Szakdolgozat

Miskolci Egyetem

Közösségi Videómegosztó Szoftver Tervezése és Implementálása

Készítette:

Nagy Máté

Programtervező Informatikus

Témavezető:

Fazekas Levente Áron

Miskolc, 2024

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

Szakdolgozat Feladat

Nagy Máté (U3ROFS) programtervező informatikus jelölt részére.

A szakdolgozat tárgyköre: Webtechnológiák

A szakdolgozat címe: Közösségi Videómegosztó Szoftver Tervezése és Implementálása

A feladat részletezése:

1. A szakdolgozat célja egy valós idejű videómegosztásra alkalmas webes applikáció tervezése és implementálása.
2. Mutassa be a WebSocket és egyéb webes, valós idejű kommunikációra alkalmas API-kat!
3. Tervezzen meg és implementáljon egy szoftvert, amely képes az alábbi funkciók elvégzésére:
   (a) Felhasználós nyilvántartása
   (b) Videómegosztásra alkalmas szobák létrehozása, nyilvántartása
   (c) Egyidejű vetítés azonos szobában tartózkodó felhasználók számára
   (d) Felhasználók közötti chat
4. Mutassa be az esetleges továbbfejlesztési lehetőségeket!

Témavezető: Fazekas Levente Áron (tanszéki mérnök)

A feladat kiadásának ideje: 2023.09.01.

. . . . . .
szakfelelős

Eredetiségi Nyilatkozat

Alulírott **Nagy Máté**; Neptun-kód: `U3ROFS` a Miskolci Egyetem Gépészmérnöki és Informatikai Karának végzős Programtervező informatikus szakos hallgatója ezennel büntetőjogi és fegyelmi felelősségem tudatában nyilatkozom és aláírásommal igazolom, hogy *Közösségi Videómegosztó Szoftver Tervezése és Implementálása* című szakdolgozatom saját, önálló munkám; az abban hivatkozott szakirodalom felhasználása a forráskezelés szabályai szerint történt.

Tudomásul veszem, hogy szakdolgozat esetén plágiumnak számít:

- szószerinti idézet közlése idézőjel és hivatkozás megjelölése nélkül;
- tartalmi idézet hivatkozás megjelölése nélkül;
- más publikált gondolatainak saját gondolatként való feltüntetése.

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

Miskolc, ... év ..... hó ..... nap

.....
Hallgató

1. A szakdolgozat feladat módosítása

szükséges (módosítás külön lapon)
nem szükséges

....
dátum témavezető(k)

2. A feladat kidolgozását ellenőriztem:

témavezető (dátum, aláírás):	konzulens (dátum, aláírás):
.....	.....
.....	.....
.....	.....

3. A szakdolgozat beadható:

....
dátum témavezető(k)

4. A szakdolgozat ..... szövegoldalt
..... program protokollt (listát, felhasználói leírást)
..... elektronikus adathordozót (részletezve)
.....
..... egyéb mellékletet (részletezve)
.....
tartalmaz.

....
dátum témavezető(k)

5. A szakdolgozat bírálatra

bocsátható
nem bocsátható

A bíráló neve: .....

....
dátum szakfelelős

6. A szakdolgozat osztályzata

a témavezető javaslata: .....
a bíráló javaslata: .....
a szakdolgozat végleges eredménye: .....

Miskolc, ...
a Záróvizsga Bizottság Elnöke

Tartalomjegyzék

1. fejezet

Bevezetés

A digitális korban élünk, ahol a technológia gyorsan fejlődik és az embereknek egyre több lehetőségük van az online interakcióra. Azonban a közösségi médiával és az egyéb digitális platformokkal eltöltött idő gyakran személytelen és egyoldalú.

A "WatchWithFriends" névre keresztelt alkalmazásom arra hivatott, hogy ezen változtasson. Az alkalmazás lehetőséget nyújt arra, hogy a felhasználók együtt nézzenek YouTube-videókat, miközben élő chaten kommunikálnak egymással. Az alkalmazásban létrehozott "szobák" révén a felhasználók egyszerűen csatlakozhatnak, és megoszthatják egymással kedvenc videóikat egy közös lejátszási listán keresztül. Az alkalmazás nem csupán egy online videómegosztó platformot egészít ki, hanem egy új közösségi élményt is kínál. Ezzel az eszközzel a barátok és családok, akár távoli földrészeken is, összehozhatók egy közös élmény kapcsán. A "WatchWithFriends" az együtt töltött idő minőségét kívánja javítani, lehetővé téve, hogy az emberek valós időben osszák meg reakcióikat, érzéseiket egy-egy videóval kapcsolatban.

Továbbá a chat funkcióval a felhasználók azonnal megvitathatják a látottakat, megoszthatják tippeiket vagy akár következő videó választási ötleteiket is.

A szoftver nem csak szórakoztató, de oktatási célokra is felhasználható. Tanárok és diákok könnyedén nézhetnek együtt oktatási anyagokat, és azonnal megbeszélhetik azok tartalmát. Ezen kívül a vállalati prezentációknál is hasznos lehet a közös videómegtekintés funkció.

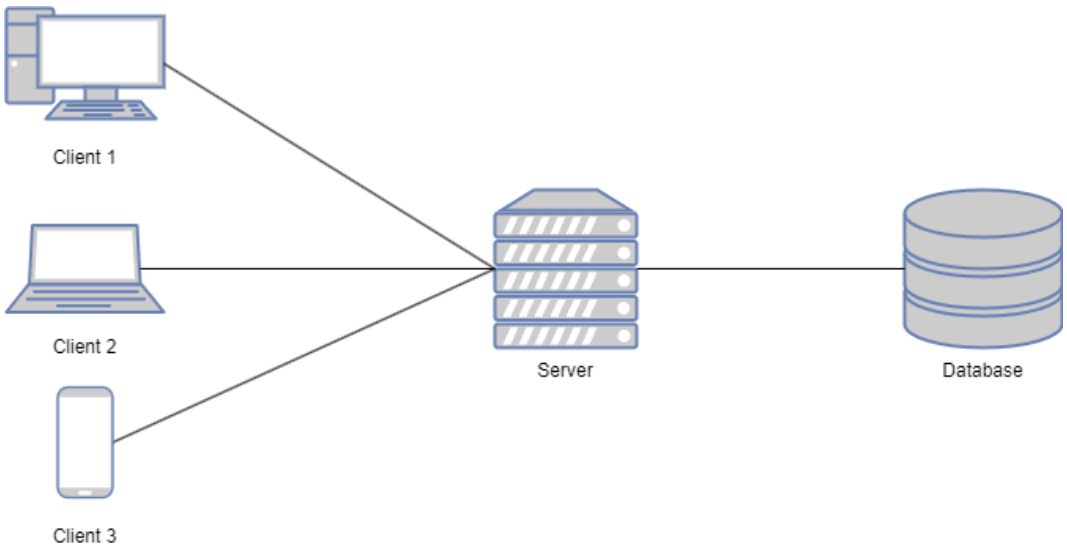
A "WatchWithFriends" egy olyan platform, ami a társasági interakciókat nem csak kiterjeszti, de egy új dimenzióba is emeli. Az alkalmazás tehát nem csak egy technológiai újítás, hanem egy közösségi eszköz is, amely összeköti az embereket függetlenül fizikai távolságuktól.

2. fejezet

Tervezés

Átfogó struktúra

A kliens-szerver-adatbázis struktúrát 3 rétegű topológiának nevezzük. A 3-rétegű architektúra előnye, hogy modularizálja az alkalmazás kódját, ami megkönnyíti a karbantartást, a tesztelést és a skálázhatóságot.



2.1. ábra: Kliens-Szerver-Adatbázis struktúra

Kliens réteg

A kliens oldal felelős a felhasználói interakciókért és a felhasználói felület megjelenítéséért. Itt találhatók a gombok, formok, és egyéb elemek, amikkel a felhasználók közvetlenül érintkeznek. A kliens oldalon futó kód gyakran aszinkron módon kommunikál a szerverrel, hogy adatokat kérjen vagy küldjön.

Szerver réteg

A szerver oldal felelős a kliens oldal által küldött kérések feldolgozásáért, és a válaszok generálásáért. Itt található az üzleti logika, és az adatbázis kezelés. A szerver oldali kód

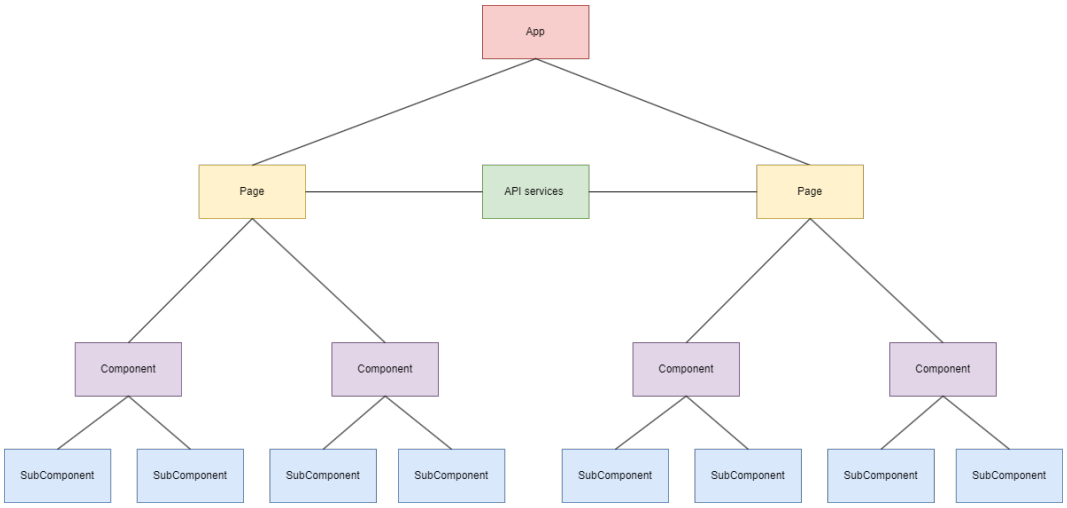
futása általában szinkron módon történik, és a kliens oldali kérések feldolgozása után küldi a válaszokat.

Adatbázis réteg

Ez a réteg tárolja az alkalmazás állapotát és adatát. Gyakran SQL (pl. MySQL, PostgreSQL) vagy NoSQL (pl. MongoDB) adatbázisokat használnak itt.

Kliens oldali logika

A kliens oldali logika szigorú hierarchikus struktúra foglalja magába. Az alkalmazáson belüli komponensek mind horizontálisan, mind vertikálisan szerveződnek.



2.2. ábra: Frontend Architektúra

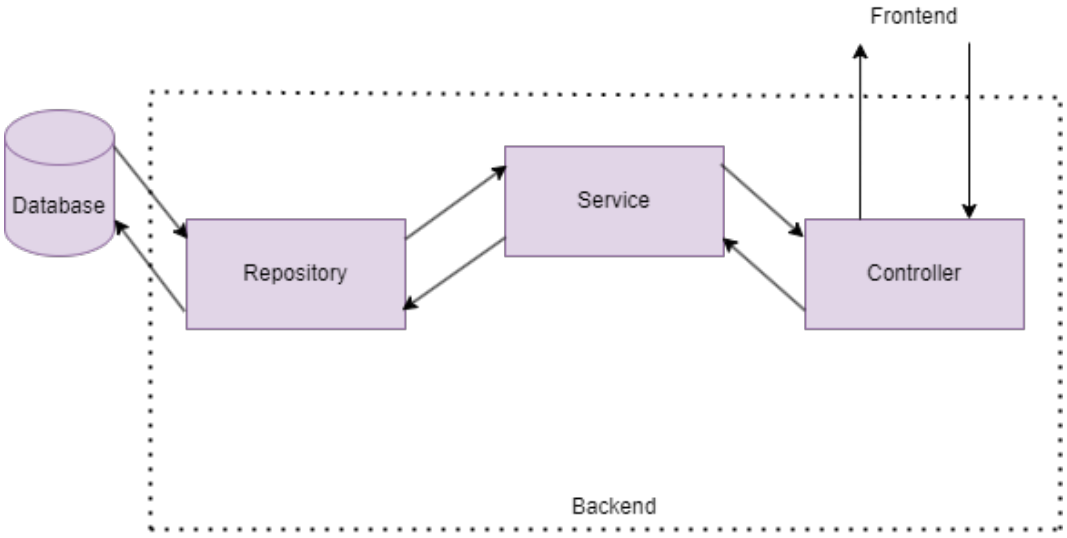
A fejlesztés során az egyik fő szempont a komponensek újrafelhasználhatósága, kód strukturáltsága volt.
A komponensek főbb tulajdonságait:

- **App:** Az alkalmazás gyökér komponense. Itt található a Router, amely a különböző útvonalakhoz rendeli a megfelelő komponenseket.
- **Page:** Az alkalmazás View komponensei. A különböző oldalakat reprezentálják. Itt találhatóak a különböző komponensek, amelyek a megjelenítést végzik. Illetve itt használom az API hívásokat is.
- **API services:** Itt találhatóak a különböző API végpontokat kezelő függvények.
- **Component:** Az alkalmazás nagyobb komponensei. Ezek a komponensek több kisebb komponensből állnak össze.
- **SubComponent:** Az alkalmazás kisebb komponensei. Ezek a komponensek csak egy adott feladatot látnak el.

Szerver oldali logika

Architektúra

Az alkalmazásnál MVC (Model-View-Controller) architektúrára építettem, illetve bővítettem a Repository és a Service rétegekkel. Az MVC architektúra egy architektúrális minta, amely a felhasználói felületet (View), az alkalmazás logikáját (Controller) és az adatokat (Model) három különálló részre osztja. A Repository réteg a Model réteghez tartozik, a Service réteg pedig a Controller réteghez. A további bontást a felelősségi körök elkülönítése, kód átláthatóságának növelése, a bővíthetőség és a tesztelhetőség indokolta.



2.3. ábra: Backend Architektúra

A rétegek főbb tulajdonságait:

- **Adatbázis**: Az alkalmazás adattára. Itt tárolódnak a statikus és dinamikus adatok.
- **Repositoryk**: Az adatbázis és az alkalmazás logikája közötti köztes réteg. CRUD műveletek, komplex lekérdezések.
- **Servicek**: Az üzleti logika helye. Itt történnek a komplex számítások, validációk és egyéb logikai műveletek.
- **Controllerek**: Az API végpontokat kezelő réteg. Fogadják a kliens kéréseit, és válaszokat generálnak.

Adatbázis

ER modell egyedei és tulajdonságai

User-ek tulajdonságai:

- Id: Elsődleges kulcs, Guid típusú.
- Name: Felhasználó neve, szöveg típusú.
- Email: Felhasználó email címe, szöveg típusú.
- PasswordHash: Felhasználó jelszavának hash-elt változata, szöveg típusú.
- Salt: Felhasználó jelszavának sója, szöveg típusú.
- BirthDate: Felhasználó születési dátuma, dátum típusú.

Image-ek tulajdonságai:

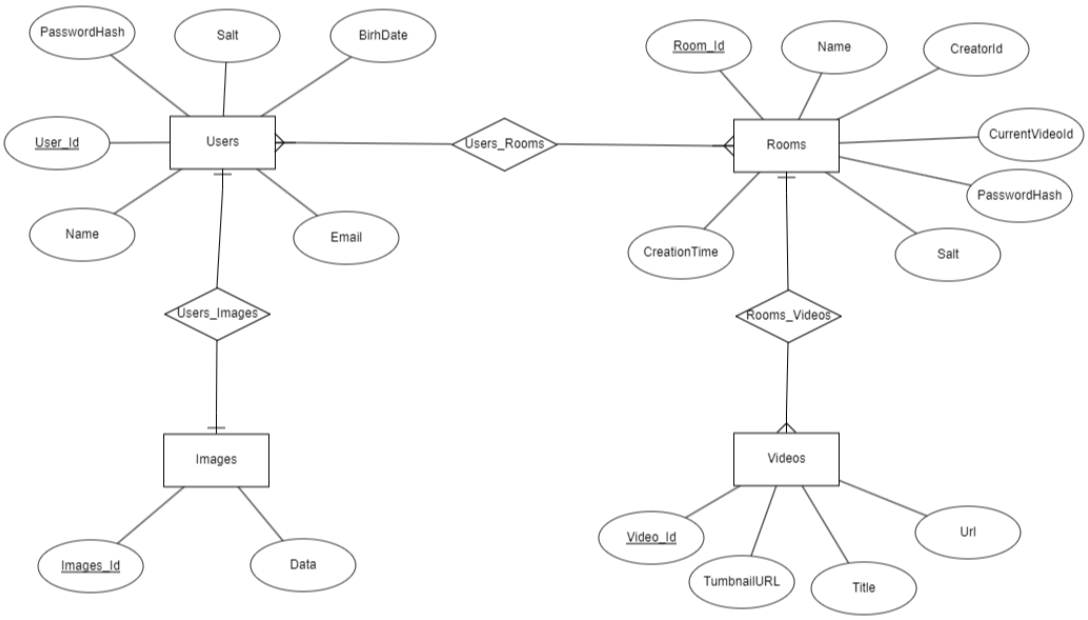
- Id: Elsődleges kulcs, Guid típusú.
- Data: Kép binárisa, byte tömb típusú.

Room-ok tulajdonságai:

- Id: Elsődleges kulcs, Guid típusú.
- Name: Szoba neve, szöveg típusú.
- CreatorId: Tulajdonos azonosítója, Guid típusú.
- CreationTime: Szoba létrehozásának ideje, dátum típusú.
- CurrentVideoId: Jelenleg lejátszott videó azonosítója, Guid típusú.
- PasswordHash: Szoba jelszavának hash-elt változata, szöveg típusú.
- Salt: Szoba jelszavának sója, szöveg típusú.

Video-k tulajdonságai:

- Id: Elsődleges kulcs, Guid típusú.
- Title: Videó címe, szöveg típusú.
- Url: Videó URL-je, szöveg típusú.
- ThumbnailURL: Videó thumbnail-je, szöveg típusú.

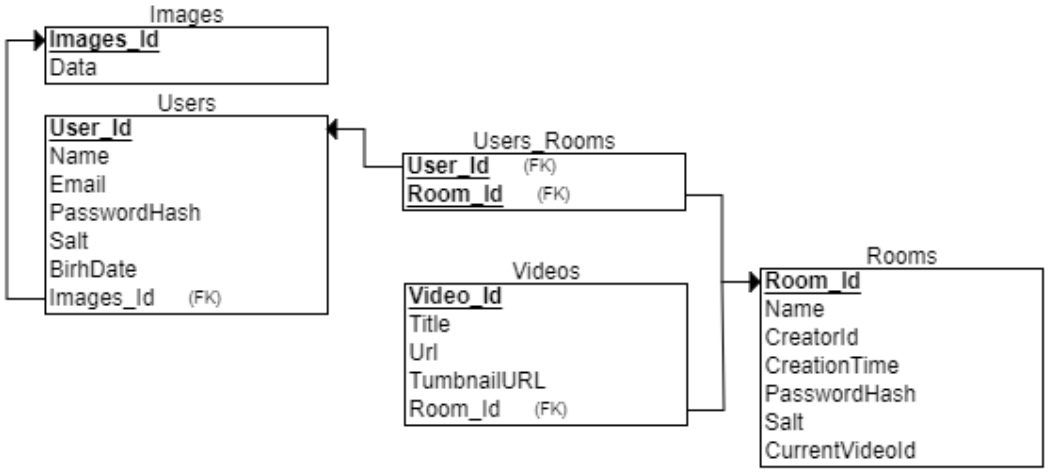


2.4. ábra: ER modell

Relációs modell

Az egyedek közötti kapcsolatok:

- Users-Images: 1:1 Egy felhasználóhoz tartozhat egy kép, és a kép csak egy felhasználóhoz.
- Users-Rooms: N:N Egy felhasználóhoz tartozhat több szoba, egy szoba is tartozhat több felhasználóhoz.
- Room-Video: 1:N Egy szobához tartozhat több videó, de egy videó csak egy szobához tartozhat.



2.5. ábra: Relációs modell

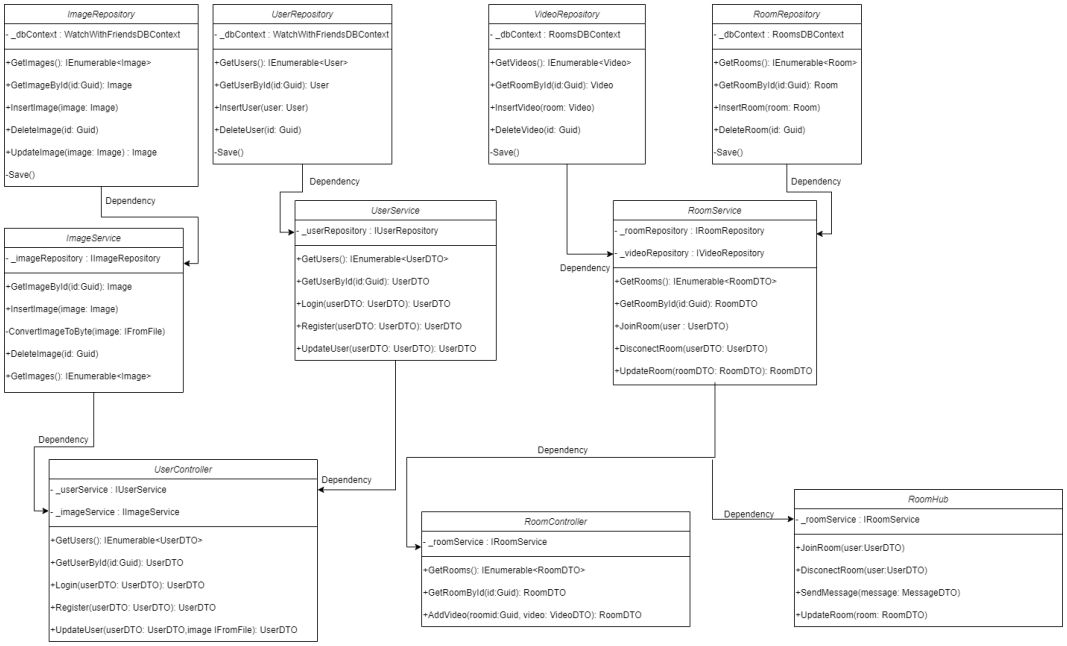
Szoftvertervezési elvek

A szoftvertervezési elvek alapvető útmutatások a kiváló kód fejlesztéséhez. Ezek az iránymutatások növelik a kód minőségét, gyorsítják a fejlesztési folyamatot és minimalizálják a hibalehetőségeket. Az ilyen elvek segítenek abban, hogy a kód könnyen újrafelhasználható és karbantartható legyen.

A kód olvashatóságát és tesztelhetőségét is javítják, ami hosszú távon időt és erőforrást spórol. A rugalmasság és a bővíthetőség is nő, tehát a kód könnyen adaptálható új funkciók vagy változások esetén. Továbbá, ezek az elvek ösztönzést nyújtanak a kódbázis redundancia kiküzsöbölésére, amely csökkenti a komplexitást és növeli a kohéziót. Az elvek célja a kód moduláris felépítése és az alacsonyabb összekapcsoltság, ami hozzájárul a jobb szervezettséghez és könnyebb karbantartáshoz.

Összességében, a szoftvertervezési elvek olyan iránymutatások, amelyek célja a hatékony, karbantartható és minőségi kód létrehozása. A szoftver implementációja során a SOLID elveket vettem alapul.

UML diagram



2.6. ábra: UML Diagram

Az architektúra alapját egy kibővített MVC (Model-View-Controller) rendszer adja, amely kifinomult kapcsolatokat biztosít az egyes szerveroldali osztályok között. Mivel a back-end és a front-end kapcsolata indirekt, ezért a megjelenítési oldalon található logika nincs semmilyen ráhatással a szerver struktúrális felépítésére. Ebben az elrendezésben az egyes osztályok nem csak az alapvető CRUD (Create, Read, Update, Delete) műveletekért felelnek, hanem a komplex üzleti logika egyedi implementációját is lehetővé teszik. Az adatmodell és a kontroller között az adatátviteli objektumok (DTO-k) és a repository minták segítenek a moduláris és tiszta kód

struktúra fenntartásában. A repository-k pedig közvetlenül a dedikált adatbázis-kontextusokhoz kapcsolódnak.

A kapcsolatok közötti szoros integráció lehetővé teszi az adatok egyszerű és következetes kezelését, miközben fenntartja a kód karbantarthatóságát és tesztelhetőségét. Az egész rendszer ezen az elrendezésen alapul, így garantálva, hogy a szerveroldali logika jól szervezett és hatékony maradjon.

3. fejezet

Technológia

3.1. Felhasználói felület

Az alkalmazás felhasználói felületét a React [1] könyvtár segítségével valósítottam meg. A React egy nyílt forráskódú JavaScript könyvtár, amelyet a Facebook fejlesztett ki. A könyvtár célja, hogy a felhasználói felület fejlesztését egyszerűbbé tegye. A React egy komponens alapú könyvtár, amely lehetővé teszi a fejlesztők számára, hogy újra felhasználható komponenseket hozzanak létre, amelyeket összeállítva komplex felhasználói felületeket hozhatnak létre.

Komponensek A React komponensek egyfajta sablonok, amelyek a felhasználói felület egy részét írják le. A komponensek egy egyszerű JavaScript objektumok, amelyeknek van egy render metódusa, amely visszaadja a komponens felhasználói felületét.

Komponensek főbb tulajdonságai:

- Paraméterek (props)
- Gyerekei (children)
- Állapotai (state)
- Életciklus metódusai (lifecycle)
- Eseménykezelői (event)
- Stílusai (style)
- Referenciák (ref)
- Kontextusai (context)
- Típusai (type)
- Kulcsai (key)

React és Angular összehasonlítása

A front-end keretrendszer választása fontos lépése az implementáció előkészítésének. Két népszerű keretrendszer képezte a lehetőségek listáját: az Angular [2] és a React [1].

Különbség a React és az Angular között:

3.1. táblázat: Különbségek React és Angular között

	React	Angular
Egyszerűség	Könnyű tanulni.	Több beépített funkció.
Flexibilitás	Rugalmas.	Kevesebb szabadság.
Teljesítmény	Virtuális DOM.	Two-Way Data Binding.
Ökoszisztéma	Nagy közösség.	Minőségi eszközök.
Fejlesztői tapasztalat	JSX.	TypeScript.

Redux vagy Context API

Az alkalmazás állapotainak kezelésére több lehetőség is felmerült, közöttük a Redux és a Context API. A Redux [3] egy állapotkezelő könyvtár, amely lehetővé teszi az alkalmazás állapotának tárolását egy központi helyen. A Context API egy React API, amely lehetővé teszi az alkalmazás állapotának tárolását és megosztását a komponensek között. Mivel az alkalmazás kis méretű, ezért a Context API-ra esett a választás.

A Context API főbb tulajdonságai:

- **Beépített megoldás:** A Context API része a React alapkönyvtárnak, nincs szükség külső függőségekre.
- **Egyszerűség:** A Context API egyszerűbb és könnyen megérthető, kevesebb boilerplate kóddal.
- **Nincs middleware:** Nem támogat middleware-eket alapból, így aszinkron állapotfrissítéseket manuálisan kell kezelni.
- **Nem optimalizált nagy alkalmazásokhoz:** Nagy alkalmazásokban hatékonytalan lehet, mert minden Context változásra újrarendereli az összes fogyasztót, hacsak nem optimalizáljuk manuálisan.
- **Kevesebb eszköz:** Kevesebb beépített eszköze van, mint a Redux-nak, de ez nem mindig hátrány, függ a projekt igényeitől.

Példa a Context API állapotkezelésére:

```
// context
const CounterContext = React.createContext();

// state and update
const [count, setCount] = useState(0);
```

```
// state refresh
setCount(count + 1);
```

Programkód 3.1. Context API állapotkezelése

A Redux főbb tulajdonságai:

- **Optimalizáció:** Redux kifejezetten optimalizált a nagyobb alkalmazások számára, és lehetővé teszi az állapot frissítéseinek finom szabályozását.
- **Middleware Támogatás:** Redux lehetőséget ad middleware-ek használatára, amelyek lehetővé teszik az aszinkron állapotfrissítések könnyebb kezelését.
- **Tesztelhetőség:** A Redux architektúrája miatt a tesztelés egyszerűbb, minden action egy független egység, és a reducer funkciók tiszta függvények.
- **Eszközök és Közösség:** Redux-nak nagy közössége van, és sok kiegészítő eszköz, például a Redux DevTools.
- **Boilerplate Kód:** Több boilerplate kód szükséges, hogy elindítsunk egy Redux-alapú állapotkezelést.

Példa a Redux állapotkezelésére:

```
// action
const incrementAction = { type: 'INCREMENT' };

// reducer
function counterReducer(state = 0, action) {
  if (action.type === 'INCREMENT') {
    return state + 1;
  }
  return state;
}

// dispatch
dispatch(incrementAction);
```

Programkód 3.2. Redux állapotkezelése

TypeScript vagy JavaScript

A TypeScript egy nyílt forráskódú, szigorúan típusos programozási nyelv, amely a JavaScriptre épül, lehetővé teszi a statikus típusok használatát, és minden érvényes JavaScript kód érvényes TypeScript kódnak is számít. Inkább a TypeScript mellett döntöttem, mert a statikus típusok használata növeli a kód minőségét és a fejlesztési sebességet, illetve csökkenti a hiba lehetőségek számát.

3.2. Szerveroldali Logika és API-k

A szerver és a kliens közötti kommunikáció megvalósításához az ASP.NET Core-t használtam. Az ASP.NET Core egy nyílt forráskódú, cross-platform, magas teljesítményű

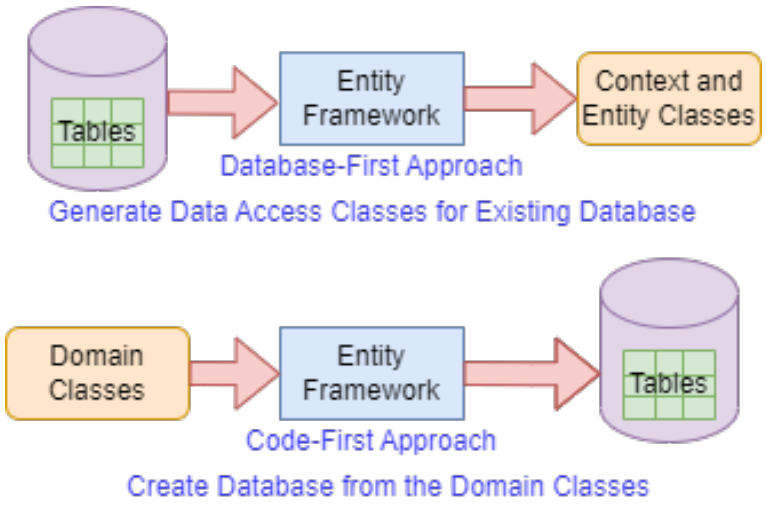
keretrendszer, amelyet a Microsoft fejlesztett ki. Az ASP.NET Core-t a webalkalmazások és a webes API-k fejlesztésére használják. A környezetet a C# programozási nyelvhez tervezték, de támogatja a többi .NET nyelvet is. Én a megvalósítás során a C# nyelvet használtam. Ebben a fejezetben bemutatom a szerveroldali logikát és az API-kat.

Adatbázis

Az alkalmazásnál MS SQL adatbázist használtam, mert a Microsoft SQL Server-t használom a személyes projekteimhez. Az MS SQL (Microsoft SQL Server) egy relációs adatbázis-kezelő rendszer (RDBMS) a Microsofttól. Jól skálázható, és számos fejlett funkcióval rendelkezik, mint például tárolt eljárások, triggerek és nézetek . Gyakran használják vállalati szintű alkalmazásokban, és támogatja a SQL nyelvet az adatok lekérdezésére és manipulálására. Támogatja az ACID tulajdonságokat és a tranzakciós integritást. Az ER modell-t használtam az adatbázis tervezéséhez. Ezt majd a Backend fejlesztési folyamatánál részletezem.

Entity Framework Core

Az Entity Framework Core [4] (EF Core) egy objektum-relációs leképzési (ORM) keretrendszer a Microsofttól, amely .NET Core és .NET 5+ alkalmazások számára készült. Lehetővé teszi a fejlesztők számára, hogy magas szintű, objektumorientált API-n keresztül dolgozzanak adatbázisokkal, anélkül hogy közvetlen SQL lekérdezéseket kellene írniuk. Támogatja a code-first és a database-first megközelítéseket, így rugalmasságot biztosít az adatmodell és az adatbázisséma kialakításában. Az EF Core lehetővé teszi a LINQ (Language Integrated Query) használatát, ami természetes módon illeszkedik a C# és más .NET nyelvekhez.



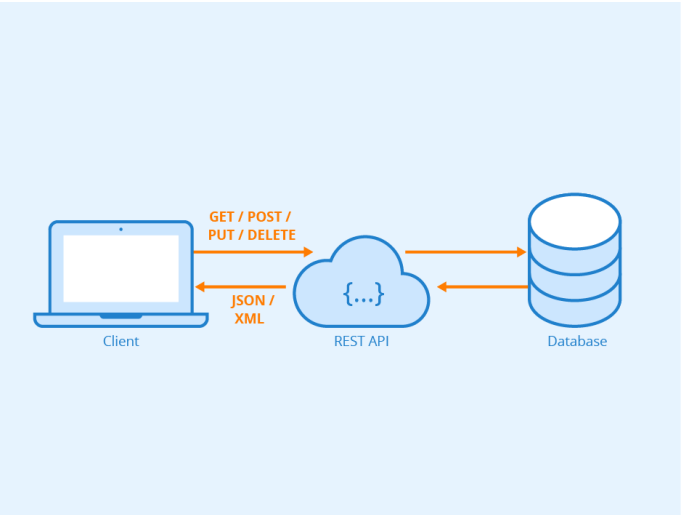
3.1. ábra: Entity Framework Core

Skálázható és teljesítmény-optimalizált, ezért alkalmas mind kis, mind nagyobb vállalati projektekben. Az EF Core támogatja a többszörös adatbázis-motorokat, beleértve az SQL Server-t, PostgreSQL-t, SQLite-ot és másokat is. Az adatmigrációk egyszerű kezelése és automatikus generálása az egyik előnye, ami könnyebbé teszi az adatbázis séma változásainak kezelését. A keretrendszer beépített támogatással rendelkezik a

tranzakciókezelésre, így az ACID tulajdonságok biztosítottak. Az EF Core rugalmas konfigurációs lehetőségekkel rendelkezik, és jól integrálódik más .NET Core szolgáltatásokkal, mint például a Dependency Injection. Folyamatosan fejlődik és aktívan karbantartott, így a legújabb .NET technológiákhoz és az adatbázis-technológiákhoz is gyorsan alkalmazkodik.

API

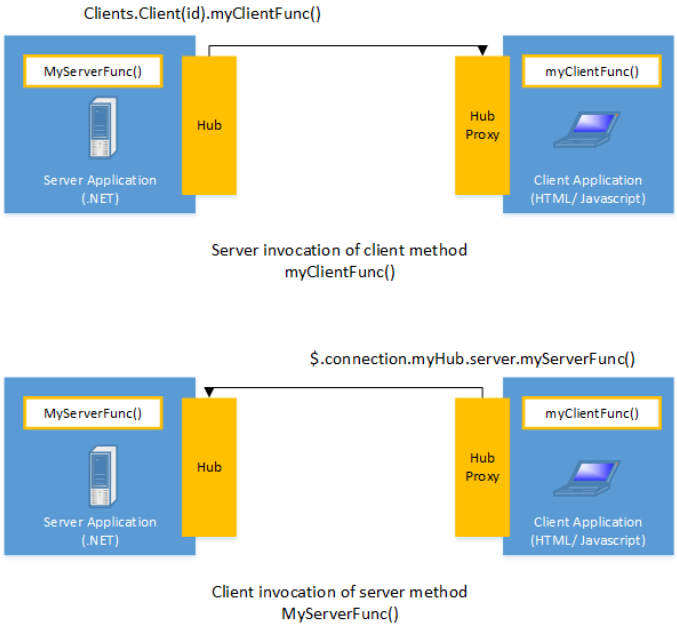
A komponensek közötti kommunikáció megvalósítására RESTful API-t használtam, annak népszerűsége miatt. A REST (Representational State Transfer) egy architektúrális stílus, amelyet a webes alkalmazásokhoz használnak. A RESTful API-kat a REST architektúra alapelvei alapján tervezik. A RESTful API-kat a HTTP protokollra építik, és a HTTP metódusokat használják a kérések kezelésére. Ennek megvalósításához a *ControllerBase* osztályt használtam a Controllerekben. Ami teljesen implementálja a RESTful API-kat, és a HTTP metódusokat.



3.2. ábra: RESTful API [5]

A videók lejátszása során különös figyelmet fordítottam az állapotkezelésre. Ennek érdekében a WebSocket protokollt alkalmaztam. A WebSocket egy kétirányú kapcsolatot tesz lehetővé a böngésző és a szerver között. Bár a HTTP protokollra épül, mégis képes valós idejű kommunikációra. A kommunikáció HTTP portokon keresztül zajlik, ami rugalmasságot ad az alkalmazásnak. SignalR-t használtam a WebSocket protokoll megvalósításához.

A SignalR egy Microsoft által fejlesztett aszinkron könyvtár, amely valós idejű webes alkalmazások építését támogatja. Lehetővé teszi a szerver és a kliensek közötti kétirányú kommunikációt, gyakran WebSockets használatával.

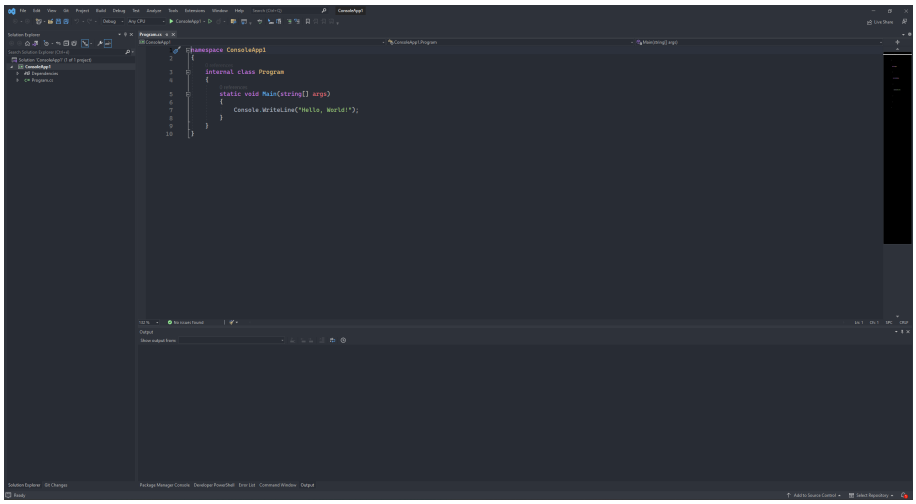


3.3. ábra: SignalR [6]

Ez eltér a hagyományos HTTP "kérés-válasz" modelltől, mivel a szerver aktívan küldhet üzeneteket a klienseknek. Gyakori alkalmazási területei közé tartoznak a chat szolgáltatások, online játékok és élő adatfrissítések. A SignalR automatikusan választja ki a legmegfelelőbb kommunikációs mechanizmust a szerver és a kliens között, például long polling, ha WebSockets nem érhető el. A könyvtár könnyen integrálható más .NET technológiákkal és keretrendszerekkel, például ASP.NET Core-al. A SignalR segítségével könnyen implementálhatók komplex forgatókönyvek, például csoportos üzenetküldés vagy kapcsolatok kezelése. Az állapotkezelés és a kiváló skálázhatóság további előnyei a SignalR használatának. Mivel a SignalR API az ASP.NET Core része, az infrastruktúrát és a biztonsági mechanizmusokat is könnyen ki lehet terjeszteni rá. Összességében a SignalR egy erőteljes eszköz a valós idejű webes alkalmazások fejlesztéséhez.

Fejlesztői környezet

Én a Visual Studio 2022-t [7] használtam a fejlesztéshez, mert ez az egyik legnépszerűbb IDE a .NET fejlesztők körében. A Visual Studio 2022 egy integrált fejlesztői környezet (Integrated Development Environment, IDE), amelyet a Microsoft fejlesztett ki. Ez a platform különösen népszerű a Windows alapú alkalmazások fejlesztésénél, de támogatja számos programozási nyelvet és technológiát, így a fejlesztők széles spektruma számára kínál megoldásokat. Az IDE magába foglalja a kódszerkesztést, a debuggolást, a verziókezelést és más, a fejlesztési ciklusban fontos eszközöket is.



3.4. ábra: Visual Studio 2022

Míg a Visual Studio elsősorban a Windows operációs rendszerre lett tervezve, a Microsoft kiadott egy könnyebb, keresztplatformos változatot is, a Visual Studio Code-ot. Ez utóbbi támogatja a Linux és macOS operációs rendszereket is, így a fejlesztők választhatnak az operációs rendszerüknek legmegfelelőbb változat közül.

Mindkét platform lehetővé teszi az együttműködést és a csapatmunkát, és számos bővítményt és sablont kínál a gyors és hatékony fejlesztés érdekében.

4. fejezet

Implementáció

Ebben a fejezetben a "WatchWithFriends" alkalmazás implementációs folyamatát mutatom be.

Szerver oldali logika

Entity Framework és Data projekt

Az adatintegritás és az adatmanipuláció kezelésének centralizálása érdekében létrehoztam egy különálló projektcsoportot, amely specifikusan az adatkezelési logika implementációjáért felel. Ebben a dedikált modulban helyeztem el az adatbázis tábláinak sémáját, valamint az adathozzáférési réteget képviselő repository-kat.
A modul architektúrája rugalmas és jól skálázható módon épül fel, köszönhetően az Entity Framework integrációjának. Az Entity Framework ORM (Object-Relational Mapping) képességeit kihasználva, egyszerűsítem és automatizálom az adatbázis műveleteket, csökkentve ezzel az ismétlődő kód és az esetleges hibák számát.
A modell definíciók és az adatbázis kontextus is ebbe a projektcsoportba kerültek, így biztosítva, hogy az adatmodellek és az adatbázis-sémák közötti szoros kapcsolat koherens és könnyen kezelhető maradjon. Az egész rendszer így válik áttekinthetővé és könnyen karbantarthatóvá.
Ezen architektúra alkalmazásával nem csak az adatkezelés egyszerűsödik, de a jövőbeli fejlesztések és bővítések is gördülékenyebben, valamint hatékonyabban valósíthatók meg.

Modellek

A modellek az adatbázisban lévő táblákat reprezentálják, és a kontextus segítségével tudom őket kezelni.

Például a User modell így néz ki:

```
public class User
{
    public Guid Id { get; set; }
    public string Name { get; set; }
    public string Email { get; set; }
    public string? PasswordHash { get; set; }
```

```
    public string? Salt { get; set; }
    public DateTime BirthDate { get; set; }
    public Guid? ImageId { get; set; }
}
```

Programkód 4.1. User modell

A táblákat reprezentáló modelleknek minden tulajdonsága egy oszlop az adatbázisban. A modellekben találhatóak navigációs tulajdonságok is, amelyek segítségével a modellek közötti kapcsolatokat tudom megvalósítani. A táblákat a migrációk segítségével tudom létrehozni, amelyek a modellek alapján generálódnak, majd az update-database parancs kiadásával tudom őket alkalmazni az adatbázison.

Kontextusok

A komplex adatkezelési igényeket kielégítendő, a szoftverarchitektúrámban két különálló adatbázis-kontextust implementáltam. Az első, a "WatchWithFriendsDBContext", főként a felhasználói információkat és a képmegosztás funkciókat kezeli. Ennek az adatállománynak az SQL Server adatbázisban való tárolása biztosítja az adatintegritást és a tranzakcionális biztonságot.

A másik kontextus, a "RoomsDBContext", azonban egészen más jellegű adatokat adminisztrál: itt a szobák és a videók információit kezelem. Erre az adatrétegre egy InMemory adatbázist alkalmazok. Ezzel a megoldással két fontos célt érek el. Egyrészt, az InMemory adatbázis lehetővé teszi a gyors és rugalmas adatmanipulációt, optimális azokhoz a szobákhoz és videókhoz, amelyek volatilis jelleggel bírnak és nem szükségesek hosszútávú tárolásra. Másrészt, a szerver leállításával automatikusan törlődnek ezek az adatok, így nincs szükség manuális karbantartásra ezen a fronton.

A két különböző kontextus párhuzamos működése lehetővé teszi az adatrétegek elkülönítését és a szoftver komponenseinek független skálázhatóságát, miközben az adatintegritás és a rendelkezésre állás is megmarad.

Például a WatchWithFriendsDBContext így néz ki:

```
public class WatchWithFriendsDBContext : DbContext
{
    public WatchWithFriendsDBContext
    (DbContextOptions<WatchWithFriendsDBContext> options)
    : base(options)
    { }

    public DbSet<User> Users { get; set; }
    public DbSet<Image> Images { get; set; }
}
```

Programkód 4.2. WatchWithFriendsDBContext

Repository réteg

A repository-k segítségével tudom kezelni az adatbázist, és a modelleket.
Például a UserRepository így néz ki:

```
public class UserRepository : IUserRepository
```

```
{
    private readonly WatchWithFriendsDBContext _context;

    public UserRepository(WatchWithFriendsDBContext context)
    {
        _context = context;
    }

    public async Task<User> GetUserById(Guid id)
    {
        return await _context.Users
            .Include(u => u.Image)
            .FirstOrDefaultAsync(u => u.Id == id);
    }

    //tobbi metodus
}
```

Programkód 4.3. User Repository

Fontosnak tartottam az async műveletek használatát, mert így a szerver nem blokkolódik, és a kliensek is gyorsabban kapják meg a válaszokat.

Extension metódus

Létrehoztam egy extension metódust, amely segítségével tudom a repository-kat a DI konténerbe regisztrálni, illetve a kontextusokat létrehozni.

```
public static void WatchWithFriendsData(
    this IServiceCollection services,
    DataBaseType dataBaseType)
{
    var IConfigugration = services
                        .BuildServiceProvider()
                        .GetRequiredService<IConfiguration>();
    var connstring = IConfigugration
                    .GetConnectionString("watchwithfriends");
    services.AddDbContext<WatchWithFriendsDBContext>(
    options =>
    {
        options.UseSqlServer(connstring);
    });
    services.AddDbContext<RoomsDBContext>(options =>
    {
        options.UseInMemoryDatabase("rooms")
    });
    services.AddScoped<IUserRepository, UserRepository>();
    //tobbi repository
}
```

Szerver oldali logika és végpontok

A szerver oldali logika az alkalmazás "háttere", itt történik minden, ami az adatok manipulációjával, az üzleti szabályok végrehajtásával és az erőforrások menedzsmentjével kapcsolatos. A végpontok (endpoints) a szerver oldali logika speciális részei, amelyek meghatározzák, hogy a kliens hogyan interakciózik a szerverrel.

Service réteg

A service réteg általában az üzleti logikát tartalmazza egy alkalmazásban. Ebben a rétegben találhatóak meg azok a függvények és metódusok, amelyek nem közvetlenül kapcsolódnak az adatok eléréséhez vagy a felhasználói felülethez, hanem inkább az alkalmazás "logikáját" képezik.

Például a UserService így néz ki:

```
public class UserService : IUserService
{
    private readonly IUserRepository _userRepository;

    public UserService(IUserRepository userRepository)
    {
        _userRepository = userRepository; // DI
    }

    public async Task<User> GetUserById(Guid id)
    {
        return await _userRepository.GetUserById(id);
    }

    //tobbi metodus
}
```

Programkód 4.4. User Service

Controller réteg

A controller réteg a végpontokat kezeli. A végpontok a kliens és a szerver közötti kommunikáció alapját képezik. A végpontok meghatározzák, hogy a kliens hogyan interakciózik a szerverrel. A végpontok a kliens kéréseit fogadják, és válaszokat generálnak.

Például a UserController így néz ki:

```
[ApiController]
[Route("[controller]")]
public class UserController : ControllerBase
{
private readonly IUserService _userService;

public UserController(IUserService userService)
{
```

```
_userService = userService; // DI
}

[HttpGet("{id}")]
public async Task<IActionResult> GetUserById(Guid id)
{
        var user = await _userService.GetUserById(id);
        if (user == null)
        {
                return NotFound();
            }
        return Ok(user);
    }

//tobbi vegpont
}
```

Programkód 4.5. User Controller

Kliens oldali logika

React projekt

A kliens oldali logika a felhasználói felületet és a felhasználói interakciókat kezeli. A kliens oldali kód gyakran aszinkron módon kommunikál a szerverrel, hogy adatokat kérjen vagy küldjön. Ezen adatok küldésére API hívásokat használok. Az API hívásokat a React alkalmazásban egy axios nevű könyvtárral valósítottam meg. Az axios [8] egy HTTP kliens, amely lehetővé teszi a kérések küldését a szervernek, és a válaszok fogadását. Az axios beállításait egy külön fájlban tárolom, így könnyen tudom őket módosítani.

Például a axios beállításai így néznek ki:

```
import axios from 'axios';
import * as AppConfig from './AppConfig';

export const httpClient = axios.create({
    baseURL: AppConfig.GetConfig().apiUrl,
    headers: {
        'Access-Control-Allow-Origin': '*',
        'Content-Type': 'application/json',
        Accept: 'application/json',
    },
});

export const updateAuthorizationHeader = () => {
    const newToken = localStorage.getItem('token');
    httpClient.defaults.headers.common["Authorization"] = newToken ?
     `Bearer ${newToken}` : "";
```

```
};
```

Programkód 4.6. Axios beállítások

Itt látható, hogy a kéréseknek milyen fejlécei lesznek, illetve hogy milyen URL-re küldjük a kéréseket, ezt a httpClient-nek adom meg, majd ezt használom a kérések küldésére. Létrehoztam egy függvényt, amely segítségével tudom a kérésekhez hozzáadni a token-t, amely a felhasználó azonosítására szolgál. A külömböző env-ekhez különböző beállításokat tudok megadni, így a fejlesztés során a fejlesztői környezetben a localhost-ra küldöm a kéréseket, a tesztelés során a teszt szerverre, a production környezetben pedig a production szerverre.

Konfigurációs fájl:

```
import joi from "joi";

interface AppConfig {
    apiUrl: string;
}

export const config: AppConfig = {
    apiUrl: import.meta.env.VITE_WATCH_2_GETHER_BACKEND_ADDRESS ??
    "https://localhost:7165/",
}

const schema = joi.object({
    apiUrl: joi.string().required(),
});

export function GetConfig(): AppConfig {
    const { error, value } = schema.validate(config);
    if (error) {
        throw new Error(`Config validation error: ${error.message}`);
    }
    else {
        return value;
    }
}
```

Programkód 4.7. Konfigurációs fájl

Az env változók validására a joi [10] nevű könyvtárat használom, amely segítségével tudom a változókat ellenőrizni, hogy megfelelnek-e a várt típusnak, illetve, hogy megvannak-e adva.

API services

Az API services rétegben találhatóak meg a különböző API végpontokat kezelő függvények.
Például a UserService így néz ki:

```
import { httpClient } from "../HttpClient";
import { UserDto } from "../models/userDto";
```

```
    export const getUserById = async (id: string):
    Promise<AxiosResponse<UserDto>> => {
    const response = await httpClient.get<UserDto>(`Users/${id}`);
    return response;
};
```

Programkód 4.8. User Service

Minden függvény egy API végpontot hív meg, és a választ visszaadja. A függvényeket a komponensekben használom, ahol a választ feldolgozom, és a megjelenítést végzem el. Csak a Pages mappában található komponensekben használok API hívásokat, a többi komponensben a Props-ok vagy Context API segítségével kapom meg az adatokat.

Router

A Router segítségével tudom a különböző oldalakat megjeleníteni.

Például a Router így néz ki:

```
    <BrowserRouter>
    <Navbar />
    <CommonSrtyles.PageContainer>
      <Routes>
        <Route path="/" element={<HomePage />} />
        <Route path="/rooms" element={<RoomsPage />} />
        <Route path="/profile" element={<ProfilePage />} />
        <Route path="/friends" element={<FriendsPage />} />
        <Route path="/room/:id" element={<RoomPageWithProvider />} />
      </Routes>
    </CommonSrtyles.PageContainer>
  </BrowserRouter>
```

Programkód 4.9. Router

A Routes és Route komponensek a React Router-ben használatosak, és alapvetően az alkalmazás útvonalainak (routes) menedzselésére szolgálnak. Itt egy kis áttekintés a fő komponensekről és tulajdonságokról:

Routes komponense

Ez a komponens az a "konténer", amiben a különböző Route komponenseket helyezzük el. Gyakorlatilag ez kezeli azt, hogy melyik Route komponens aktív egy adott pillanatban.

Route komponens

A Route komponensek felelnek a különböző útvonalakért. Két fő tulajdonságuk van:

path:
Ez határozza meg, hogy milyen URL címnél aktíválódjon a Route. Például, ha a path="/about", akkor a /about URL-nél fog betöltődni az adott Route.

element:
Ez a tulajdonság tartalmazza azt a komponenst, ami meg fog jelenni, amikor az adott Route aktív. Tehát ha a path="/about" és az element=<About />, akkor az About komponens fog megjelenni, amikor az /about URL-re navigálunk.

Komponensek

A React komponensek a UI logikájának egységei, és lehetnek függvény vagy osztály alapúak. Függvényes komponensek egyszerűbbek és könnyebben kezelhetőek, míg az osztály alapúak sokkal több beépített metódust és ciklusfüggvényt kínálnak. A komponenseknek tulajdonságai (props) amivel a komponensek közötti adatáramlást valósítottam meg. A komponenseket alkomponensekre bontottam, hogy a kód minél jobban átlátható legyen, és a komponensek újra felhasználhatóak legyenek. Ezt a styled-components [9] könyvtár segítségével készítettem, amely lehetővé teszi a CSS kódok beágyazását a komponensekbe.

Példa a styled-components használatára:

```
import styled from "styled-components";

export const RoomListItemHeader = styled.div`
    font-size: 30px;
    font-weight: bold;
    margin-left: 15px;
    margin-top: 10px;
`;
```

Programkód 4.10. Styled-components

Így egyszerűen adom meg a komponensek stílusát, és a komponensek újra felhasználhatóak lettek. Illetve ezekből a komponensekből összeállt egy nagyobb komponens.

```
export const RoomListItem = (props: RoomListItemProps): JSX.Element => {
    //tobbi kod
    return (
    <Styles.RoomListItemContainer onClick={join}>
        <Styles.RoomListItemContentContainer>
            <Styles.RoomListItemImage
            src={
            `${AppConfig.GetConfig().apiUrl}Users/${props.creatorId}/image`
                }
                />
            <Styles.RoomListItemHeader>
                {props.name}
            </Styles.RoomListItemHeader>
            <Styles.RoomListItemInfo>
                <div>Creator: {creator?.name}</div>
```

```
                <Styles.RomListItemMembers><Person2Icon />
                    {" " + props.userCount}
                </Styles.RomListItemMembers>
            </Styles.RoomListItemInfo>
        </Styles.RoomListItemContentContainer>
    </Styles.RoomListItemContainer>
    )
};
```

Programkód 4.11. RoomListItem komponens

Context API

Azokat az adatokat amelyeket több komponensben is használok, a Context API-val valósítottam meg. Az alkalmazásban a felhasználói adatokat és a szobákkal kapcsolatos adatokat is a Context API-al valósítottam meg.

Példa a Context API használatára:

```
import { createContext, useState } from "react";

//Context API inicializalasa

interface AuthContextType {
    //adat es setter
}

export const AuthContext = createContext<AuthContextType | null>(null);

//Context API Provider

export const AuthProvider = ({ children }:
 { children: React.ReactNode }) => {
    //adat es setter
    return (
        <AuthContext.Provider
            value={{
                //adat es setter
            }}
        >
            {children}
        </AuthContext.Provider>
    );
};

//Context API hasznalata
const authContext = useContext(AuthContext);
```

Programkód 4.12. Context API

Források

[1] *React*. URL: https://reactjs.org/ (elérés dátuma 2023. 09. 27.).

[2] *Angular*. URL: https://angular.io/ (elérés dátuma 2023. 09. 27.).

[3] *Redux*. URL: https://redux.js.org/ (elérés dátuma 2023. 09. 27.).

[4] *Entity Framework Core*. URL: https://docs.microsoft.com/en-us/ef/core/ (elérés dátuma 2023. 09. 27.).

[5] *RESTful API*. URL: https://www.interserver.net/ (elérés dátuma 2023. 09. 18.).

[6] *SignalR*. URL: https://learn.microsoft.com/ (elérés dátuma 2023. 09. 18.).

[7] *Visual Studio*. URL: https://visualstudio.microsoft.com/ (elérés dátuma 2023. 09. 18.).

[8] *Axios*. URL: https://axios-http.com/ (elérés dátuma 2023. 09. 27.).

[9] *Styled Components*. URL: https://styled-components.com/ (elérés dátuma 2023. 09. 27.).

[10] *Joi*. URL: https://joi.dev/ (elérés dátuma 2023. 09. 27.).

CD Használati útmutató

Ennek a címe lehet például *A mellékelt CD tartalma* vagy *Adathordozó használati útmutató* is.

Ez jellemzően csak egy fél-egy oldalas leírás. Arra szolgál, hogy ha valaki kézhez kapja a szakdolgozathoz tartozó CD-t, akkor tudja, hogy mi hol van rajta. Jellemzően elég csak felsorolni, hogy milyen jegyzékek vannak, és azokban mi található. Az elkészített programok telepítéséhez, futtatásához tartozó instrukciók kerülhetnek ide.

A CD lemezre mindenképpen rá kell tenni

- a dolgozatot egy `dolgozat.pdf` fájl formájában,
- a LaTeX forráskódját a dolgozatnak,
- az elkészített programot, fontosabb futási eredményeket (például ha kép a kimenet),
- egy útmutatót a CD használatához (ami lehet ez a fejezet külön PDF-be vagy MarkDown fájlként kimentve).