Tickets Count 0), and 'Star Wars - A birodalom visszavág' (Average 4, Tickets Count 3). The bottom window is also titled 'MovieStatisticsWindows' and contains a table with reviews. The columns are 'User' and 'Review'. The rows show reviews from 'admin2' ('Nagyon tetszett a film') and 'Anonymous' ('Igazán borzalmás film, viszont a képi világ lenyűgözte')).

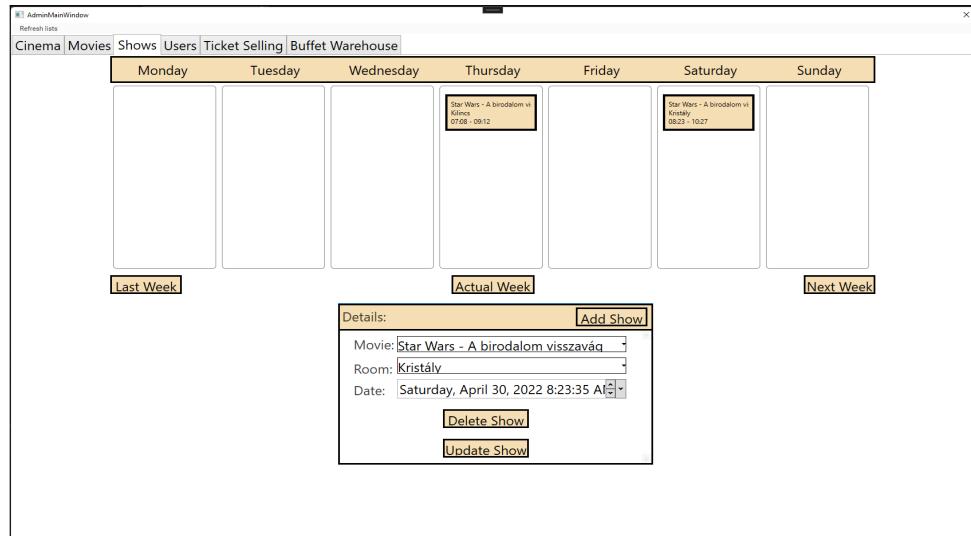
Title	Average	Tickets Count
Terminátor 3 - A gépek lázadása	0	0
Lego Kaland	0	0
Star Wars - A birodalom visszavág	4	3

User	Review
admin2	Nagyon tetszett a film
Anonymous	Igazán borzalmás film, viszont a képi világ lenyűgözte

2.26. ábra. A filmek statisztikai felülete véleményekkel

"Shows" tab

Ez a felület felelős az előadások megtekintéséért, és azok szerkesztéséért. Egy táblázatos formában tekinthetjük meg a hétfő napjait, majd minden nap kártyákon olvashatjuk le az aktuális előadásokat (2.27. ábra).



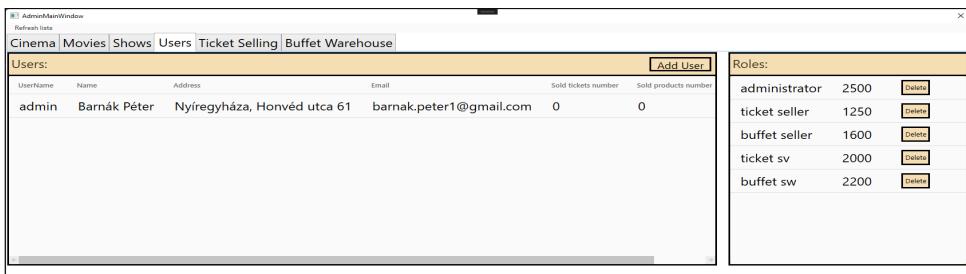
2.27. ábra. Az előadások felület

Lehetőségünk van megtekinteni az előző, illetve a következő heti előadásokat is. A múlt heteket: a *Last Week* gombbal a következőket pedig a *Next Week* gombbal. A felületen található egy harmadik gomb is: (*Actual Week*), ezzel az aktuális hétfeladásait hozhatjuk ismét előtérbe.

Ahhoz, hogy szerkeszteni tudjuk az előadásokat kattintsunk kétszer egy kártyára, ekkor a film adatai márás szerkeszthetővé válnak az erre alkalmas területen. Természetesen új előadást is tudunk hozzáadni, illetve törölni is tudjuk a már meglévőket. A törléshez ismételten csak ki kell választanunk egy műsort és a szerkesztő terület alsó részében lévő *Delete Show* gombbal már törölhetjük is.

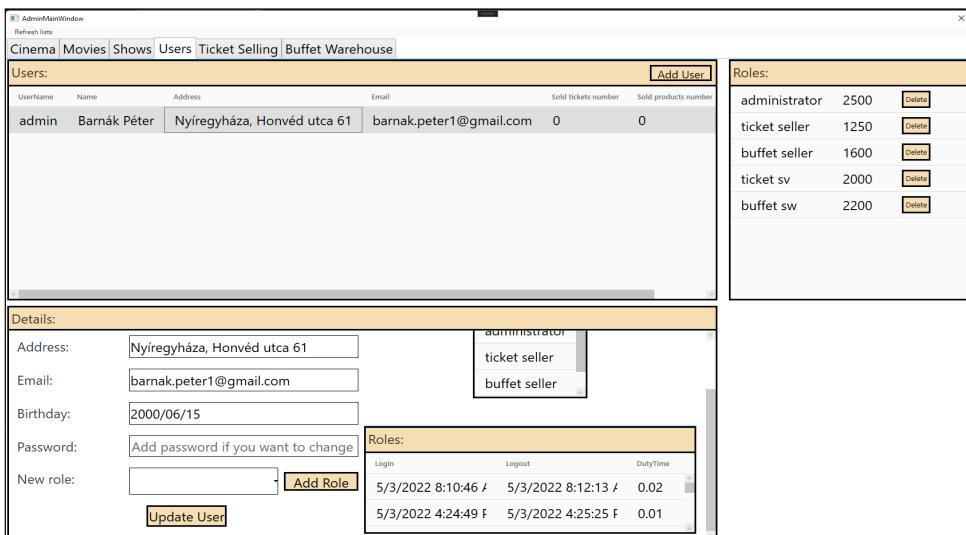
"Users" tab

Ezen az oldalon (2.28) az alkalmazottakat illetve az alkalmazáson belül használt szerepköröket láthatjuk listákba szervezve. Ezen listáknál is elérhetők a "Listák alap tulajdonságai" résznél leírt módosító műveletek. Egy kivételével, a szerepkörök listájához nem adhatóak hozzá újabbak. Pontosabban a definiált eljárással nem. Új szerepkör hozzáadása csak a felhasználó adatlapjáról tehető meg.



2.28. ábra. Az alkalmazottak felülete a szerepkörökkel

Kiválasztva egy felhasználót, nem csak az adatait szerkeszthetjük, hanem, mint a Filmek tab esetében, itt is két újabb lista lesz elérhető számunkra. Az elsőben az adott felhasználó szerepkörei láthatóak, míg a másikban jelenléti céllal kilistázásra kerülnek a felhasználó bejelentkezései és kijelentkezései, illetve a kettő között eltöltött idő órában számolva. Ezt a 2.29. ábra illusztrálja.



2.29. ábra. Egy felhasználó be- és kijelentkezési adatai a "Details" szekcióban

Mint azt már korábban említettem, a új szerepkört csak a felhasználók szerkesztő felületén adhatunk hozzá. Ezt a *New role* mellett található dropdown segítségével tehetjük meg. Ez a funkció is ugyanúgy működik, mint a filmek esetében a színészek és kategóriáknál láthattuk már korábban.

"Ticket Selling" tab

Ez az oldal szolgál a jegyek eladásáért (2.30. ábra). Pontosabban ez az előzmény oldala a jegyeladásoknak, hiszen az nem itt történik, itt csak ki tudjuk választani, hogy melyik előadásra szeretnénk jegyet értékesíteni.



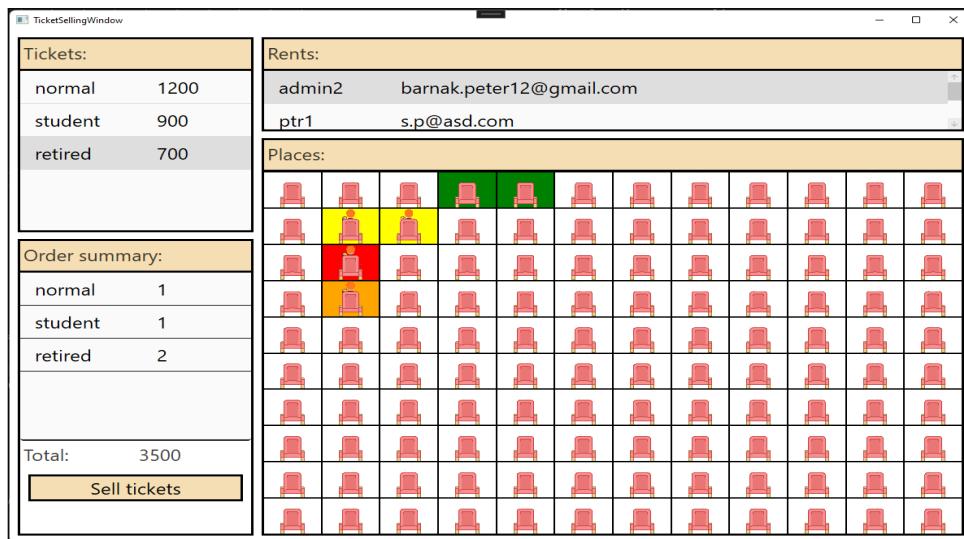
2.30. ábra. A jegyeladásokhoz tartozó előadásokat megjelenítő felület

Az oldal két részre osztható. Az első egy kereső sáv, ahol egy már megismert dropdown segítségével szűrhetjük a lejjebb található listánkat a kiválasztott filmre. Ezenkívül található még egy dátum választó mező is, mellyel kiválaszthatjuk, hogy mely nap előadásaira lennénk kíváncsiak.

A keresősáv alatt található egy oszlopokra rendezett lista, ahol az egyes oszlopok a termeket reprezentálják. Ezekben az oszlopokban pedig az adott teremhez tartozó előadások listája jelenítődik meg.

Ezen a tabon már nem érvényesek a "Listák alap tulajdonságai". Itt már szerkeszteni nem tudunk egyáltalán.

A jegyeladás, mint már említettem külön ablakban történik. Ahhoz, hogy ezt az ablakot elhozzuk, válasszunk ki egy előadást, majd kattintsunk rá duplán. Ekkor egy új ablakban láthatóvá válik a jegyeladás tényleges felülete, amely a 2.31. ábrán látható.



2.31. ábra. A jegyeladások tényleges felülete

Ez az ablak is több kis részre bontható, nézzük is őket így a könnyebb bemutatás kedvéért.

A bal felén az ablaknak két lista látható. A felsőben az aktuális jegytípusok elérhetőek, a jegyeladáshoz innen tudjuk majd kiválasztani a kívánt jegytípust. Az alsó részben pedig az egyes jegytípusok mellett a belőlük kiválasztott darabszám jelenik meg.

A jobb oldal is szintén két részre osztható. A felső területen egy listába szedve azon személyek nevét láthatjuk akik foglaltak az előadásra. Egy foglalást úgy tudunk értékesíteni, hogy az adott felhasználó nevét kiválasztjuk a listából, ekkor a foglalt helyek, illetve a foglalt helyek száma megjelenik a megfelelő részen a képernyőnél.

A helyek kiválasztását, illetve foglalás esetén megtekintését a jobb oldal alsó részében elhelyezett *Places* területen lehetjük meg. Ez egy NxM területű négyzet, ahol az N a terem sorainak számát az M pedig a terem oszlopainak számát jelöli. Ha egy helyet szeretnénk kiválasztani, először is válasszuk ki annak a jegytípusát, majd ezt követően válasszunk a négyzetrácsunkon egy nekünk tetsző helyet. Ekkor, mint a 2.31. ábra is mutatja, zöld színnel kijelölésre kerül az általunk kívánt hely, és a bal oldali *Order summary* nevezetű listában változik a jegytípus száma, illetve az eladás összege a *Total* címke mellett.

A négyzetrácsos felületen kétféle mezővel találkozhatunk, amint azt a 2.31 is mutatja. Egyszer láthatunk egy üres széket ábrázoló képet. Ez a kép azt jelöli, hogy erre a helyre még nem foglaltak jegyet. Ezt láthatjuk zöld háttérrel is, ez a szín jelöli, hogy éppen azt a széket jelöltük ki és szeretnénk eladni. Található még egy

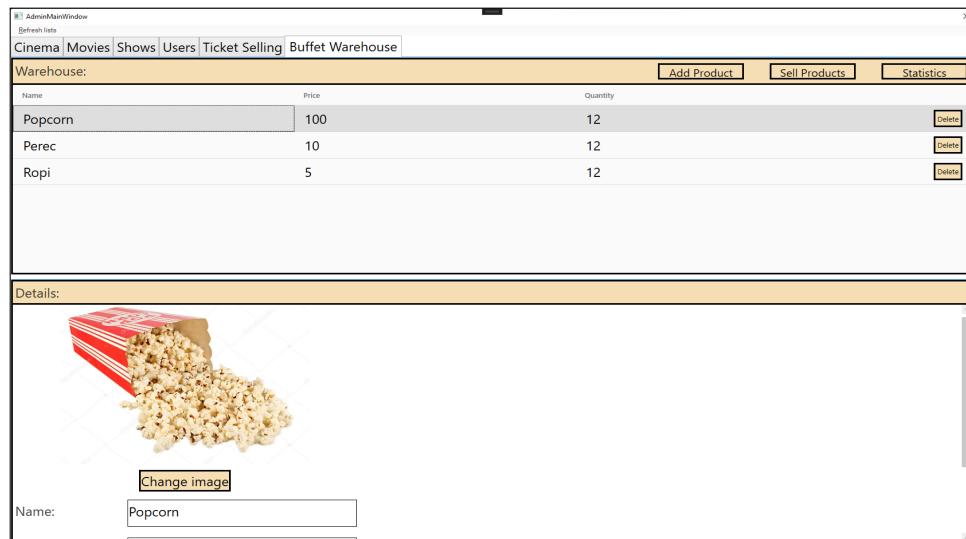
másik fajta kép is. Ez azt reprezentálja, hogy valaki már ül azon a helyen, azaz foglalt oda, vagy már megvette azt a helyet. Ha a kép háttere piros, akkor a jegy már eladott arra a helyre, ha narancssárga, akkor még csak foglalás szól oda. Ha citromsárga egy ilyen hely háttere, akkor az azt jelenti, hogy éppen azt a foglalást akarjuk szerkeszteni. Narancssárgával jelölt helyet nem adhatunk el, csak akkor, ha kiválasztjuk a foglalást.

Fenn áll a lehetősége, hogy egyszerre egy helyet több alkalmazott próbál meg eladni, ilyenkor a rendszer eladás közben *Bad request* hibaüzenettel jelez a felhasználók felé.

Ha a folyamat sikeres volt, a rendszer visszalép a *Ticket selling* tabra.

"Buffet Warehouse" tab

Ez az utolsó oldala az adminisztrátor felületünknek(2.32. ábra). Ezen az oldalon a moziban árult termékeket tudjuk szerkeszteni, eladni, illetve statisztikát tudunk az eladásairól lekérni.

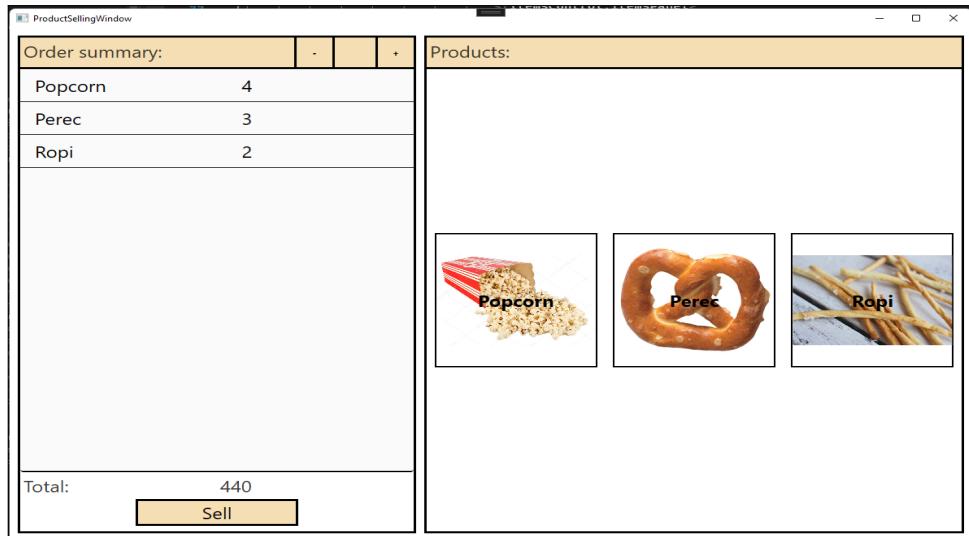


2.32. ábra. A raktárat reprezentáló felület

A termékek módosítása természetesen itt is a "Listák alap tulajdonságai"-ban leírtak szerint történik, nincs benne változás.

A termékek esetében is van lehetőségünk kép hozzáadására, csak úgy mint a filmek esetében. A funkció ugyanaz.

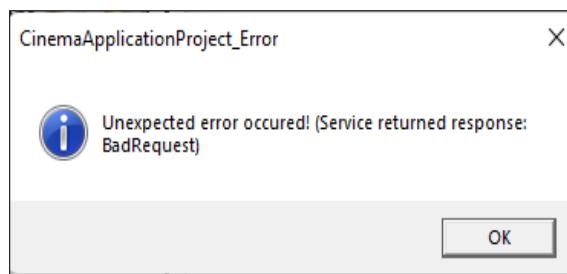
Mint már említettem, ezen az oldalon tudjuk a termékek eladását is kezelni, a *Sell Product* gomb segítségével. Erre a gombra kattintva jelenik meg a 2.33. ábrán látható felület.



2.33. ábra. A termékeket értékesítő felület

A felület itt is két részből áll. A bal oldali felén található egy lista, melyben az eladásra kívánt termékeket láthatjuk darabszámmal ellátva, azaz, hogy hány darabot szeretnénk ezekből a termékekből eladni. Ezek száma változtatható a lista felső részén található + és - gombokkal. Ha egy termék darabszáma eléri a 0-át, kikerül a listából. A lista alatt található egy összesítő címke (*Total*), amely mellett az eladásunk végösszegét láthatjuk. A termékek eladását a *Sell* gombbal tehetjük meg.

Ha több terméket szeretnénk eladni, mint amennyi a raktárban van, akkor a *Sell* gomb lenyomását követően hibaüzenetet kapunk a rendszertől, amely a 2.34. ábrán látható. Ezzel az üzenettel már a jegyeladásnál találkozhattunk.

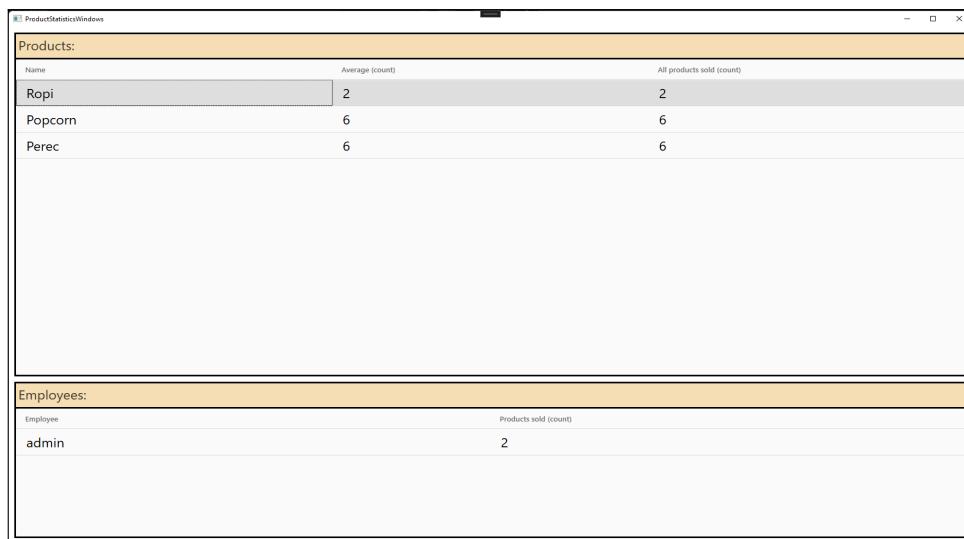


2.34. ábra. Hibaüzenet, ha a kért számú termék értékesítése nem lehetséges

A jobb oldali területen pedig egy kártyákból álló felületet láthatunk. minden egyes kártya egy terméket ábrázol, képekkel illetve a termék nevével megjelenítve. Ezen a felületen maximum három oszlop szerepelhet, ezzel is segítve a könnyebb átláthatóságot.

Az egyes kártyákat kiválasztva adhatunk hozzá a kosárhoz új termékeket. Ha a termék már szerepel a kosárba, akkor annak a darabszáma folyamatosan nő, természetesen, ezt a már említett + gombbal is megtehetjük.

Végezetül pedig tekintsük meg a termékekhez tartozó statisztikai felületet, amelyet a *Statistics* gombra kattintva érhetünk el, a *Buffet Warehouse* felületen. Ez a nézet is egy új ablakban nyílik meg, melyet a 2.35. ábrán láthatunk.



The screenshot shows a Windows-style application window titled 'ProductStatistics(Windows)'. It contains two main sections: 'Products:' and 'Employees:'.

Products:

Name	Average (count)	All products sold (count)
Ropi	2	2
Popcorn	6	6
Perec	6	6

Employees:

Employee	Products sold (count)
admin	2

2.35. ábra. A termékek statisztikáját bemutató ablak

Ezen a felületen is két listával találkozhatunk. A felsőben az eladott termékek jelennek meg, mellettük pedig, két szám olvasható. Az első azt mondja meg, hogy az adott termékből átlagosan hány darabot szoktak eladni vásárlásonként, míg a második pedig azt, hogy összesen hány darabot adtak el ebből a termékből eddig.

Lehetőségünk van kiválasztani is egy terméket a felső listából, ekkor az alsó listában megjelennek az ezt a terméket értékesítő alkalmazottak nevei, illetve a nevek mellett az is, hogy hány darabot adtak el az adott termékből.

2.3.5. További felületek

A könnyebb és gyorsabb áttekinthetőség érdekében a következő szerepkörökhez tartozó felületeket már nem mutatnám be egyesével külön-külön olyan részletességgel, mint eddig, mivel ezek funkcióiról már írtam korábban az *Admin felület* című bekezdésben. Viszont, hogy szó essen róluk, a következő párt bekezdésben ábrákkal mellékelve feltüntetném őket, csak hogy egy összefogottabb képet kaphassunk a teljes alkalmazásról.

A többi felületen már szerkesztési lehetőségeink nincsenek, tehát a "*Listák alap tulajdonságai*" részben tárgyalt műveletek ezekre a felületekre már nem érvényesek.

"Ticket Supervisor" tab

Ez a pénztár területi vezetők felülete. A felületen az *Admin* felületről két tab érhető el. Az egyik a "*User*" tab, a másik pedig a "*Ticket Selling*" tab. A "*Ticket Selling*" ugyanúgy működik, tehát a jegyértékesítésre szolgál. Viszont a *User* tab esetében van egy kis különbség a kettő között. Míg az admin felületen elérhetünk minden alkalmazottat, addig ebben az esetben, csak a pénztáros szerepkörrel rendelkezőket láthatjuk, és a listában csak a felhasználónevük, a teljes nevük jelenik meg, mellette pedig az, hogy az adott felhasználó hány jegyet adott már el.

A felületet a 2.36. ábra mutatja be.

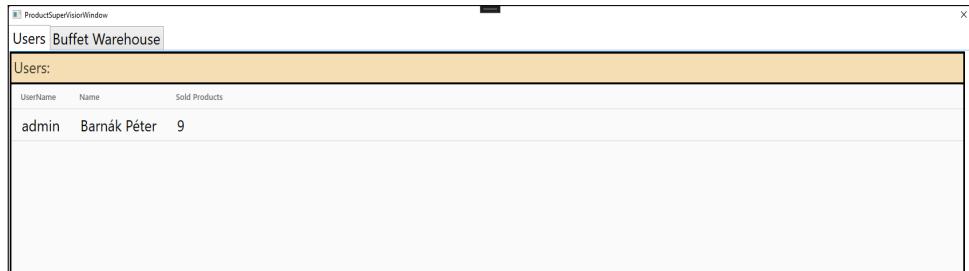
UserName	Name	Sold Tickets
admin	Barnák Péter	0

2.36. ábra. A pénztár területei vezető felületének főképernyője

2.3.6. "Product Supervisor" tab

Ez az ablak a büfé területi vezetőinek a felülete. Ők is, mint a "*Ticket Supervisor*"-ok, két tab-bal rendelkeznek. Számukra is elérhető egy *User* felület, ahol a büfé szerepkörrrel rendelkező alkalmazottak felhasználónevét, nevét, és az általuk eladott termékek számát láthatják. Illetve rendelkeznek egy "*Buffet Warehouse*" tab-bal, ahol, mint már az *Admin* felületnél említettem, a büfénben értékesített termékeket tudják kezelni, viszont az ő esetükben a "*Listák alap tulajdonságai*" részben definiált metódusok elérhetőek, tehát tudják szerkeszteni az értékesíteni kívánt termékek adatait. Erre azért van szükség, hiszen így ők is képesek tölni a raktárat, és ha kell esetleges hibákat tudnak ott javítani. A "*Buffet Warehouse*" tab korábbi bemutatásánál említve lett egy statisztikai funkció is, ez ennél a szerepkörnél nem elérhető.

A szerepkörhöz tartozó felületet a 2.37. ábra mutatja be.



2.37. ábra. A büfé területi vezetőjének főképernyője

2.3.7. "Product Seller" tab

Ezen a felület lényegében a "*Buffet Warehouse*" tabon leírt *Sell Products* felület. Az ehhez a szerepkörhöz tartozó alkalmazottak feladata a büfén történő termékek értékesítése, így nekik csak ehhez az ablakhoz van hozzáférésük.

A *Sell Product* funkcióhoz tartozó kép a 28. oldalon található.

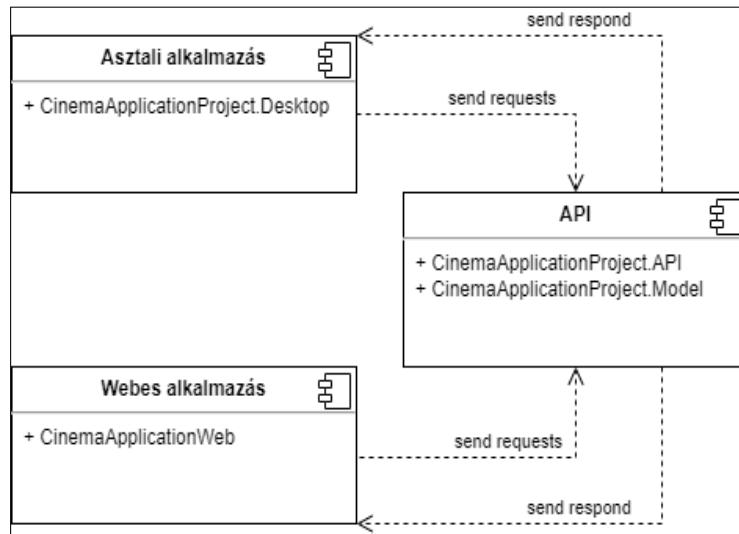
2.3.8. "Ticket Seller" tab

Ehhez a felülethez hozzáférő alkalmazottak dolga a jegyek eladása, tehát ők az *Admin* felület *Ticket Selling* tab funkcióihoz férnek hozzá. Természetesen ezzel együtt, elérhetővé válik a jegyeladáshoz tartozó külön ablak is, hiszen itt tudnak egy adott előadáshoz jegyeket árulni.

Ehhez a funkcióhoz tartozó képet, leírással együtt a 26.oldalon találjuk.

3. fejezet

Fejlesztői dokumentáció



3.1. ábra. Az alkalmazások kommunikációja

Ebben a fejezetben a fejlesztési folyamatokról, választott technológiáról, illetve megoldásokról lesz szó, immár fejlesztői szemmel nézve. Mivel a szakdolgozatban nem csupán egy alkalmazás lett megvalósítva, hanem egyszerre három egységet fogal magába, így ezeket a könnyebb áttekintés miatt külön szekciókban szeretném egyesével bemutatni. Ez a három egység nem más, mint a felhasználói dokumentációban már bemutatott webes illetve asztali alkalmazás, és az ezek mögött az üzleti logikát végző *REST API* [4]. Kezdetben tekintsük át a választott technológiákat.

3.1. A szakdolgozatban használt technológiák

A szakdolgozatomban elkészített alkalmazások közül a webes felület Javascript [5] nyelven *Vue.js* [6] keretrendszer felhasználásával készült, míg az asztali alkalmazás C# [7] nyelven íródott. Az ezeket kiszolgáló *REST API* szintén C#-ban készült, viszont ebben az esetben az *ASP.NET Core* [8] keretrendszer felhasználva.

A *Vue.js* a szabványos HTML [9], CSS és Javascript alapokra épül, és egy komponensalapú programozási modelljének köszönhetően segít a felhasználói felületek hatékony fejlesztésében. A komponensalapú programozási modell lehetővé teszi számunkra, hogy egy adott részét a weboldalunknak (pl egy űrlapot, vagy egy kereső sávot) elegendő legyen egyszer definiálnunk, majd azt az oldalunkon belül bárminnyiszer fel tudjuk használni kódismétlés nélkül.

Az *ASP.NET Core* egy cross-platform keretrendszer (platformfüggetlen), az *ASP.NET* [10] utódja. Elterjedt moduláris webfejlesztő keretrendszer, mely segítségével gyorsan és könnyen készíthetőek webalkalmazások, vagy akár alkalmazás-programozási felületek (angolul application programming interface azaz *API*) is. Felhasználásakor egy manapság sokat alkalmazott architektúrát, az *MVC*-t [11] választottam.

Az *MVC* (model-view-controller) architektúra lényege, hogy az egyes kliensek által küldött HTTP kérések egy vezérlőhöz futnak be, amelyek metódusokon keresztül elérlik az előre definiált üzleti logikát, azaz a modellt, ami pedig a végrehajtása eredményét visszaküldi a vezérlőnek. Ekkor a vezérlő az eredményekről létrehoz egy adott nézetet, amelyet visszaküld az őt meghívó kliensnek. A szakdolgozatomban az *API* egy webszolgáltatás, ezért itt a nézet mint olyan nincs kihasználva, a válasz egyszerű *JSON* [12] (kis méretű, ember által olvasható adatcserére használt formátum) formában történik.

Az *MVC* architektúrán kívül egy másik architekturális megvalósítás is használatba került a szakdolgozat elkészülésénél. Ez az MVVM [13] architektúra, amelyet az asztali alkalmazás fejlesztésénél vettet igénybe. Lényege, hogy az alkalmazás fejlesztői szemszögből három külön részre különül el. Ezek a részek a Modell, a Nézet és a Nézetmodell (az architektúra neve is ezen szavak angol verzióból tevődik össze, Model-View-Viewmodel). A nézet feladata a megjelenítés, a modellé pedig az üzleti logika megvalósítása. A kettő közé ékelődik be a nézetmodell, amely kommunikál az üzleti logikával, a tőlük megkapott értékeket pedig megjeleníthető formátummá ala-

kítja a nézet számára. Például objektumokat inicializál, listákat hoz létre amelyeket a nézet megjelenít. A nézet a nézetmodelltől az objektumokat, listákat, úgynévezett adatkötéseken (DataBinding) keresztül kapja meg, illetve utasításokkal kommunikál azzal. A nézetmodell metódus hívásokon keresztül kéri el az adatokat a modelltől, az pedig eseményeken (event) keresztül közli a módosításait a nézetmodellel. Az architektúra segít abban, hogy egy jól konstruált programot kapunk, viszont ennek hátránya, hogy összetettsége miatt a tervezési fázis több időt vesz időbe.

Az adatbázis létrehozásánál egy olyan keretrendszer használatát vett igénybe, amely segítségével *SQL* [14] parancsok nélkül, C# nyelven írt utasítások segítségével készíthetem el az adatbázisaimat. Ez a keretrendszer az *Entity Framework Core* [15] (röviden EF). Ez az *Entity Framework* [16] alapjaira épül, viszont azzal ellentétben ez már egy platformfüggetlen keretrendszer. Az *Entity Framework Core* kitűnően használható webes alkalmazásoknál illetve webszolgáltatásoknál is az *ASP.NET Core* keretrendszerrel együtt. Az EF segítségével az adatbázis táblákat entitás osztályokkal tudjuk létrehozni. Az adatbázisunkat két féle módon valósíthatjuk meg. Az első a Code-first mód. Lényege, hogy az adatbázis az entitás osztályokból generálódik le, tehát ezeket az entitásokat készítjük el először. A másik megvalósítási forma a Database-first megvalósítás. Itt az entitás osztályok az adatbázisból generálódnak le. A szakdolgozatomban én a Code-first megközelítést választottam.

3.2. Fejlesztéshez használt eszközök

A fejlesztési folyamat egésze Windows 11-es operációs rendszeren történt.

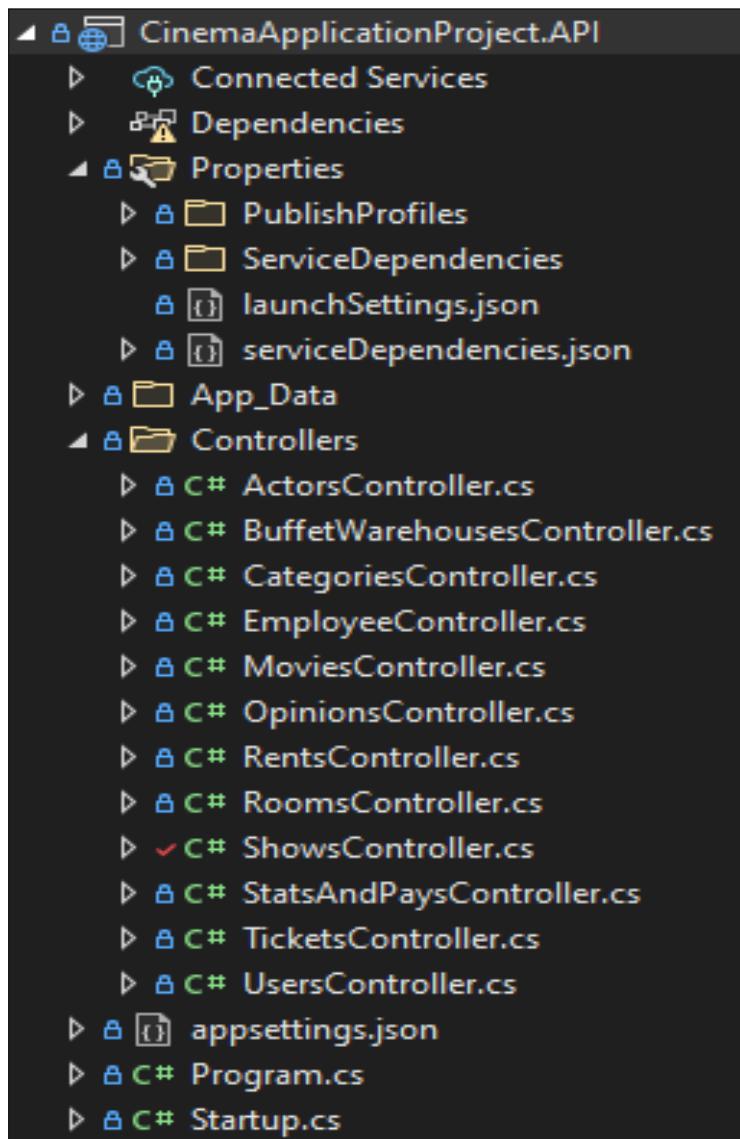
Az alkalmazás fejlesztéséhez különböző eszközöket vettettem segítségül. Az első ilyen eszköz a *Visual Studio Community 2022* [17], amelyben egy *solution*-ben (több projekteket összefogó projekt) készült el az asztali alkalmazás, illetve az *API* (ehhez tartozóan a modell is illetve a teszterei is).

A második fejlesztő eszköz a JetBrains által kiadott WebStorm 2021.2.3 [18] webfejlesztő felület. Mint a neve is utal rá, ez az a program amelyben a webes felület fejlesztése zajlott.

Ahhoz, hogy a fejlesztés verzió kezelése megfelelően le legyen kezelve, akkor is, ha valamilyen meghibásodás történik a fejlesztő számítógépen a *Github* [19] felületet alkalmaztam.

A fejlesztés során az *API* gyors és egyszerű tesztelését a *Postman* [20] alkalmazáson keresztül tettem, ahol pillanatok alatt tudtam egyszerű HTTP kéréseket küldeni az *API*-nak, ezzel ellenőrizve az interfész megfelelő működését.

3.3. Webszolgáltatás (ASP.NET Core WebApi) bemutatása



3.2. ábra. Az API architektúrája

Kezdetben az *API* bemutatásával szeretném kezdeni, mivel lényegében ez az a szoftver, amely az asztali illetve webes alkalmazás számára biztosít kommunikációt az adatbázissal, így ő az aki a teljes üzleti logikát végzi. A legtöbb fel-

adata neki van, emiatt a hozzátartozó kód a legösszetettebb mind közül. A megvalósításához a Microsoft által kiadott *ASP.NET Core* keretrendszer választottam, és az egyszerűbb fejlesztés érdekében a kódot Visual Studioban készítettem el. Az *API* két részből áll, egyszer a tényleges REST interfésből, másszor pedig egy adatbázist leíró modellből. Mindkettőt a Visual Studio segítségével egy *Solution*-ben készítettem el, viszont külön projektként, ezáltal biztosítva azt, hogy az esetleges továbbfejlesztésnél a kód könnyen és hamar, egy helyről elérhető legyen. Az *API* projektje a CinemaApplicationProject.API névre, míg a modellé a CinemaApplicationProject.Model névre hallgat a *Solution*-ön belül. Az adatbázis leírására szolgáló modell megvalósításáról kicsit később esik majd szó, először az *API* működésére térnék ki.

Ahhoz, hogy az alkalmazásprogramozási felületünk megfelelően elinduljon azért a projekten belül található *Startup.cs* állomány felel. Ennek az állománynak a feladata az *API* indításakor való helyes beállítása. Kezdetben az adatbázis leírását tartalmazó kontextus fájlt adhatjuk meg. Az 3.6. ábrán ez a fájl *DatabaseContext* néven szerepel. Ezen fájl feladata az adatbázisban található táblák regisztrálása a hozzájuk tartozó entitás modellekben keresztül. Nem csak az egyes táblákat adhatjuk meg benne, hanem azok tulajdonságait is beállíthatjuk itt. Megadhatjuk, hogy milyen kulcsok szerepelnek az adott táblában, milyen néven jelenjen meg az adatbázisban a tábla és ezeken kívül rengeteg más tulajdonságot is testre szabhatunk ebben az állományban. Az adatbázis kontextus megadása után definiálnunk kell az adatbázis típusát is, amelyet egy *ConnectionString* segítségével tettem meg az alkalmazáson belül. Az általam választott adatbázis típus az *MSSQL* lett. Ahhoz, hogy az adatbázist megfelelően beállítsa az *API*, szükség volt egy ezt a célt szolgáló NuGet Package [21] telepítésére is a projekten belül.

A *NuGet Package*-ek lényegében előre elkészített "programok", amelyek egy-egy funkcionálitás végrehajtására hivatottak. Az alkalmazás megfelelő működése miatt a későbbiekbén még találkozni fogunk ezekkel. Telepítésük rendkívül egyszerű. A Visual Studio felületén belül elegendő csak a kiválasztott projektre kattintanunk jobb kíkkel, majd a megjelenő menüben a "*Manage NuGet Packages*" menüpontot kiválasztva már hozzá is adhatjuk a szükséges csomagokat a projektünkhez. Elegendő a nevét beírnunk a keresősávba, majd a megfelelő .NET verzió kiválasztása után, már telepíthetjük is a kívánt package-ét. Az alkalmazás .NET 6.0-ás verzióban készült, ezért a *NuGet Package*-ek ennek a verziónak megfelelően lettek kiválasztva.

Kanyarodjunk vissza a *Startup.cs* állományunkra. Az adatbázis kontextust függőségi befecskendezéssel (angolul dependency injection) adtuk meg. A függőségi befecskendezés lényege, hogy egy adott objektumot úgy hozunk létre, hogy a kliens objektum létrehozásánál a szolgáltató objektumot beinjectáljuk (tehát nem egy konkrét példányt adunk át), ezáltal biztosítva azt, hogy a kliens nem fog tudni a szolgálató objektum metódusainak megvalósításáról, csakis azokhoz a metódusokhoz tartozó interfész fogja ismerni.

Az adatbázis felállítása után a *CORS* (Cross-Origin Resource Sharing) beállítása következik. A *CORS* lényege, hogy két különböző domain-en lévő alkalmazás között engedi meg a kommunikációt HTTP kérések esetén. Erre azért van szükség, mivel a böngészők egy olyan biztonsági protokollt használnak, amely nem engedi meg, hogy más domain-en lévő webszolgáltatásuktól (más néven API) kérjenek le adatokat. A CORS ezen problémára biztosít megoldást, mégpedig úgy, hogy egy teljesen biztonságos hálózaton engedi meg a két különböző domain-en lévő kliens számára a kommunikációt. A *CORS* beállításáért az alkalmazásban a *Microsoft.AspNet.WebApi.Cors NuGet Package* felel. A package felhasználásával saját *CORS* szabályt definiáltam, amelyben a feltüntettem az összes lehetséges címet, amelyről kérés érkezhet az *API* felé.

A *Startup.cs* állományban lett az adatbázis lekérdezéseket tartalmazó szolgáltatás is beállítva. Ez a 3.6 ábrán *DatabaseService.cs* néven található. Ebben az állományban lett definiálva minden adatbázis lekérdező vagy módosító művelet. Ezek a műveletek a *LINQ* (Language-Integrated Query) osztály segítségével készültek el. Ez az osztály biztosítja számunkra azt, hogy bonyolult *SQL* lekérdezések helyett egyszerűbb C# nyelven megírt utasításokkal tudunk adatbázis lekérdező műveleteket végrehajtani.

A *DatabaseService* szolgáltatás egy leszármazott osztály. Szülvőként definiálva lett számára egy interfész, azért, hogy a *Startup.cs* állományban a függőségi befecskendezés segítségével tudjuk meghatározni ezt a szolgáltatást. Egy ilyen szolgáltatást három különböző módon tudjuk felhasználni az *API*-n belül. Lehet *Transient*, *Scoped* és *Singleton*. A *Transient* esetben minden kontrollerhez szolgáltatás objektum társul, *Scoped* esetben minden kérés esetén új objektum jön létre, míg a *Singleton* megvalósításánál ugyanazt az objektumot használja minden kérés esetében. Én a szakdolgozatomban a *Transient* megvalósítást választottam. A megvalósításkor még függőségi befecskendezéssel meg kell adnunk a szülő interfész, illetve egy konkrét

megvalósítását az interfésznek. Természetesen ilyenkor a program a későbbi felhasználáskor csak az interfést szolgáltatja majd számunkra, tehát a tényleges megvalósítás nem lesz elérhető. A megvalósítás csak a *Transient* objektum létrehozásához szükséges.

Ezt követően minden készen áll arra, hogy a webszolgáltatásunk működőképes legyen. Indítás után az *API* által nyújtott funkciók kontrollerek endpoint-jain (magyarul végpontok) keresztül érhetőek el. A következőkben ezeket az endpoint-okat ismertetném kontrollerek szerint csoportosítva. Az egyes kontrollerek neve utal arra, hogy milyen funkcionalitás érhető el rajtuk keresztül.

3.3.1. Kontrollerek bemutatása

Ebben a részben az *API*-ban definiált kontrollerek működését nézzük át, és azok végpontjait mutatom be táblázatos formában. A táblázatok első oszlopa minden esetben azt fogja bemutatni, hogy milyen HTTP kéréssel érhetőek el az egyes végpontok. Ezt követően a következő oszlopban lesz a konkrét elérési útja leírva az adott kérésnek. Majd az adott endpoint meghívásával lefutó metódus neve olvasható. Az utolsó oszlopból pedig minden esetben egy részletesebb leírást mellékelek a metódus működéséről.

ActorsController

Az adott kontroller felel a színészekkel kapcsolatos műveletekért.

Az endpoint-jait az 3.1 táblázat tartalmazza.

HTTP kérés	Elérés	Metódus	Leírás
GET	api/Actors	GetActors()	Visszaadja az adatbázisban található színészek listáját.
GET	api/Actors/{id}	GetActor(id)	Visszaadja a megadott ID-val rendelkező színészt.
POST	api/Actors	PostActros(actor)	A paraméterül kapott színészt, hozzáadja az adatbázishoz, ha az még nem szerepel benne. Ezt követően, összeköti a paraméterül kapott objektumban található filmmel a színészt.
DELETE	api/Actors/{movieId}/{id}	DeleteActor(movieId,id)	Két paraméterrel rendelkezik, az egyik egy film azonosítója, míg a másik egy választott színészé. A megadott film színészei közül eltávolítja a megadott színészt. Az adatbázisból teljesen nem törlődik a színész.

3.1. táblázat. Az ActorsController végpontjainak leírása

BuffetWarehousesController

Az adott kontroller biztosítja a büfé raktárához való hozzáférést.

Az endpointjait a 3.2 táblázat tartalmazza.

HTTP kérés	Elérés	Metódus	Leírás
GET	api/BuffetWarehouses	GetBuffet-Warehouse()	Visszaadja az raktárban található összes terméket mennyiséggel és árral együtt.

HTTP kérés	Elérés	Metódus	Leírás
GET	api/BuffetWarehouses/{id}	GetStat ()	Visszaadja a büfében értékesített termékeket és azokról készített statisztikai adatokat (pl melyik termékből ki mennyit értékesített)
GET	api/BuffetWarehouses/statistics	GetStat ()	A paraméterül kapott színészt, hozzáadja az adatbázishoz, ha az még nem szerepel benne. Ezt követően, összeköti a paraméterül kapott objektumban található filmmel a színészt.
PUT	api/BuffetWarehouses/{id}	PutProduct (id,product)	Két paraméterrel rendelkezik, az első a módosítandó termék azonosítója, a másik pedig maga a termék. A termék esetében minden adat továbbításra kerül, a rendszer az ID alapján összehasonlítja az adatbázisban lévő elemmel a paraméterül kapott elemet, és a változtatásokat elmenti.
Post	api/BuffetWarehouses	PostProduct (product)	A paraméterül kapott terméket hozzáadja az adatbázishoz, ha az még nem szerepel benne.

HTTP kérés	Elérés	Metódus	Leírás
Post	api/BuffetWarehouses /sell	SellProduct (products)	Paraméterül kap egy termékeket tartalmazó listát. minden elemet a paraméterről kapott listában megvizsgál, ha van elegendő termék a raktárban, akkor levonja a megadott darabszámot és elmenti az adott eladási műveletet az eladásokat kezelő táblába. Ha nincs annyi termék mint amennyit el szeretnénk adni, arról a rendszer <i>Bad request</i> hibaüzenettel tájékoztat.
Delete	api/BuffetWarehouses /{id}	DeleteProduct (id)	Kitörli az adatbázisból a megadott ID-val rendelkező terméket.

3.2. táblázat. Az BuffetWarehouseController végpontjainak leírása

CategoriesController

Az adott kontroller felel a kategóriákkal kapcsolatos műveletekért.

Az endpointjait a 3.3 táblázat tartalmazza.

HTTP kérés	Elérés	Metódus	Leírás
GET	api/Categories	GetCategories()	Visszaadja az adatbázisban található kategóriák listáját.
GET	api/Categories/{id}	GetCategoryById(id)	Visszaadja a megadott ID-val rendelkező kategóriát.
POST	api/Categories	PostCategory(category)	A paraméterül kapott kategóriát, hozzáadja az adatbázishoz, ha az még nem szerepel benne. Ezt követően, összeköti a paraméterül kapott objektumban található filmmel a kategóriát.
DELETE	api/Categories/{movieId}/{id}	DeleteCategory(movieId,id)	Két paraméterrel rendelkezik, az egyik egy film azonosítója, míg a másik egy válaszott kategóriáé. A film kategóriái közül eltávolítja a megadott kategóriát. Az adatbázisból teljesen nem törlődik a kategória.

3.3. táblázat. Az ActorsController végpontjainak leírása

EmployeeController

Az adott kontroller biztosítja az alkalmazottakkal kapcsolatos műveleteket.

Az endpointjait a 3.4 táblázat tartalmazza.

HTTP kérés	Elérés	Metódus	Leírás
GET	api/Employee	GetEmployees()	Visszaadja az adatbázisban szereplő összes alkalmazot-tat.

HTTP kérés	Elérés	Metódus	Leírás
GET	api/Employee /{role}	GetEmployeesByRole (role)	Paraméterül egy szerepkör kapott azonosítóját, és visszaadja az ahhoz tartozó összes alkalmazottat.
POST	api/Employee /Login	Login (login)	A paramétere egy felhasználónévet és jelszót tartalmazó objektum, ami egy alkalmazottat reprezentál. A megadott alkalmazottat bejelentkezteti a rendszerbe.
POST	api/Employee /Logout /{id}	Logout(id)	Paraméterként megkapja egy felhasználó azonosítóját, és azzal az azonosítóval rendelkező felhasználót kijelentkezteti.
GET	api/Employee /roles /{id}	GetRoles (id)	A paraméterül kapott ID-val rendelkező felhasználó szerepköreit adja vissza
POST	api/Employee	PostUser (newUser)	Paraméterül egy új felhasználó adatait kapja meg egy objektumban. Ezt a felhasználót felveszi az adatbázisba. A felhasználóhoz tartozó szerepköröket is beállítja.

HTTP kérés	Elérés	Metódus	Leírás
PUT	api/Employee/{id}	PutUser (id,employee)	Két paraméterrel rendelkezik, az első a módosítandó alkalmazott azonosítója, a másik pedig maga az módosítandó alkalmazott. Az alkalmazott minden adata továbbításra kerül, a rendszer az ID alapján összehasonlítja az adatbázisban lévő elemmel a paraméterül kapott elemet, és a változtatásokat elmenti.
DELETE	api/Employee/{id}		A paraméterként megkapott ID-val rendelkező alkalmazottat kitörli az adatbázisból.

3.4. táblázat. Az EmployeeController végpontjainak leírása

MoviesController

Az adott kontroller a filmekkel kapcsolatos műveletek hozzáférését biztosítja.

Az endpointjait a 3.5 táblázat tartalmazza.

HTTP kérés	Elérés	Metódus	Leírás
GET	api/Movies	GetMovies()	Visszaadja az adatbázisban szereplő összes filmet.

HTTP kérés	Elérés	Metódus	Leírás
GET	api/Movies/today	GetTodaysMovies()	Visszaadja a mai naphoz tartozó filmeket.
GET	api/Movies/statistics	GetStatistics()	Az adatbázisban található filmekről statisztikát készít és ezzel tér vissza. Ez a statisztika tartalmazza egy film nézettségi számait, illetve véleményeit is.
GET	api/Movies/{title}	GetMoviesByTitlePart(title)	A paramétere egy film címének részlete. Azokat a filmeket adja vissza, melyek címe az alábbi szövegrészlettel kezdődik.
GET	api/Movies/title/{category}	GetMoviesByCategory(category)	Paraméterül egy kategória nevét kapja meg, és az ahhoz tartozó filmeket adja vissza.

HTTP kérés	Elérés	Metódus	Leírás
PUT	api/Movies/{id}	PutMovie (id, movie)	Két paraméterrel rendelkezik, az első a módosítandó film azonosítója, a másik pedig maga a módosítandó film. A film minden adata továbbításra kerül, a rendszer az ID alapján összehasonlítja az adatbázisban lévő elemmel a paraméterül kapott elemet, és a változtatásokat elmenti.
POST	api/Movies	PostMovie (movie)	Paraméterül egy új film adatait kapja meg egy objektumban. Ezt a filmet felveszi az adatbázisban.
DELETE	api/Movies/{id}	DeleteMovie (id)	A paraméterként megkapott ID-val rendelkező filmet kitörli az adatbázisból.

3.5. táblázat. Az MoviesController végpontjainak leírása

OpinionsController

Az adott kontroller a vélemények műveleteit tartalmazza.

Az endpoint-jait a 3.6 táblázat tartalmazza.

HTTP kérés	Elérés	Metódus	Leírás
GET	api/Opinions {id}	GetOpinions (id)	Paraméterül vár egy film azonosítót, és visszaadja az ahhoz tartozó összes véleményt.
POST	api/Opinions	PostOpinion (opinion)	Paraméterül egy új film véleményt kapja meg egy objektumban. Ezt az elemet felveszi az adatbázisba.

3.6. táblázat. Az OpinionsController végpontjainak leírása

RentsController

Az adott kontroller felel a foglalásokkal kapcsolatos műveletekért.

Az endpoint-jait az 3.7 táblázat tartalmazza.

HTTP kérés	Elérés	Metódus	Leírás
GET	api/Rents {id}	GetRentsById (id)	Paraméterül vár egy előadás azonosítót, és visszaadja az ahhoz tartozó összes foglalást.

HTTP kérés	Elérés	Metódus	Leírás
GET	api/Rents/sell/{id}	GetRentsForSellById (id)	Paraméterül vár egy eladás azonosítót, és visszaadja az ahhoz tartozó összes foglalást. Az előző endpoint-tal ellentétben ez a foglalásokhoz tartozóan pl jegytípust is visszaad, ami az értékesítésnél kell.
GET	api/Rents/sellUsers /{id}	GetRentsForSellById (id)	Paraméterül vár egy eladás azonosítót, és visszaadja az ahhoz tartozó összes foglaló felhasználó adatait.
POST	api/Rents	PostRent(rent)	Paraméterül egy új foglalás adatait kapja meg egy objektumban. Ezt a foglalást rögzíti az adatbázisban.

3.7. táblázat. Az RentsController végpontjainak leírása

RoomsController

Az adott kontroller feladata a termekkel kapcsolatos műveletek kiszolgálása.

Az endpoint-jait a 3.8 táblázat tartalmazza.

HTTP kérés	Elérés	Metódus	Leírás
GET	api/Rooms	GetRooms()	Az adatbázisban szereplő minden termet visszaad.
GET	api/Rooms/{id}	GetRoom(id)	Visszatér a paraméterül kapott azonosítóval rendelkező teremmel.
PUT	api/Rooms/{id}	PutRoom (id)	Két paraméterrel rendelkezik, az első a módosítandó terem azonosítója, a másik pedig maga az módosítandó terem. A terem minden adata továbbításra kerül, a rendszer az ID alapján összehasonlítja az adatbázisban lévő elemmel a paraméterül kapott elemet, és a változtatásokat elmenti.
POST	api/Rooms	PostRoom(room)	Paraméterül egy új terem adatait kapja meg egy objektumban. Ezt a termet felveszi az adatbázisba.
DELETE	api/Rooms/{id}	DeleteRoom (id)	A paraméterként megkapott ID-val rendelkező termet körli az adatbázisból.

3.8. táblázat. Az RoomsController végpontjainak leírása

ShowsController

Az adott kontroller feladata az előadásokkal kapcsolatos műveletek kiszolgálása.

Az endpoint-jait a 3.9 táblázat tartalmazza.

HTTP kérés	Elérés	Metódus	Leírás
GET	api/Shows	GetShows()	Az adatbázisban szereplő minden előadást visszaad.
GET	api/Shows/{date}	GetShowsByDate(date)	Paraméterül vár egy dátumot, és a megadott dátumú előadásokat visszaadja a rendszer.
GET	api/Shows/show/{id}	GetShowById(id)	Paraméterül egy azonosítót vár, és visszaadja a megadott azonosítóval.
GET	api/Shows/available-Dates	GetAvailableDates()	Dátumokat ad vissza, mégpedig azon napok dátumait, amelyeken van aktív előadás.
GET	api/Shows/today	GetTodaysShows()	Az aktuális nap előadásait adja vissza.

HTTP kérés	Elérés	Metódus	Leírás
PUT	api/Shows/{id}	PutShow (id,show)	Két paraméterrel rendelkezik, az első a módosítandó előadás azonosítója, a másik pedig maga az módosítandó előadás. Az előadás minden adata továbbításra kerül, a rendszer az ID alapján összehasonlítja az adatbázisban lévő elemmel a paraméterül kapott elemet, és a változtatásokat elmenti.
POST	api/Shows	PostShow(show)	Paraméterül egy új előadás adatait kapja meg egy objektumban. Ezt a termet felveszi az adatbázisban.
DELETE	api/Shows/{id}	DeleteShow (id)	A paraméterként megkapott ID-val rendelkező termet kitörli az adatbázisból.

3.9. táblázat. Az ShowsController végpontjainak leírása

StatsAndPaysController

Az adott kontroller a szerepkörök műveleteit biztosítja.

Az endpoint-jait a 3.10 táblázat tartalmazza.

HTTP kérés	Elérés	Metódus	Leírás
GET	api/StatsAndPays	GetRoles()	Az adatbázisban szereplő minden szerepkört visszaad.
GET	api/StatsAndPays/{id}	GetStatById (id)	Visszaadja a megadott ID-val rendelkező szerepkört.
PUT	api/StatsAndPays/{id}	PutRole (id,show)	Két paraméterrel rendelkezik, az első a módosítandó szerepkör azonosítója, a másik pedig maga az módosítandó szerepkör. Az szerepkör minden adata továbbításra kerül, a rendszer az ID alapján összehasonlítja az adatbázisban lévő elemmel a paraméterül kapott elemet, és a változtatásokat elmenti.
POST	api/StatsAndPays	PostRole (show)	Paraméterül egy új előadás adatait kapja meg egy objektumban. Ezt a termet felveszi az adatbázisba.
DELETE	api/StatsAndPays/{id}	DeleteRole (id)	A paraméterként megkapott ID-val rendelkező termet kitörli az adatbázisból.

3.10. táblázat. Az StatsAndPaysController végpontjainak leírása

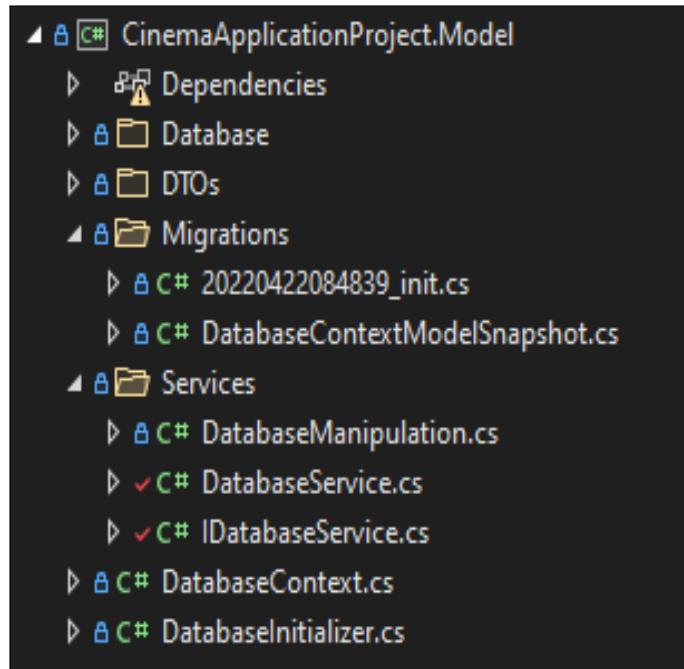
TicketsController

Az adott kontroller a jegytípusok műveleteit biztosítja. Az endpoint-jait a 3.11 táblázat tartalmazza.

HTTP kérés	Elérés	Metódus	Leírás
GET	api/Tickets	GetTickets()	Az adatbázisban szereplő minden jegytípust visszaad.
GET	api/Tickets/{id}	GetTicketById (id)	Visszaadja a megadott ID-val rendelkező jegytípust.
PUT	api/Tickets/{id}	PutTickets (id,ticket)	Két paraméterrel rendelkezik, az első a módosítandó jegytípus azonosítója, a másik pedig maga az módosítandó jegytípus. Az jegytípus minden adata továbbításra kerül, a rendszer az ID alapján összehasonlítja az adatbázisban lévő elemmel a paraméterül kapott elemet, és a változtatásokat elmenti.
POST	api/Tickets	PostTickets (ticket)	Paraméterül egy új jegytípus adatait kapja meg egy objektumban. Ezt felveszi az adatbázisba.
DELETE	api/Tickets/{id}	DeleteTickets (id)	A paraméterként megkapott ID-val rendelkező szerepkört kitörli az adatbázisból.

3.11. táblázat. A TicketsController végpontjainak leírása

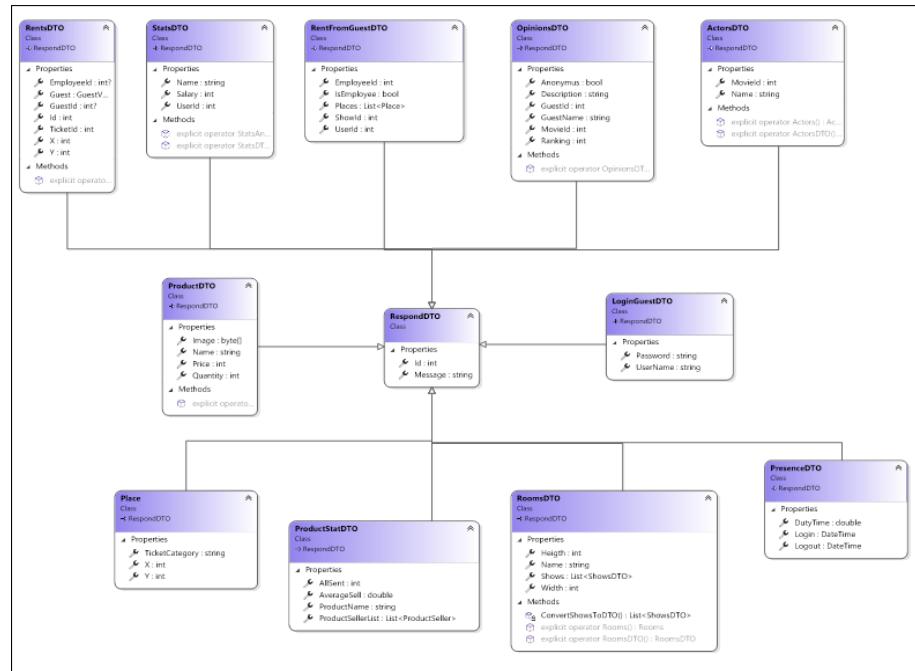
3.3.2. A modell bemutatása



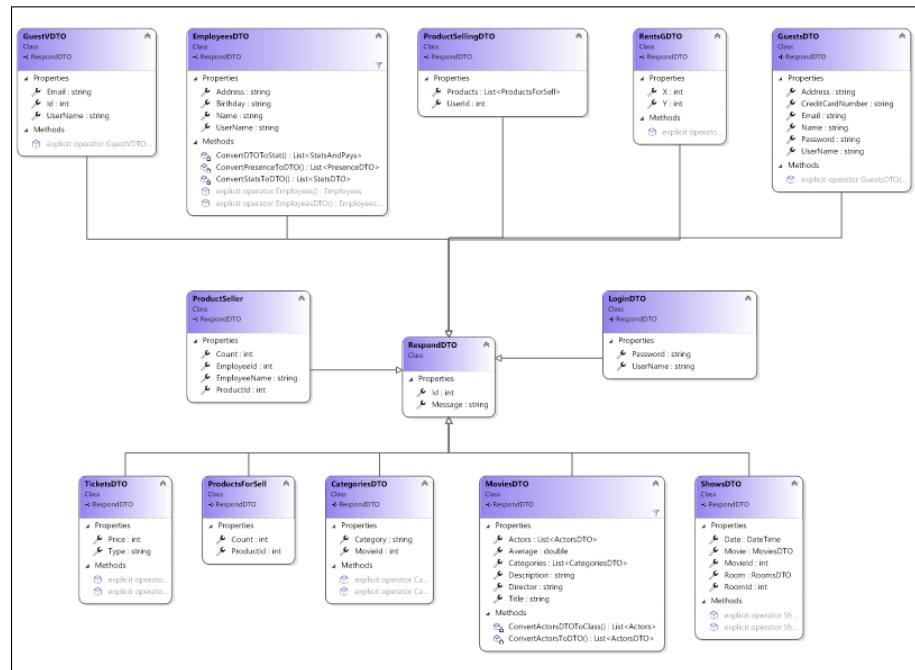
3.3. ábra. A modell architektúrája

Ebben a szekcióban a már említett *CinemaApplicationProject.Model* projekt-ről lesz szó. Ez a projekt tartalmazza az adatbázis létrehozásához szükséges entitás modelleket, DTO [22] objektumokat, illetve a már említett DatabaseService.cs állományt és az ő interfészét definiáló IDatabaseService. Az API indításakor ezt a projektet hivatkozza be, tehát nekünk ezt kézzel nem kell elindítanunk, megte-szi ő helyettünk. A projektről készült osztálydiagramokat több részre bontottam a könnyebb átláthatóság miatt. Illetve a hatalmas terjedelmük miatt egyes osztályok kevesebb adattal vannak megjelenítve a diagramokon, viszont ezeket az osztályokat a *Visual Studio*-n belül teljes egészében megtekinthetjük. A következő ábrákon ezek láthatóak.

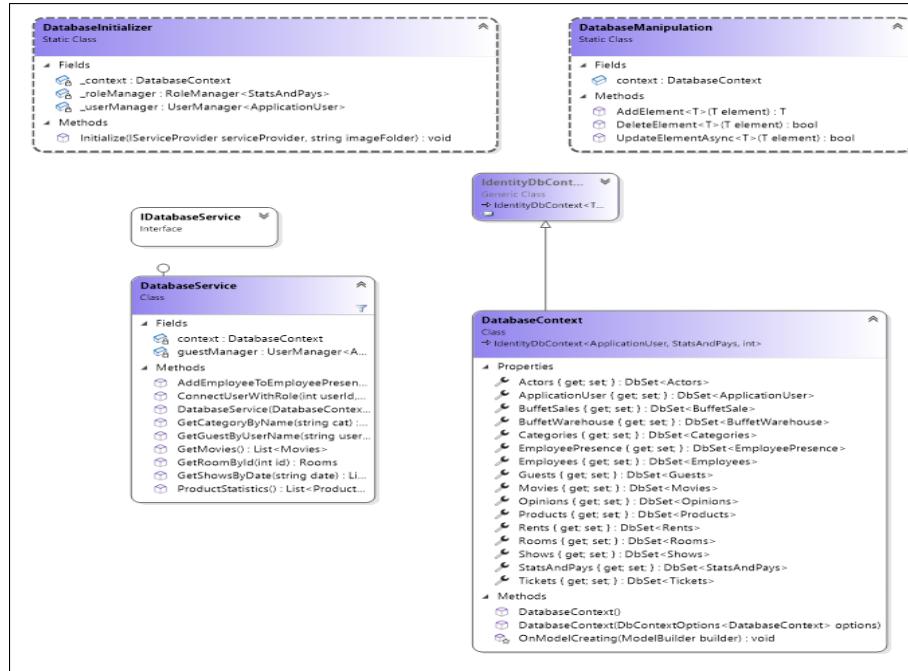
3. Fejlesztői dokumentáció



3.4. ábra. A DTO osztályok egyik fele



3.5. ábra. A DTO osztályok másik fele



3.6. ábra. A modell többi osztálya

Az alábbi ábrákon az adatbázist leíró entitás osztályokról nem készítettem osztálydiagrammokat, ezekről a következő részben esik majd szó, egy adatbázist bemutató ábrával együtt.

Mint láthatjuk a 3.4. ábrán illetve a 3.5. ábrán a Modell projektünkben rengeteg DTO osztály található. A DTO-k, más néven Data Transfer Object-ek, lényege, hogy egy adott adatbázis entitást nem teljes valójában tesznek elérhetővé, hanem csak a szükséges információkat bocsátják ki a külvilág számára. Ezenfelül JSON szerializálhatóságot is biztosítanak, azaz ha egy objektum hivatkozik önmagán belül egy olyan objektumra, ami reprezentációjában hivatkozik az elsőre, akkor ez egy végtelen hivatkozási láncot eredményezne, amit nem lehetne leírni egy JSON fájlba. Ezt orvosolják számunkra a DTO osztályok.

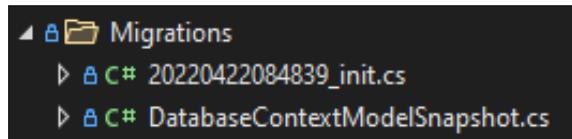
A legtöbb DTO osztály egy adatbázis entitásból származik, azokat reprezentál-ják. Ahhoz, hogy egy adatbázis entitást DTO-vá lehessen alakítani *explicit operator*-okat definiálunk a transzfer osztályokba. Ezek feladata egy entitás átalakítása DTO-vá, illetve visszafelé is. A DTO-k JSON szerializálhatóságát a kontrollerek esetében tudjuk kihasználni, mivel a kontrollerek JSON objektumok formájában válaszolnak a kérésekre, így azok előállításánál lényeges, hogy létrehozhatóak legyenek, máskülönben hibára futna a válasz.

Mint a fenti ábrákon láthatjuk, a modellünk nem csak adatbázis entitásokból és DTO-kból áll, tartalmaz mellette más fájlokat is. Itt található a már említett

DatabaseService és az Ő interfészét tartalmazó *IDatabaseService* is. Ezekről már esett szó, így rájuk nem térnék ki külön. Található ezeken kívül egy *DatabaseManipulation* osztály is. Ez az alapvető adatbázis módosító műveleteket tartalmazza: hozzáadás, frissítés és törlés. Ezeket generikus metódusok segítségével oldottam meg, ezáltal egy metódus ki tudja szolgálni az összes adatbázis entitást. Található még egy *DbContext* osztály is, amely, mint már említettem, az adatbázis kontextus osztálya, azaz ebben az osztályban inicializálónak az adatbázis táblái, illetve tudjuk ezeket a táblákat testre szabni különböző tulajdonságokkal.

Tartalmaz még egy *DatabaseInitializer* nevű osztályt is. Ez egy alap adatbázis inicializációt hajt végre, tölti fel olyan adatokkal amelyek az időszámhoz elengedhetetlenek.

A Modell-en belül található egy *Migrations* mappa is. Erről a mappáról nem készítettem osztálydiagramot, mivel ennek a mappának a tartalma automatikusan generált, és továbbfejlesztésnél is folyamatosan változna. Ez a mappa, mint a neve is utal rá, migrációkat tartalmaz. A migrációk feladata az adatbázist érő módosítások lekövetése. Tegyük fel, hogy van egy használatban lévő alkalmazásunk amihez készítünk egy új funkciót, amely adatbázis módosítással jár. Ekkor nem lenne célszerű, ha újra kellene építenünk az adatbázist, hiszen elvesztenénk minden benne lévő adatot. Helyette szeretnénk, ha az új adatbázis módosítása úgy menne végbe, hogy az ne járjon adatvesztéssel. Ilyenkor létrehozunk egy migrációs állományt, amely tartalmazza az új adatbázis módosítást, majd ezt lefuttatva adatvesztés nélkül létre tudtuk hozni a kívánt változtatást az adatbázison. A *Migrations* mappa tartalmát a 3.7. ábra mutatja be.



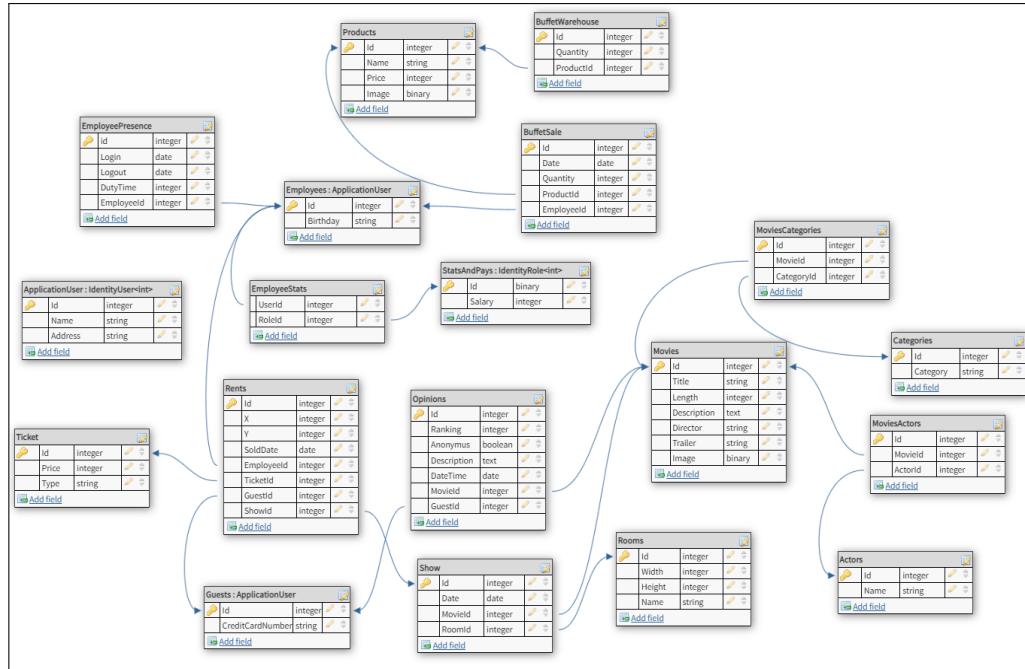
3.7. ábra. Migrációs állományok

Mint látható a mappában jelenleg kettő darab fájl van. Az első az inicializáló migráció, ez állítja be az alap adatbázis sémát. E mellett található egy *DbContextModelSnapshot* állomány is. Ez az állomány tartalmazza az adatbázis aktuális állapotát. Amikor új migrációt hozunk létre és futtatjuk le, akkor a rendszer ezzel a fájllal hasonlítja össze az adatbázis sémánk állapotát.

A *Visual Studio*-n belül lehetőségünk van migrációk létrehozására, illetve azok végrehajtására is. A fejlesztő felületen belül a *Package Manager Console*-t nyissuk meg. Ezt a felső menüsáv *View* menüpontjának azon belül is az *Other Windows* almenüpontja alatt találjuk. Ahhoz, hogy migrációs utasításokat tudunk kiadni szükséges egy *NuGet Package* telepítése is. Ez nem más mint a *Microsoft.EntityFrameworkCore.Tools*. Fontos, hogy ezt is a megfelelő .NET verzióval telepítsük fel. Ezt követően már használhatjuk is a migrációs utasításokat. Az első ilyen utasítás az *Add-Migration*. Ez készít egy új migrációs fájlt a jelenlegi entitás modellek alapján. Ez még nem hajtódiik végre az adatbázison, csak a migrációs fájl jön létre. Ahhoz, hogy a módosítások végbe is menjenek ki kell adnunk az *Update-Database* utasítást. Ha vissza szeretnénk vonni egy módosítást (vagy esetleg egy régebbi verzióra visszaállítanánk az adatbázist) akkor azt a *Remove-Migration* parancsal tehetjük meg.

3.3.3. Adatbázis leírása

Mint már a használt technológiáknál említettem az adatbázist leíró entitás osztályok elkészítéséhez az Entity Framework Core keretrendszeret vettem segítségül és ennek felhasználásával Code-first módon készítettem el az adatbázisomat. Az adatbázis 17 darab általam létrehozott táblából, illetve kettő automatikusan generált táblából áll. Ezeket a 3.8. ábrán láthatjuk.



3.8. ábra. Migrációs állományok

A következő részben az egyes táblákat illetve azok oszlopait mutatnám be a könnyebb érthetőség kedvéért.

Products

A termékeket tároló tábla.

- **Id (Elsődleges kulcs)**: A termék azonosítója
- **Name**: A termék neve
- **Price**: A termék ára
- **Image**: A termékhez tartozó kép bináris formátumban.

BuffetWarehouse

A mozi raktárát reprezentáló tábla.

- **Id (Elsődleges kulcs)**: A raktár bejegyzésének azonosítója.
- **Quantity**: A termék darabszáma a raktárban.
- **ProductId (Idegen kulcs a Product táblára)**: A termék azonosítója.

ApplicationUser

Az alkalmazás minden felhasználóját tároló tábla. Ez nem csak az alkalmazottakat, hanem a vendégek adatait is tárolja. Mint a 3.8. ábrán látható, ez egy *IdentityUser* nevű osztályból származik. Ez az osztály az *ASP.NET Core* által használt felhasználó tábla, benne alapértelmezetten definiált oszlopokkal (pl felhasználó név, email, telefonszám stb). Az implementációmánál ezt a táblát használom fel, egy pár extra oszloppal kibővítve.

- **Id (Elsődleges kulcs):** A felhasználó azonosítója
- **Name:** A felhasználó teljes neve.
- **Address:** A felhasználó címe.

Employees

Az alkalmazottakat tároló tábla. Ez az ApplicationUser-ből származik le, tehát az ott definiált oszlopok itt is jelen vannak, viszont ezeket kibővítettem külön az alkalmazottakra nézve.

- **Id (Elsődleges kulcs):** Az alkalmazott azonosítója
- **Birthday:** A alkalmazott születésnapja.

Guests

A vendégeket tároló tábla. Ez az ApplicationUser-ből származik le, tehát az ott definiált oszlopok itt is jelen vannak, viszont ezeket kibővítettem külön a vendégekre nézve.

- **Id (Elsődleges kulcs):** A vendég azonosítója
- **CreditCardNumber:** A vendég bankkártyaszáma.

BuffetSale

A bőfű eladásait rögzítő tábla.

- **Id (Elsődleges kulcs):** Az eladás azonosítója.

- **Date:** Az eladás dátuma.
- **Quantity:** Az eladott termékek darabszáma.
- **ProductId (Idegen kulcs a Product táblára):** Az eladott termék azonosítója.
- **EmployeeId (Idegen kulcs az Employess táblára):** Az eladó alkalmazott azonosítója.

EmployeePresence

Az alkalmazottak jelenlétét tartalmazó tábla.

- **Id (Elsődleges kulcs):** Az eladás azonosítója.
- **Login:** A bejelentkezés dátuma.
- **Logout:** A kijelentkezés dátuma.
- **DutyTime:** A bejelentkezés és a kijelentkezés között eltöltött idő.
- **EmployeeId (Idegen kulcs az Employees táblára):** Az eladó alkalmazott azonosítója.

StatsAndPays

A szerepköröket tartalmazó tábla. Mint láthatjuk ez az *IdentityRole* osztályból származik. Ezt is az *ASP.NET Core*-nak köszönhetően érhetjük el, és csak úgy mint az *IdentityUser* esetében itt is alapvető oszlopokkal rendelkezünk (pl egy szerepkör neve).

- **Id (Elsődleges kulcs):** Az eladás azonosítója.
- **Salary:** A szerepkörhöz tartozó fizetés.

EmployeeStats

Kapsoló tábla, mely egy alkalmazottat kapcsol össze az ő szerepkörével. Egy alkalmazotthoz több szerepkör is tartozhat illetve fordítva is igaz ez. Ez a tábla is rendelkezik egy ősosztállyal, mégpedig az *IdentityUserRole*-ral. Alapértelmezetten

nem kellett volna ezt a kapcsoló táblát létrehozni, viszont mivel itt saját *IdentityUser* táblánk és saját *IdentityRole* táblánk van, így erre szükség van a megfelelő kapcsolat kialakításáért.

- **UserId (Idegen kulcs az Employees táblára)**: Egy felhasználó azonosítója.
- **RoleId (Idegen kulcs a StatsAndPays táblára)**: Egy szerepkör azonosítója.

Ticket

A jegytípusokat tartalmazó tábla.

- **Id (Elsődleges kulcs)**: A jegytípus azonosítója.
- **Price**: Az adott jegytípus ára.
- **Type**: A jegytípus neve (pl Diák vagy Nyugdíjas).
- **DutyTime**: A bejelentkezés és a kijelentkezés között eltöltött idő.
- **EmployeeId (Idegen kulcs az Employees táblára)**: Az eladó alkalmazott azonosítója.

Categories

A kategóriák táblája.

- **Id (Elsődleges kulcs)**: A kategória azonosítója.
- **Category**: A kategória neve.

Actors

A színeszeket reprezentáló tábla.

- **Id (Elsődleges kulcs)**: A kategória azonosítója.
- **Name**: A színész neve.

Movies

A filmeket reprezentáló tábla.

- **Id (Elsődleges kulcs)**: A film azonosítója.
- **Title**: A film címe.
- **Length**: A film hossza.
- **Description**: A film leírása.
- **Director**: A film rendezője.
- **Trailer**: A film előzetesének hivatkozása.
- **Image**: A film borítóképe bináris formátumban.

MoviesActors (Automatikusan létrehozott tábla)

Kapcsoló tábla, amely a színészek és filmek közötti kapcsolatot definiálja. Egy színésznek több filmje is lehet, illetve ez fordítva is igaz.

- **Id (Elsődleges kulcs)**: A kapcsolat azonosítója.
- **MovieId (Idegen kulcs az Movies táblára)**: A film azonosítója.
- **ActorId (Idegen kulcs az Actors táblára)**: A színész azonosítója.

MoviesCategories (Automatikusan létrehozott tábla)

Kapcsoló tábla, amely a kategóriák és filmek közötti kapcsolatot definiálja. Egy kategória több filmhez is besorolható, illetve egy filmnek több kategóriája is lehet.

- **Id (Elsődleges kulcs)**: A kapcsolat azonosítója.
- **MovieId (Idegen kulcs az Movies táblára)**: A film azonosítója.
- **CategoryId (Idegen kulcs az Categories táblára)**: A kategória azonosítója.

Rooms

Kapcsoló tábla, amely a kategóriák és filmek közötti kapcsolatot definiálja. Egy kategória több filmhez is besorolható, illetve egy filmnek több kategóriája is lehet.

- **Id (Elsődleges kulcs):** A kapcsolat azonosítója.
- **Width:** A terem szélessége.
- **Height:** A terem hosszúsága.
- **Name:** A terem neve.

Show

Az előadásokat tartalmazó tábla.

- **Id (Elsődleges kulcs):** A kapcsolat azonosítója.
- **Date:** Az előadás időpontja.
- **MovieId (Idegen kulcs a Movies táblára):** Az előadáshoz tartozó film azonosítója.
- **RoomId (Idegen kulcs a Rooms táblára):** Az előadáshoz tartozó terem azonosítója.

Rents

A foglalásokat, illetve jegyeladásokat reprezentáló tábla.

- **Id (Elsődleges kulcs):** A foglalás azonosítója.
- **X:** A választott hely X koordinátája (sor azonosító).
- **Y:** A választott hely Y koordinátája (oszlop azonosító).
- **SoldDate:** A foglalás eladásának dátuma.
- **EmployeeId (Idegen kulcs a Employees táblára):** A foglalást értékesítő alkalmazott azonosítója
- **TicketId (Idegen kulcs a Ticket táblára):** A foglalt hely típusa

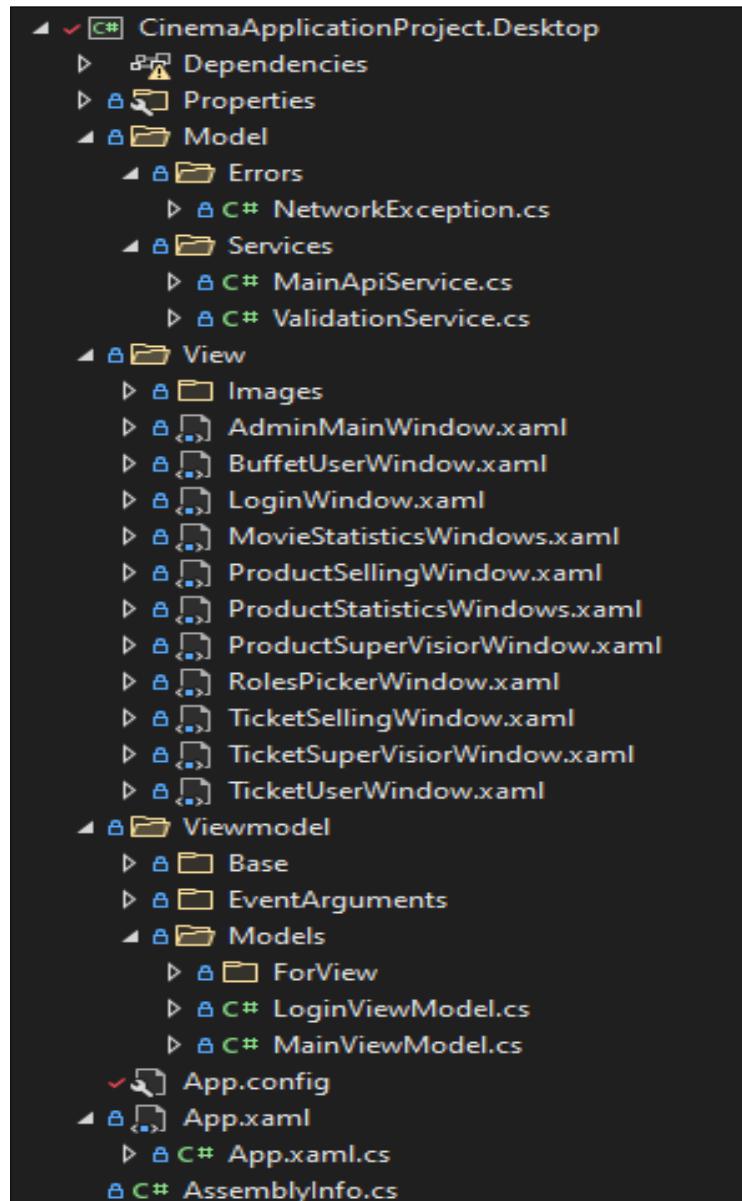
- **GuestId (Idegen kulcs a Guests táblára):** Az előadáshoz tartozó terem azonosítója.
- **ShowId (Idegen kulcs a Show táblára):** Az foglalt jegyhez tartozó előadás azonosítója.

Opinions

A filmekhez tartozó véleményeket tartalmazó tábla.

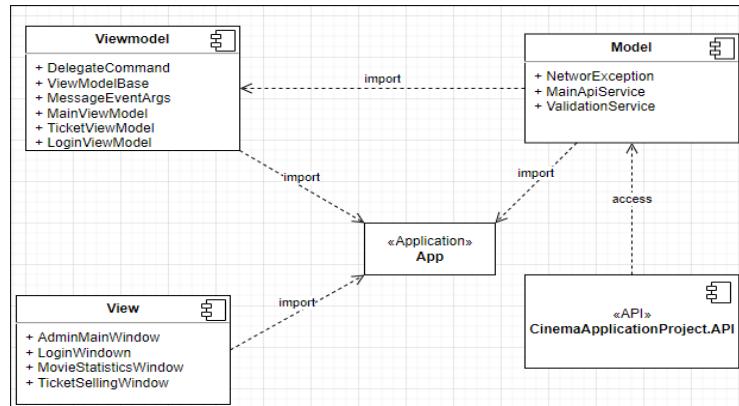
- **Id (Elsődleges kulcs):** A foglalás azonosítója.
- **Ranking:** A vélemény értékelésére számként.
- **Anonymus:** A felhasználó anonimitását jelző érték.
- **DateTime:** A vélemény elkészülte.
- **MovieId (Idegen kulcs a Movies táblára):** A film azonosítója, amelyet a felhasználó értékelt.
- **GuestId (Idegen kulcs a Guests táblára):** Az előadáshoz tartozó terem azonosítója.

3.4. Asztali alkalmazás bemutatása



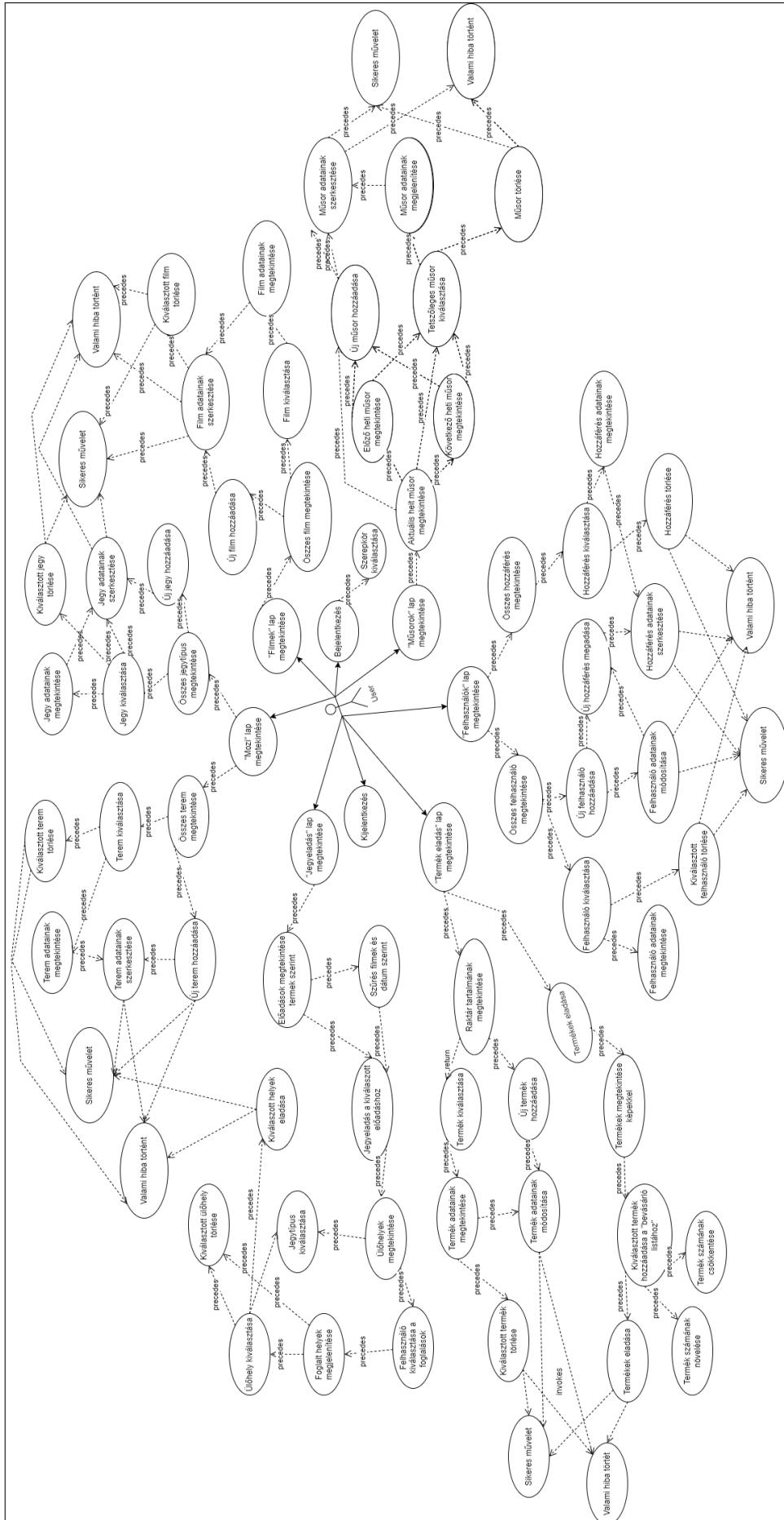
3.9. ábra. Az asztali alkalmazás architektúrája

Az asztali alkalmazás, mint már a felhasznált architektúrák részben említettem, egy MVVM architektúrában megírt alkalmazás, tehát a működés három külön rétegre oszlik. Ezt a három réteget egy közös vezérlő réteg felügyeli, és indítja el az alkalmazáson belül. Ezt a vezérlő réteget az App.xaml reprezentálja. Az alkalmazásról csomagdiagram készült, mely az egyes rétegek közötti kommunikációt mutatja be, ez a 3.10. ábrán látható. Helyhiány miatt a diagramon felsorolt objektumok száma kevesebb mint amennyi ténylegesen odatartozik. Ezekről teljes képet majd az egyes részek leírásánál láthatunk.



3.10. ábra. Az alkalmazás osztálydiagrammja

Mielőtt végignéznénk az alkalmazás egyes rétegeit, lássuk milyen funkciók elérhetőek az alkalmazásunkban. Ezeket a funkciókat UML (Unified Modeling Language) [23] segítségével mutatom be, amely a 3.11. ábrán látható. Ez az UML diagram minden funkciót ábrázol, azaz az adminisztrátor szerepkörrel rendelkező felhasználók érhetik el. Az egyes szerepkörökkel is ezeket a funkciókat lehet elérni, csak korlátozott számban. Ezen korlátozásokról az UML diagram alatti szekcióban olvashatunk bővebben.



3.11. ábra. Az alkalmazás UML diagramja

Pénztár területi vezető (Ticket Supervisor)

- Bejelentkezés
- Jegyeladás
- Felhasználók megtekintése (szerkesztési lehetőségek nélkül)
- Kijelentkezés

Pénztáros (Ticket Seller)

- Bejelentkezés
- Jegyeladás
- Kijelentkezés

Büfé területi vezető (Ticket Supervisor)

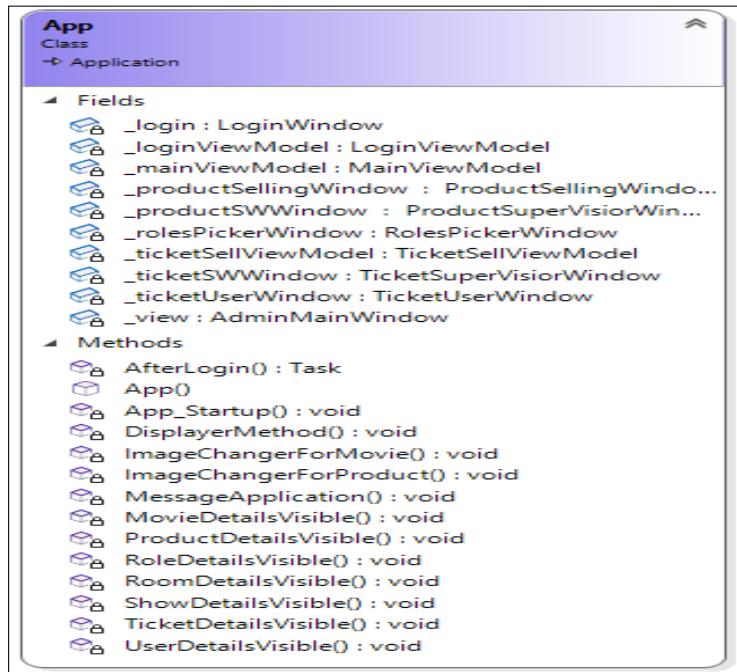
- Bejelentkezés
- Termékeladás
- Felhasználók megtekintése (szerkesztési lehetőségek nélkül)
- Termékek megtekintése és szerkesztése
- Kijelentkezés

Büfés (Product Seller)

- Bejelentkezés
- Jegyeladás
- Kijelentkezés

3.4.1. Vezérlés

Mint említettem a vezérlést az App.xaml végzi. Ez az osztály inicializálja az egyes rétegekhez tartozó osztályokat illetve indítja el a kezdeti képernyőt. Az App osztály osztálydiagramja a 3.12. ábrán látható.

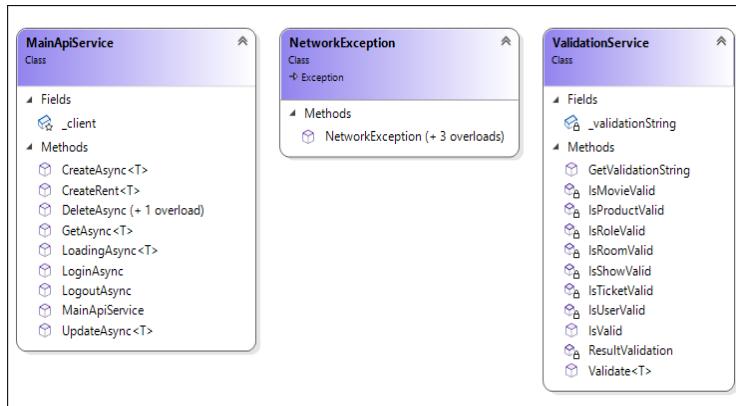


3.12. ábra. A vezérlés osztálydiagramja

Az inicializálásért az *App_Startup* metódus felel, ez az amelyik indításkor végrehajtódik. Itt nem csak az egyes rétegek osztályait inicializálom, hanem a nézetmodell által használt eseményekre is feliratkozok. Ezek az események felelnek az egyes részletek megjelenítéséért a felhasználói felületen. Ezekre az eseményekre egy-egy külön metódussal iratkoltam fel, ezért szerepel ennyi külön metódus az osztályon belül.

3.4.2. Modell

Mint a fentiekben már leírtam, a modell feladata az API-val való kapcsolattartás, az ehhez tartozó osztályok a 3.13. ábrán láthatóak.



3.13. ábra. Az modell osztálydiagramja

A kapcsolattartást a *Main ApiService* végzi. Ez az az osztály mely példányosítva lett a nézetmodellen belül. A benne definiált metódusok minden generikus metódusok, azaz minden adatbázis objektumra értelmezettek és használhatóak. Ezáltal a kód könnyebben átlátható és szerkeszthetővé vált. Ezt a vezérlő osztály konstruálja. A *Main ApiService*, mint az ábrán is látszik, visszatérése egy *Task* objektum. Ez azt jelenti, hogy a metódusok *async* módon futnak, azaz a várakoznak az *API* által adott válaszra, addig pedig a program futását is blokkolják.

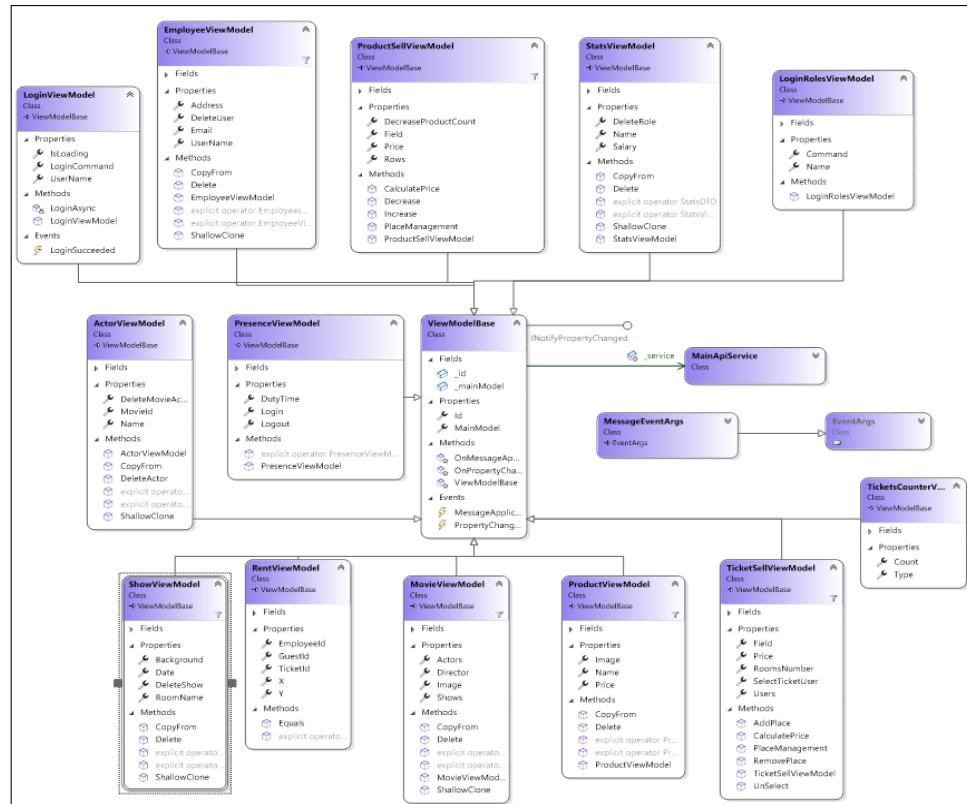
A Modellen belül található egy *NetworkException* osztály is, amely egy hibát hivatott jelezni. Ha a *Main ApiService* bármely metódusa hibára fut, akkor ezzel jelzi azt az alkalmazásnak.

Található még itt egy harmadik osztály is. Ez a *Validation Service*. Ennek a feladata a beadott adatok validációja, ellenőrzése, még mielőtt azokat elküldené a *Main ApiService* megfelelő metódusa az *API*-nak.

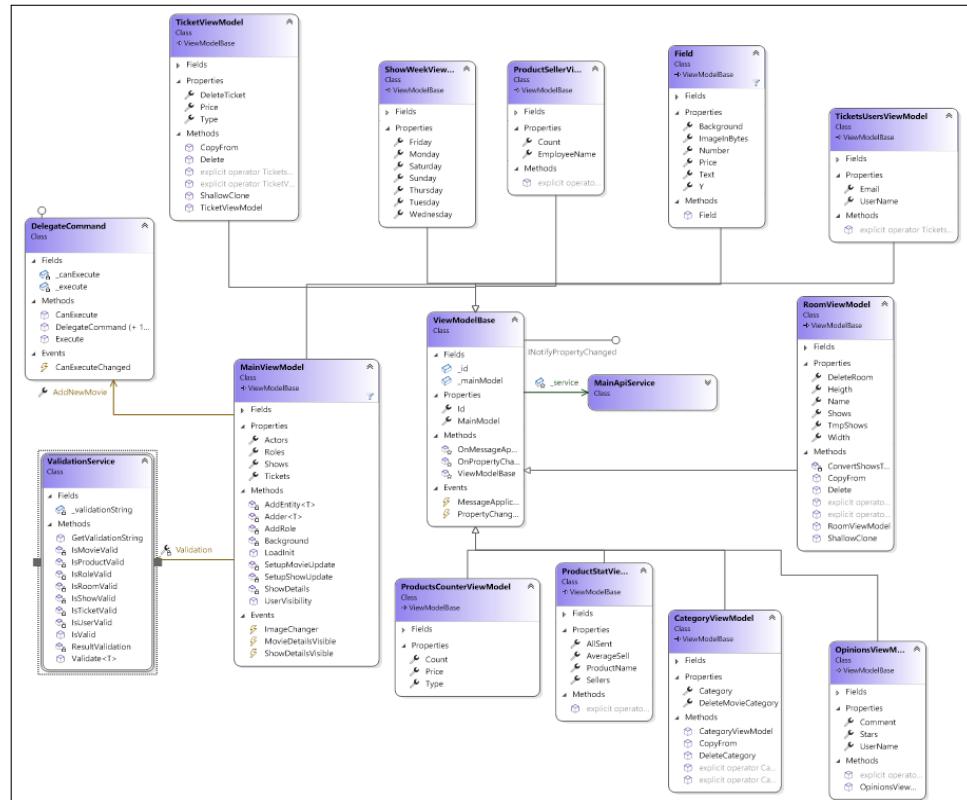
3.4.3. Nézetmodell

A nézetmodell osztálydiagramja a 3.14. ábrán és a 3.15 látható. A könnyebb áttekinthetőség miatt két részre bontottam ezt. Helyhiány miatt, egyes metódusok a képről lemaradtak, így ez csak reprezentációs céllal van jelen.

3. Fejlesztői dokumentáció



3.14. ábra. A nézetmodell egyik osztálydiagramja



3.15. ábra. Az nézetmodell másik osztálydiagramja

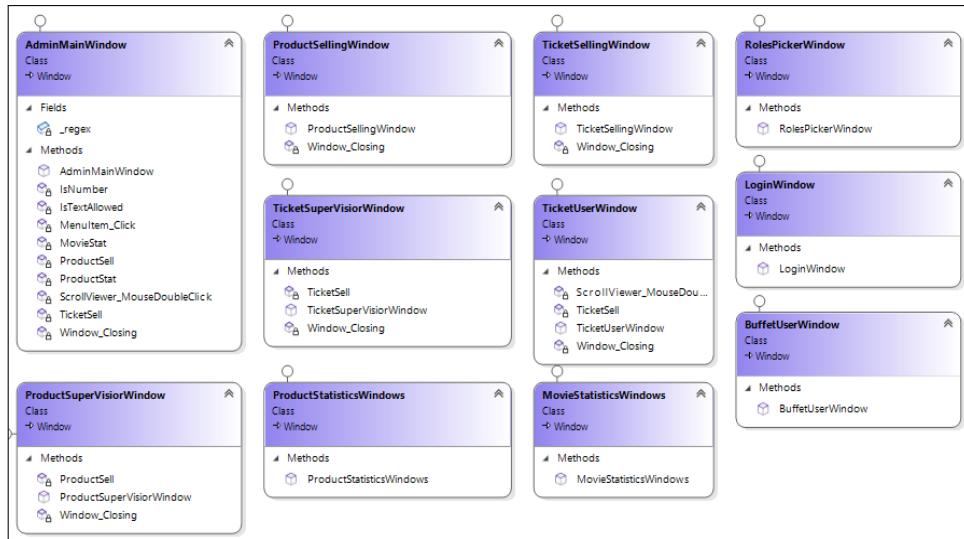
A nézetmodell a nézetet kiszolgáló réteg, ezáltal itt történik a megjelenítendő adatok előállítása. Ő az a réteg, aki metódusokon keresztül kommunikál a modellel, ezáltal egy közvetett kapcsolat által elérve az *API*-t. A működésének nagy részét a *MainViewModel* látja el. Ez egy leszármazott osztály, az ő szülöje, a *ViewModelBase*, definiál minden olyan metódust és adattagot, amely ahhoz kell, hogy a nézet által megjeleníthető objektum legyen készíthető. A *Main ApiService* is itt inicializálódik egy *ConfigurationManager* osztály segítségével. Ez az osztály az alkalmazás config beállításaiból szerzi meg a szükséges információt. Jelen esetben ez a szükséges információ az adatbázis eléréséhez szükséges URL.

Ebben a rétegben található egy *MessageEventArgs* osztály is, amely segítségével a nézetmodell visszajelzéseket küld egy-egy hibáról a felhasználó számára. Ezeket a visszajelzéseket eseményeken keresztül teszi, melyeket a vezérlő réteg kezel le.

A nézetmodellen belül a *MainViewModel*-en kívül rengeteg másik *ViewModel* osztály található még ebben a rétegen. Ezek feladata az *API*-tól visszakapott DTO-k átalakítása a nézet számára, hogy azokat az adatokat megfelelően meg tudja jeleníteni. Ezek is a *ViewModelBase*-ból származnak, tehát ők is rendelkeznek minden szükséges művelettel. Itt is, mint az adatbázis entitások és DTO-k esetében, a DTO-kból való átalakítást *explicit operator*-ök segítségével oldottam meg. Ezekkel definiáltam, hogy az egyes DTO-k adattagjai mely nézetmodell adattagokba kerüljenek, illetve, hogy ha kellett, ekkor módosítottam őket a megjelenítés szerint.

3.4.4. Nézet

Ez a réteg tartalmazza a felület teljes leírását. minden nézet egy-egy .xaml kiterjesztésű fájlba lett definiálva, illetve ezek a fájlok egy ablak teljes leírást tartalmazzák. Az ablakokon belül az egyes funkciók úgynevezett tabokon vannak külön-külön elszeparálva. Előfordul, hogy egy tabon több funkció is szerepel (pl a "Cinema" tab az adminisztrátori felületen) ennek a könnyebb kezelhetőség az oka. A nézet réteg osztálydiagramja a 3.16. ábrán található.



3.16. ábra. Az nézet réteg osztálydiagramja

Az nézet rétegen szereplő ablakok nagy része egy-egy külön szerepkörhöz tartozik, ezekre az adott fájl nevei is utalnak. Vannak viszont közös használatban lévő ablakok. Ezeket is két csoportra oszthatjuk. Van olyan, amelyeket minden felhasználó elérhet, ez a *LoginWindow.xaml* illetve a *RolesPickerWindow.xaml*. Ezek a bejelentkező ablakot, illetve a bejelentkezés után felugró szerepkör választó ablakot definiálják. A szerepkör választó ablak csak abban az esetben jelenik meg, ha egy felhasználónak több szerepköre is van.

A másik csoport pedig az olyan ablakokat tartalmazza, amelyeket több szerepkör is elérhet, viszont nem mindenkinél elérhető. Ezek például a jegyárusító (*TicketSellingWindow.xaml*) illetve a termék árusító (*ProductSellingWindow.xaml*) felületek.

3.5. Webes alkalmazás bemutatása

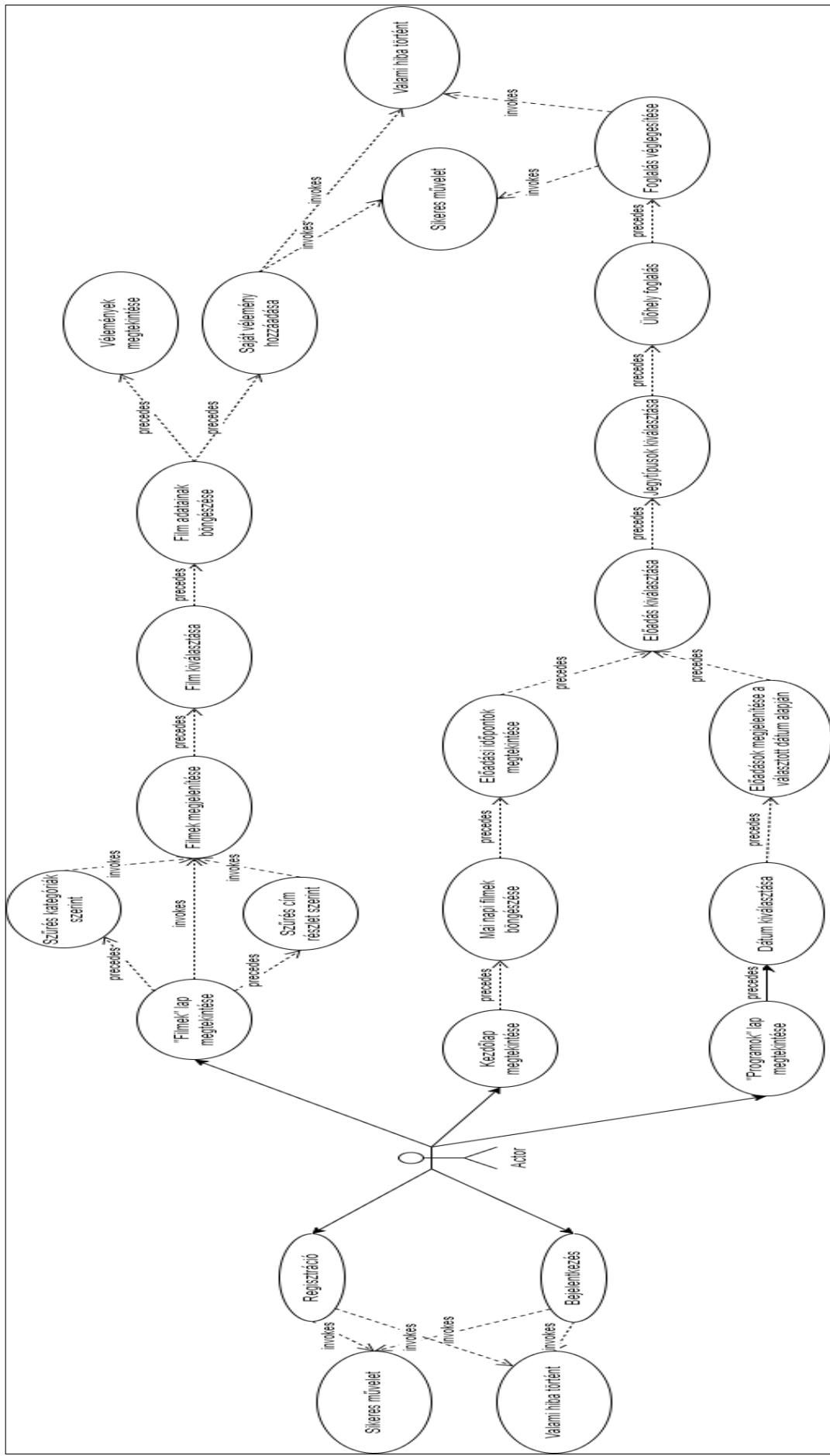
```
> └── dist
> └── node_modules library root
> └── public
└── src
    ├── assets
    └── components
        └── CardComponent.vue
        └── CategoryAndSearchComponent.vue
        └── DatepickerComponent.vue
        └── ErrorcardComponent.vue
        └── InputErrorComponent.vue
        └── LoginComponent.vue
        └── NavbarComponent.vue
        └── OpinionsComponent.vue
        └── ShowCardComponent.vue
        └── SignupComponent.vue
        └── Slider.vue
    ├── plugins
    ├── router
    ├── store
    └── views
        └── AddOpinion.vue
        └── Home.vue
        └── MovieDetails.vue
        └── Movies.vue
        └── Program.vue
        └── Reserve.vue
    └── App.vue
    └── headers.js
    └── main.js
```

3.17. ábra. A webes alkalmazás architektúrája

A szakdolgozatomhoz tartozó weboldalt *Vue.js* segítségével építettem fel. A *Vue.js* segítségével egy oldalból álló weboldalt tudunk úgy készíteni, hogy azt a hatást kelti, mintha az több oldalból állna. Ezt komponensek segítségével tehetjük. minden komponens külön funkcióval rendelkezik, illetve őket akár többször is felhasználhatjuk a weboldalunkon belül.

3.5.1. Az alkalmazás UML diagramja

A weboldalhoz is készült egy UML diagram, csak úgy mint az asztali alkalmazás esetében. Ezáltal kerül bemutatásra a webes alkalmazásban található funkciók. Ez a 3.18. ábrán láthatjuk.



3.18. ábra. A weboldal UML diagramja

3.5.2. Az alkalmazás struktúrája

Ahhoz, hogy létre tudjunk hozni egy *Vue.js* által elkészített weboldalt, mindenéppen szükségünk van, egy HTML fájlra, benne egy blokkal (div), amelynek azonosítója(id) az "app" nevet kapja. Ez határozza meg, hogy az általunk .vue fájlokba készített komponensek, hova is kerüljenek az oldalunkon. Ezt követően ezzel a html fájllal már nem is kell foglalkoznunk, innentől a nézet szerkesztését már .vue kiterjesztésű fájlokban fogjuk végezni.

Mielőtt rátérnék a tényleges megvalósításra, szükségünk van egy *main.js* állományra is. Ez az az állomány, amely a szükséges csomagokat importálja, illetve példányosít számunkra az "app" azonosítóval egy új Vue [24] objektumot. Enélkül nem tudunk *Vue.js* alapú alkalmazást létrehozni. Csomagok természetesen komponenseken belül is importálásra kerülhetnek, viszont ezzel, hogy ebben a *main.js* állományban elvégezzük ezt, az adott csomagok globálisan, az egész alkalmazáson belül elérhetőek lesznek.

Ahhoz, hogy új Vue objektumot tudjunk létrehozni, szükséges egy induló .vue kiterjesztésű fájl is, ami az alkalmazásunk alapfelülete lesz. Ezt általában az *App.vue* fájlból találjuk. Ez az a fájl aminek a tartalma az első bekezdésben említett "app" azonosítóval rendelkező blokkba kerül bele. Ezzel már létre tudunk hozni egy egyedi dizájnnal rendelkező weboldalt.

Általánosságban elmondható, hogy minden .vue kiterjesztésű fájl három részből tevődik össze. Az első rész a *template* szekció, ahol megtervezhetjük az alkalmazásunk felületét. Ezt lehetjük akár HTML attribútumokkal, akár általunk már definiált *Vue* komponensekkel, vagy akár külső csomagból származó komponensek felhasználásával is. A szakdolgozatom esetében két külső komponens csomag került importálásra, az egyik a *Bootsrap Vue* a másik pedig a *Vuetify* keretrendszer. Mind két keretrendszer ingyenesen felhasználható, jól dokumentált illetve ingyenesen bárki számára elérhető rendszer. Ezek által biztosított komponenseket be tudtam építeni a saját komponenseimbe, így egy még letisztultabb felhasználói felületet létrehozva.

A második szekció a *script*. Itt tudjuk az alkalmazásunk felületéhez szolgáló logikát elvégezni. Ez egy nézetmodellbeli rétegnek felel meg, itt történik az megjelenítendő adatok előállítása, illetve a kommunikáció a modell-el, ami jelen esetben az *API*. Ez a rész tartalmazza a szükséges importokat (pl a felhasználandó komponenseket itt kell behivatkoznunk), illetve a nézetmodell leírását *Vue* specifikus módon.

A Vue specifikus nézetmodell főbb részei amelyek az alkalmazásban is felhasználásra kerültek a következőek:

- Name: Az adott komponens nevét adja meg
- Components: A felhasznált komponensek listája
- data(): A komponenshez tartozó változók inicializálása
- methods: A komponens által használt metódusok definíciói
- created: A komponens létrehozásakor lefutó metódusok halmaza.

A harmadik szekció pedig a *style* rész. Ezt az alapvető webfejlesztési struktúrából már ismerhetjük, itt hozhatunk létre olyan CSS [25] osztályokat, melyek a dizajnt határozzák meg. Ez a rész elhagyható, ha mi saját css osztállyal rendelkezünk, ahol minden értéket már definiáltunk előre.

Ahhoz, hogy a felhasználóknak egy oldallal rendelkező weboldalnak tűnjön a felületünk, a *Vue.js*-nek szüksége van egy csomagra. Ez a csomag nem más mint a *vue-router* [26] melyet, egy egyszerű node utasítással telepíthetünk az alkalmazásunkhoz. Ez is, mint a neve utal is rá, a *Vue.js* szerkesztői készítettek. Célja, egy olyan komponens biztosítása, amellyel könnyen kezelhető egy weboldal esetében a több oldalas (értsd: nem csak egy kezdőlappal, hanem több oldallal rendelkezik) megvalósítás. Ezt egy *router-view* komponens segítségével teszi. Ezt a komponensem elegendő egyszer definiálunk az *App.vue* fájlunkban. Ilyenkor a definiált területen (*<router-view>* tag-ek között) fogja megjeleníteni az éppen paraméterként megkapott komponenseket. Erről példát a 3.1. ábrán láthatunk.

```
1 <template>
2   <div id="background">
3     <Slider />
4     <NavbarComponent />
5     <div id="app">
6       <router-view></router-view>
7     </div>
8   </div>
9 </template>
```

3.1. forráskód. App.vue template szekciójá

Ezen a példán látható, hogy az oldalunk felületén meg fog jelenni, egy *Slider* komponens, alatta egy navigációs rész, majd az alatt egyből a paraméterül kapott komponenseinket jeleníti meg a *router-view*.

A *vue-router*-nek szüksége van egy elérési utakat, illetve hozzájuk tartozó komponenseket definiáló fájlra is. Ebben adjuk meg, hogy az egyes felületeink minden címen legyenek elérhetőek. Egy példa erre, hogy ha a szakdolgozatomban a kezdőlapot szeretnénk elérni, akkor azt, tesztkörnyezetben futtatás esetén, a `http://localhost:8080` -al tehetjük meg, míg például a filmeket megjelenítő felületet már a `http://localhost:8080/movies` elérési úttal. Ezek miatt a címek miatt is úgy tűnhet egy felhasználónak, hogy egy több oldalas weboldalon van, holott tudjuk, hogy ezekkel az utakkal csak a *router-view* komponensben megjelenített komponenseket definiáljuk.

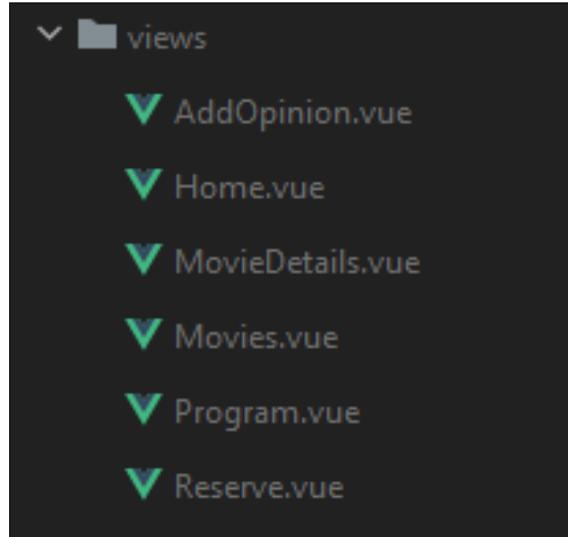
A *router-view* számára az elérési utakból és komponensek nevéből álló párokat a `router/index.js` fájlban találhatjuk, ezen belül is a `routes` konstans listában szerepelnek. Ez az állomány tartalmazza még a *vue-router* beállítását is.

Az egyes komponensek között fontos az adattovábbítás. Alapértelmezetten a Javascript *emit* utasításával tudnánk a szülő komponensnek adatot küldeni, az pedig a *Vue.js* által behozott props tulajdonság használatával tudná egy másik gyerek komponensnek a kívánt adatokat. Ez a fajta kommunikáció sok munkát igényel, illetve nagyon lassú folyamat. Ahhoz, hogy ezt gyorsítani tudjuk a *Vue.js* fejlesztői által készített *Vuex* [27] keretrendszerre. Ez egy állapot kezelő rendszer, ennek segítségével létre tudunk hozni olyan objektumokat, amelyeket több komponensen keresztül elérünk. Ezeket az objektumokat hívjuk állapotoknak (angolul state-nek). A state-eket a projekten belül egy globális állományban, a `store/index.js`-ben definiáltam. Állapotokhoz tudunk készíteni lekérdezőket (gettereket), módosítókat (mutations), műveleteket (actions), és modulokat (modules). Alkalmasomban az action-ök és a mutation-ök csak beállító azaz, setter funkcióval rendelkeznek. Ha egy state értékét módosítani szeretném, akkor az adott .vue fájlban a state-hez tartozó action-t hívom meg, ami pedig a definíciója szerint meghívja az ehhez tartozó mutation-t. Fontos még megjegyezni, hogy ahhoz, hogy egy komponensben elérjük a kívánt state-et és annak műveleteit szükséges azokat importálni a komponens *script* szekciójában.

Ahhoz, hogy az API-val tudjon kommunikálni a weboldal, az *axios* nevű csomagot használtam. Ezzel a könyvtárral HTTP kéréseket tudunk végrehajtani, megadott címekre, illetve aszinkronitást is biztosít, mivel ígéret alapú csomagról van szó.

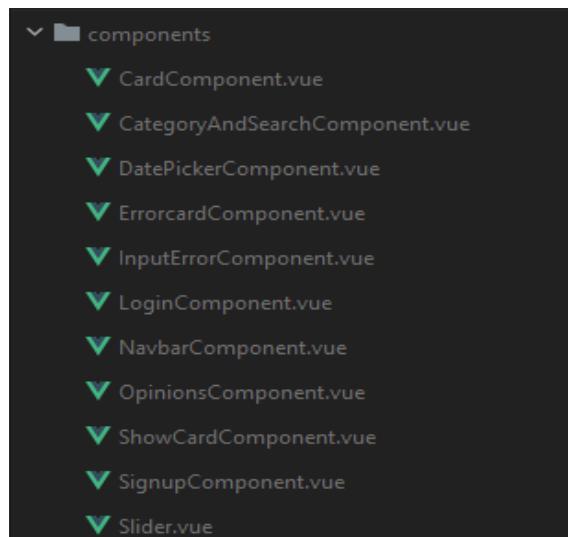
3.5.3. Az alkalmazás nézetei és komponensei

Az alkalmazáson belül hat különböző nézet osztállyal találkozhatunk, ezeket a 3.19. ábrán láthatjuk. Ezek reprezentálják az alkalmazás egyes főoldalait.



3.19. ábra. A "views" mappa tartalma

A főoldalakon való megjelenítést illetve működést pedig az egyes komponensek segítik, melyekből összesen 11 található az alkalmazásban. Ezek a komponensek bár-mikor újrafelhasználhatóak, így egy továbbfejlesztés esetén is nyugodtan fel tudjuk őket bárhol használni, nem kell újat definiálnunk. Illetve, ezáltal a komponensek javítása is egyszerűbb, hiszen elegendő egyetlen helyen javítanunk őket, és ez a változás az összes előfordulásánál végbe megy majd. Ezeket a komponenseket a 3.20. ábra mutatja be.



3.20. ábra. A "components" mappa tartalma

Ezen kívül az `assets` mappában található még négy darab kép fájl, ezek az alkalmazásban használt képeket tartalmazzák. Ilyen képek például a háttér, illetve az alkalmazás felső részén megjelenő logó.

3.6. Tesztelés

Mind a webes alkalmazás, mind az asztali alkalmazás esetében a funkciók ellenőrzése miatt teszteket készítettem. Két fajta tesztelési módszert alkalmaztam ehhez.

Először is egységes teszteket készítettem, amelyek az *API* egyes végpontjai megfelelő működését vizsgálják. Ezeket a *solution*-ben a *CinemaApplicationProject.API* projektben definiáltam. minden kontrollerhez tartozóan egy külön teszt fájlt rögzítettem. Ezekben a teszt fájlokban lettek definiálva a tesztek. Viszont ahhoz, hogy megfelelően tudjam az egyes végpontokat tesztelni, ahhoz egy külön teszt adatbázisra volt szükségem. Ezt a teszt adatbázis, viszont az éles adatbázissal ellentétben nem egy szerveren található, hanem memóriában jön létre, ezáltal biztosítva a gyorsabb elérést, amely a tesztek esetében elengedhetetlen. Mivel ennek a projektnek önállóan kell működnie az *API*-tól függetlenül, így itt az adatbázis felépítését a teszteseteknek kell végezniük. Így minden teszteset konstruktorában ezt a folyamatot definiáltam. Ahhoz, hogy adatokkal is rendelkezzen az adatbázis, létrehoztam egy *TestDbInitializer* osztályt, amely a memóriában lévő adatbázist tölti fel adatokkal. Ezeknél a teszteknek fontos, hogy ne párhuzamosan fussanak, mert akkor a memóriában található adatbázist akár egyszerre két teszt is szerkesztheti, ami következtében előfordulhat, hogy nem az elvárt eredményt kapjuk. Ennek megelőzése képen a teszt osztályokat a *Collection("Sequential")* attribútummal láttam el, amely lehetővé teszi, hogy a tesztek szekvenciálisan egymás után fussanak le. Fontos, hogy az egyes teszt futások után az adatbázis visszaálljon alap helyzetbe, ezért ezt a teszt végeztével töröljük (majd a következő teszt konstruktora újra létrehozza). A törlést a rendszer automatikusan végzi, mivel a tesztek implementálják az *IDisposable* interfészről kapott *Dispose()* metódust.

Egyesegtesztekből összesen 79 darab készült. Ezek futási eredménye a 3.21. ábrán látható. Mivel rengeteg teszt van, így ezen az ábrán a teljesség igény nélkül, csak néhányat jelenítettem meg. A többi teszt a már említett *solution*-ben megtalálható.

◀ ✓ CinemaApplicationProject.APITest (79)	3.4 sec
◀ ✓ CinemaApplicationProject.APITest (79)	3.4 sec
◀ ✓ ActorsControllerTest (7)	1.6 sec
✓ ConnectActorToMovies	30 ms
✓ DeleteActor	166 ms
▷ ✓ GetActorByIdTest (3)	45 ms
✓ GetActorsTest	1.3 sec
✓ GetInvalidActorTest	88 ms
▷ ✓ BuffetWarehouseControllerTest (9)	207 ms
▷ ✓ CategoryControllerTest (7)	155 ms
◀ ✓ EmployeesControllerTest (4)	167 ms
✓ DeleteEmployee	69 ms
✓ GetEmployeesTest	47 ms
✓ PostEmployee	22 ms
✓ PutEmployee	29 ms
◀ ✓ MoviesControllerTest (12)	339 ms
✓ DeleteMovie	21 ms
✓ GetInvalidMovieTest	13 ms
▷ ✓ GetMovieByIdTest (3)	40 ms
✓ GetMoviesByCategoryTest	53 ms
✓ GetMoviesByTitlePartTest	26 ms
✓ GetMoviesStatTest	75 ms
✓ GetMoviesTest	39 ms
✓ GetMoviesTodayTest	36 ms
✓ PostMoviesItemTest	16 ms
✓ PutMoviesItemTest	20 ms
▷ ✓ OpinionsControllerTest (2)	66 ms
▷ ✓ RentsControllerTest (6)	171 ms
◀ ✓ RolesControllerTest (7)	176 ms
✓ DeleteRoleTest	14 ms
▷ ✓ GetRoleByIdTest (3)	47 ms
✓ GetRolesTest	21 ms
✓ PostRoleTest	72 ms
✓ PutMoviesItemTest	22 ms
▷ ✓ RoomsControllerTest (8)	143 ms
▷ ✓ ShowsControllerTest (10)	235 ms
▷ ✓ TicketsControllerTest (7)	109 ms

3.21. ábra. Az egységtesztek eredménye

A második tesztelési mód amit alkalmaztam a rendszertesztelest. Ennek lényege, hogy elvárt viselkedéseket definiálunk, és megvizsgáljuk, hogy azok megfelelően működnek-e. Ennek eredményét a 3.12. táblázatban láthatjuk. Ebben egységesen az asztali alkalmazás illetve a weboldal is tesztelésre került.

Teszteset	Elvárt viselkedés	Megfelelt?
Helyes felhasználónév és jelszó megadás után a bejelentkezés gombra kattintunk.	Sikeresen bejelentkezünk a weboldalra.	Igen

Kiválasztunk egy tetszőleges filmet.	Az oldal befrissül és a választott film adatai megjelenítődnek.	Igen
A film adatlapján a vélemény hozzáadása gombra kattintunk, bejelentkezett felhasználó esetén.	A vélemény hozzáadására használt felület megjelenik.	Igen
A film adatlapján a vélemény hozzáadása gombra kattintunk, de nincs bejelentkezett felhasználó.	Hibaüzenetet kapunk, mely tájékoztat arról, hogy jelentkezzünk be.	Igen
A vélemény hozzáadása felületet megfelelően kitöltöttük, és a hozzáadás gombra kattintunk.	Az új vélemény megjelenik a film adatlapján.	Igen
Az előadások felületen kiválasztunk egy tetszőleges előadást, bejelentkezett felhasználóval.	Az előadáshoz tartozó felület jelenik meg a kijelzőn.	Igen
Az előadások felületen bejelentkezés nélkül kiválasztunk egy tetszőleges előadás.	Hibaüzenet formájában értesít a felület, hogy jelentkezzünk be.	Igen

A kívánt helyek kiválasztása után a foglalás gombra kattintunk.	Sikeres foglalás történik.	Igen
A regisztrációs felületen minden adatot megfelelően kitöljtük és véglegesítjük a regisztrációt.	Sikeres regisztráció történik.	Igen
A regisztrációs felületen egyes adatokat nem töltünk ki és véglegesítjük a foglalást.	A hiányzó mezők mellett megjelenik egy kitöltésre felszólító üzenet.	Igen
Az asztali alkalmazás bejelentkező felültén megfelelő adatokat adunk meg.	Sikeres bejelentkezés történik.	Igen.
Sikeres bejelentkezés után a szerepkör választásnál kiválasztunk egy rangot.	A kívánt szerepkörhöz tartozó felület jelenik meg.	Igen
A szerepkör választó felületen szerepkörök jelennek meg	Az aktuális felhasználó szerepkörei olvashatóak a listában.	Igen
Egy választott film esetén a kép módosító gombra kattintunk	Megjelenik a rendszer által használt fájl kezelő fület.	Igen.

A filmek felületen a statisztikát megjelenítő gombra kattintunk	Elérhetővé válik a kívánt felület.	Igen.
A jegyeladás felületre váltunk	A felületen az azonos filmek azonos színnel jelennek meg.	Igen.
Az előadás felületen a múlt heti előadásokat tekintjük meg	Megfelelő előadások jelenítődnek meg a felületen.	Igen.
Az egyes tab-ikonok az adatmódosítás során ki-hagyunk egy adatot	Megfelelő validációs hibaüzenet jelenítődik meg.	Igen.
A termék eladás felületen egy termékből többet választunk ki, mint amennyi a raktárban található, majd azt értékesítjük.	Hibaüzenet lép fel.	Igen.
Az alkalmazás bezárásával kijelentkezünk.	Megfelelően mentésre kerül a kijelentkezésünk.	Igen.

3.12. táblázat. Fekete-doboz tesztek

4. fejezet

Összegzés

Szakdolgozatom témája egy olyan rendszer készítése volt, amely egy mozihálózat működését segíti. Ehhez tartozóan készült egy weboldal, ahol a mozilátogatók tudnak jegyet foglalni egy előadásra, illetve véleményt fogalmazhatnak meg egy-egy filmről. Valamint készült egy asztali alkalmazás is amely az alkalmazottak igényeit szolgálja ki. Ezekhez pedig összekötő kapocsként készült egy *API* is, amely két nagyobb programrész logikájaként szolgál.

A szakdolgozat fejlesztése alatt tovább bővítettem tudásomat mind a webfejlesztés irányába, mind pedig az asztali alkalmazások felé. Igaz tanulmányaim során már volt szerencsém az *ASP.NET Core* keretrendszerrel, illetve az Entity Framework Core-ral megismerkedni, viszont ez idő alatt, míg a szakdolgozatomon dolgoztam sikerült ezeket a rendszereket még jobban megismerni és elsajátítani.

A webfejlesztésnél használt *Vue.js* keretrendszer hatalmas segítséget nyújtott a weboldal létrehozásában, örülök, hogy erre esett a választásom. Alapvetően nem szerettem sosem a weboldalak fejlesztését, viszont ez a keretrendszer olyan dolgokat adott hozzá, amelyek megkedveltették velem ezt a folyamatot.

Mindent együttvéve sokat tanultam ebből a pár hónap folyamatos fejlesztésből. Már ismert technológiákba ástam bele mélyebben magam, illetve új dolgokat is tanultam. Ezeken kívül sikerült megértenem, hogy mennyit tud könnyíteni a fejlesztési folyamatokon, ha egy alkalmazás jól meg van tervezve, illetve megfelelően strukturált. Úgy gondolom, hogy a fejlesztés közben szerzett tapasztalatokból nagyon sokat profitáltam a későbbiekre nézve.

4.0.1. Továbbfejlesztési lehetőségek

Egy program sosincs kész. Főképp egy olyan program amelyet az év minden napján használnak. Éppen ezért az ilyen programok esetében szükség van továbbfejlesztésre. Úgy gondolom, hogy sikerült olyan kódot írnom, amelyet könnyen és gyorsan tovább lehet fejleszteni.

A következőkben vázlatpont szerűen szeretnék egy pár dolgot felsorolni, amellyel ezt az alkalmazást ki lehetne bővíteni a későbbiekben.

- Olyan alkalmazott felület létrehozása, amely akár érintőképernyős eszközökön is működik.
- Több fajta statisztika készítése, akár diagram formájában is.
- Beosztás igény készítése az alkalmazáson belül az alkalmazottak számára.
- Chat felület létrehozása az alkalmazottaknak, ahol a munkájukkal kapcsolatos kérdéseket tehetik fel egymásnak.
- Kasszában lévő pénz nyilvántartása.
- Súgó az alkalmazottak számára.

Irodalomjegyzék

- [1] *WinRar*. <https://www.win-rar.com/start.html?&L=0>. Utolsó elérés: 2022-05-13.
- [2] *.Net 6*. <https://docs.microsoft.com/en-us/dotnet/core/whats-new/dotnet-6>. Utolsó elérés: 2022-05-13.
- [3] *Node.js hivatalos oldala*. <https://nodejs.org/en/>. Utolsó elérés: 2022-05-13.
- [4] *A REST API jelentése*. <https://www.ibm.com/docs/hu/mam/7.6.1?topic=apis-rest-api>. Utolsó elérés: 2022-05-13.
- [5] *A JavaScript nyelv*. <https://en.wikipedia.org/wiki/JavaScript>. Utolsó elérés: 2022-05-13.
- [6] *A Vue.js hivatalos oldala*. <https://vuejs.org/>. Utolsó elérés: 2022-05-13.
- [7] *A C# nyelv*. [https://en.wikipedia.org/wiki/C_Sharp_\(programming_language\)](https://en.wikipedia.org/wiki/C_Sharp_(programming_language)). Utolsó elérés: 2022-05-13.
- [8] *ASP.Net Core keretrendszer*. <https://dotnet.microsoft.com/en-us/learn/aspnet/what-is-aspnet-core>. Utolsó elérés: 2022-05-13.
- [9] *A HTML nyelv*. <https://developer.mozilla.org/en-US/docs/Web/HTML>. Utolsó elérés: 2022-05-13.
- [10] *ASP.NET keretrendszer*. <https://dotnet.microsoft.com/en-us/apps/aspnet>. Utolsó elérés: 2022-05-13.
- [11] *MVC architektúra*. <https://hu.education-wiki.com/3886982-what-is-mvc>. Utolsó elérés: 2022-05-13.
- [12] *JSON*. <https://en.wikipedia.org/wiki/JSON>. Utolsó elérés: 2022-05-13.
- [13] *Az MVVM architektúra leírása*. <https://bhawk.hu/mi-fan-terem-az-mvvm/>. Utolsó elérés: 2022-05-13.
- [14] *Az SQL*. <https://en.wikipedia.org/wiki/SQL>. Utolsó elérés: 2022-05-13.

- [15] *Az Entity Framework Core.* <https://docs.microsoft.com/en-us/ef/core/>. Utolsó elérés: 2022-05-13.
- [16] *Az Entity Framework bemutatása.* <https://www.entityframeworktutorial.net/what-is-entityframework.aspx>. Utolsó elérés: 2022-05-13.
- [17] *Visual Studio.* <https://visualstudio.microsoft.com/>. Utolsó elérés: 2022-05-13.
- [18] *A WebStorm IDE hivatalos oldala.* <https://www.jetbrains.com/webstorm/>. Utolsó elérés: 2022-05-13.
- [19] *GitHub leírása.* <https://kinsta.com/knowledgebase/what-is-github/>. Utolsó elérés: 2022-05-13.
- [20] *Postman.* <https://www.postman.com/>. Utolsó elérés: 2022-05-13.
- [21] *NuGet Package leírása.* <https://docs.microsoft.com/en-us/nuget/what-is-nuget>. Utolsó elérés: 2022-05-13.
- [22] *DTO jelentése.* https://en.wikipedia.org/wiki/Data_transfer_object. Utolsó elérés: 2022-05-13.
- [23] *UML fogalma.* https://en.wikipedia.org/wiki/Unified_Modeling_Language. Utolsó elérés: 2022-05-13.
- [24] *Vue ismertető.* <http://faragocsaba.hu/vue>. Utolsó elérés: 2022-05-13.
- [25] *A CSS nyelv.* <https://en.wikipedia.org/wiki/CSS>. Utolsó elérés: 2022-05-13.
- [26] *vue-router.* <https://router.vuejs.org/>. Utolsó elérés: 2022-05-13.
- [27] *Vuex.* <https://vuex.vuejs.org/>. Utolsó elérés: 2022-05-13.

Ábrák jegyzéke

2.1. "Home" oldal	7
2.2. Választott mozifilm, kinyitott előadások füllel	7
2.3. Hibaüzenet, az előadások hiányáról	8
2.4. Teljes oldalas kép a filmek adatlapjáról, véleményekkel	8
2.5. Menüsáv a lenyíló User menüvel	9
2.6. Regisztrációs felület	9
2.7. Bejelentkezési felület	10
2.8. Vélemény hozzáadása	11
2.9. "Movies" oldal	12
2.10. "Programs" oldal	13
2.11. Hibaüzenet, ha nincs előadás a választott napra	13
2.12. Reserve felület, kiválasztott helyekkel	14
2.13. Bejelentkezésre szolgáló felület	15
2.14. Szerepköröket kiválasztó ablak	15
2.15. "Refresh list" gomb	16
2.16. Validációs hibaüzenet helytelenül kitöltött mezőkről	17
2.17. A mozi felület	17
2.18. A mozi felület elérhető szerkesztővel	18
2.19. A lefrissült terem lista	18
2.20. Új elem hozzáadása a "Details" szekcióban	19
2.21. Sikeres hozzáadás eredménye	19
2.22. A filmek felület	19
2.23. Képpel rendelkező film	20
2.24. Film hiányos képpel	20
2.25. Egy film színészei és kategóriái az ezekhez tartozó beviteli mezőkkel .	21
2.26. A filmek statisztikai felülete véleményekkel	22
2.27. Az előadások felület	23

2.28. Az alkalmazottak felülete a szerepkörökkel	24
2.29. Egy felhasználó be- és kijelentkezési adatai a "Details" szekcióban	24
2.30. A jegyeladásokhoz tartozó előadásokat megjelenítő felület	25
2.31. A jegyeladások tényleges felülete	26
2.32. A raktárat reprezentáló felület	27
2.33. A termékeket értékesítő felület	28
2.34. Hibaüzenet, ha a kért számú termék értékesítése nem lehetséges	28
2.35. A termékek statisztikáját bemutató ablak	29
2.36. A pénztár területei vezető felületének főképernyője	30
2.37. A büfé területi vezetőjének főképernyője	31
 3.1. Az alkalmazások kommunikációja	32
3.2. Az API architektúrája	35
3.3. A modell architektúrája	54
3.4. A DTO osztályok egyik fele	55
3.5. A DTO osztályok másik fele	55
3.6. A modell többi osztálya	56
3.7. Migrációs állományok	57
3.8. Migrációs állományok	59
3.9. Az asztali alkalmazás architektúrája	66
3.10. Az alkalmazás osztálydiagramma	67
3.11. Az alkalmazás UML diagramja	68
3.12. A vezérlés osztálydiagramja	70
3.13. Az modell osztálydiagramja	71
3.14. A nézetmodell egyik osztálydiagramja	72
3.15. Az nézetmodell másik osztálydiagramja	72
3.16. Az nézet réteg osztálydiagramja	74
3.17. A webes alkalmazás architektúrája	75
3.18. A weboldal UML diagramja	77
3.19. A "views" mappa tartalma	81
3.20. A "components" mappa tartalma	81
3.21. Az egységesztek eredménye	83

Táblázatok jegyzéke

3.1.	Az ActorsController végpontjainak leírása	39
3.2.	Az BuffetWarehouseController végpontjainak leírása	41
3.3.	Az ActorsController végpontjainak leírása	42
3.4.	Az EmployeeController végpontjainak leírása	44
3.5.	Az MoviesController végpontjainak leírása	46
3.6.	Az OpinionsController végpontjainak leírása	47
3.7.	Az RentsController végpontjainak leírása	48
3.8.	Az RoomsController végpontjainak leírása	49
3.9.	Az ShowsController végpontjainak leírása	51
3.10.	Az StatsAndPaysController végpontjainak leírása	52
3.11.	A TicketsController végpontjainak leírása	54
3.12.	Fekete-doboz tesztek	86