

**SAPIENTIA ERDÉLYI MAGYAR TUDOMÁNYEGYETEM
MAROSVÁSÁRHELYI KAR,
INFORMATIKA SZAK**



SAPIENTIA
ERDÉLYI MAGYAR
TUDOMÁNYEGYETEM

Project Raccoon: Felhasználói élmény fejlesztése a
szerződéskötések és tulajdonok eladásának interneten keresztüli
megvalósításához

DIPLOMADOLGOZAT

Témavezető:
Győrfi Ágnes,
Egyetemi tanársegéd
Dr. Kupán Pál,
Egyetemi docens

Végzős hallgató:
Sallai József-Adrian

2023

UNIVERSITATEA SAPIENTIA DIN CLUJ-NAPOCA
FACULTATEA DE ȘTIINȚE TEHNICE ȘI UMANISTE,
SPECIALIZAREA INFORMATICĂ



UNIVERSITATEA SAPIENTIA

Proiect Raccoon: Dezvoltarea experienței de utilizator pentru
finalizarea contractelor și vânzărilor de proprietăți prin
intermediul internetului

LUCRARE DE DIPLOMĂ

Coordonator științific:

Győrfi Ágnes,
Asistent universitar
Dr. Kupán Pál,
Conferențiar universitar

Absolvent:

Sallai József-Adrian

2023

**SAPIENTIA HUNGARIAN UNIVERSITY OF
TRANSYLVANIA
FACULTY OF TECHNICAL AND HUMAN SCIENCES
COMPUTER SCIENCE SPECIALIZATION**



SAPIENTIA
HUNGARIAN UNIVERSITY
OF TRANSYLVANIA

Project Raccoon: Ensuring the User Experience of Contract
Creation and Property Sales Through the Web

BACHELOR THESIS


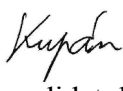

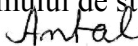

Scientific advisor:

Győrfi Ágnes,
Assistant professor
Dr. Kupán Pál,
Associate professor

Student:

Sallai József-Adrian

2023

UNIVERSITATEA „SAPIENTIA” din CLUJ-NAPOCA Facultatea de Științe Tehnice și Umaniste din Târgu Mureș Programul de studii: Informatică		Viza facultății:
LUCRARE DE DIPLOMĂ		
Coordonator științific: dr. Kupán Pál Îndrumător: Györfi Ágnes	Candidat: Sallai Jozsef-Adrian Anul absolvirii: 2023	
<p>a) Tema lucrării de licență: Proiect Raccoon: Dezvoltarea experienței de utilizator pentru finalizarea contractelor și vânzărilor de proprietăți prin intermediul internetului</p> <p>b) Problemele principale tratate:</p> <ul style="list-style-type: none"> - Studiu bibliografic privind proiectarea interfețelor unei aplicații web folosind React și TailwindCSS. - Studiu bibliografic privind dezvoltarea unui serviciu de chat prin intermediul WebSockets. - Dezvoltarea unui mecanism pentru internaționalizarea și localizarea unei aplicații web. - Dezvoltarea unei interfețe de utilizator intuitivă și robustă. - Testarea aplicației în condiții reale. <p>c) Desene obligatorii:</p> <ul style="list-style-type: none"> - Diagrame de proiectare pentru aplicația software realizată. <p>d) Softuri obligatorii:</p> <ul style="list-style-type: none"> - Aplicație web bazată pe Next.js pentru crearea, gestionarea și semnarea contractelor de vânzare-cumpărare - Serviciu WebSocket pentru comunicarea real-time între părțile aferente unui contract <p>e) Bibliografia recomandată: https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API</p>		
<p>f) Termene obligatorii de consultații: săptămânal, preponderent online</p> <p>g) Locul și durata practicii: Universitatea „Sapientia” din Cluj-Napoca, Facultatea de Științe Tehnice și Umaniste din Târgu Mureș, sala / laboratorul 414 Primit tema la data de: 20.09.2022 Termen de predare: 02.07.2023</p>		
Semnătura Director Departament 	Semnătura coordonatorului  	
Semnătura responsabilului programului de studiu 	Semnătura candidatului 	

Declarație

Subsemnatul/a SALAI JOSEF-ADRIAN, absolvent(ă) al/a specializării
INFORMATICA, promoția 2023 cunoscând
prevederile Legii Educației Naționale 1/2011 și a Codului de etică și deontologie profesională
a Universității Sapiientia cu privire la furt intelectual declar pe propria răspundere că prezenta
lucrare de licență/proiect de diplomă/disertație se bazează pe activitatea personală,
cercetarea/proiectarea este efectuată de mine, informațiile și datele preluate din literatura de
specialitate sunt citate în mod corespunzător.

Localitatea, CORUNCA
Data: 15/06/2023

Absolvent

Semnătura.....

Kivonat

Dolgozatom témája egy olyan webes alkalmazás megtervezése és lefejlesztése, amelynek segítségével az interneten keresztül lehet egyszerűen és kényelmesen adásvételi szerződéseket létrehozni, kitölteni, valamint véglegesíteni. Kitűzött célnak tekintettem azt, hogy az alkalmazást bárki könnyedén tudja használni és bárhol is el tudja érni, valamint egy olyan felhasználói élményt nyújtson, amely egyensúlyba helyezi a kényelmet és funkcionalitást.

Az évek során egyértelművé vált, hogy az online környezet által nyújtott szolgáltatások egyre nagyobb teret nyernek a mindennapokban. Manapság a legtöbb tevékenységünket, legyen az vásárlás, szórakozás, vagy akár munka, az interneten keresztül végezzük el, otthonunk nyugalmából.

Az egyik legfontosabb ágazata az online szolgáltatásoknak az állami ügyintézés. Ha az állampolgárok számára lehetővé tesszük, hogy ezeket az ügyeket az interneten keresztül is el tudják intézni, akkor jelentősen megkönnyíthetjük az emberek életét. Egy online állami rendszernek köszönhetően nem szükséges a pontos időpontra való személyes jelenlét, valamint a hosszú sorokban állás csak azért, hogy néhány papírt kitöltsünk, aláírjunk és leadjunk.

Jelen dolgozatomban a fent említett felhasználási esetek egyikére, az adásvételi szerződésekkel kapcsolatos ügyintézésekre fejlesztettem egy szoftvert. Arra a kérdésre kerestem a választ, hogy milyen technológiákat érdemes használni, hogy könnyen lehessen egy ilyen jellegű alkalmazást lefejlesztetni, valamint milyen design keretrendszert és praktikákat tudnék alkalmazni, hogy a felhasználói élményt maximalizálni lehessen. Igyekeztem arra is választ keresni, hogy milyen módon lehetne olyan webes standardokat is alkalmazni, mint például a WebSockets.

A dolgozatban bemutatom a fejlesztés során használt technológiákat, a szoftver végfelhasználói részéhez tartozó tervezési és implementációs folyamatokat, valamint a különböző felhasználói eseteket, amelyeket a szoftvernek támogatnia kell.

Végül a fejlesztés és a tesztelés során szerzett tapasztalatokat összefoglalom, valamint a továbbfejlesztési lehetőségeket is bemutatom.

Kulcsszavak: webes alkalmazás, adásvételi szerződés, felhasználói élmény

Rezumat

Tema tezei mele este dezvoltarea unei aplicații web care poate fi folosită pentru a crea, completa și finaliza contracte de vânzare-cumpărare în mod ușor și convenabil prin intermediul internetului. Scopul dorit de mine a fost proiectarea aplicației pentru a fi ușor de utilizat, accesibilă de oriunde, și pentru a oferi o experiență de utilizator care balansează confortul și funcționalitatea.

De-a lungul anilor, a devenit evident că serviciile oferite de lumea online joacă un rol din ce în ce mai important în viața noastră de zi cu zi. În prezent, multe dintre activitățile noastre, fie că este vorba de cumpărături, divertisment sau chiar muncă, sunt realizate prin intermediul internetului, din confortul propriei noastre locuințe.

Unul dintre cele mai importante domenii ale serviciilor online este administrația publică. Dacă le oferim cetățenilor posibilitatea de a-și rezolva problemele administrative online, le putem ușura sarcina în mod semnificativ. Mulțumită unui sistem online de stat, cetățenii nu trebuie să fie prezenți la o anumită oră și să stea la cozi lungi doar pentru a completa, semna și depune câteva documente.

În această teză, am dezvoltat un software pentru una dintre cazurile de utilizare menționate mai sus, și anume contractele de vânzare-cumpărare. Am încercat să găsesc răspunsuri la următoarele întrebări: ce tehnologii ar trebui folosite pentru a facilita dezvoltarea unei astfel de aplicații și ce framework-uri și practici de design pot fi folosite pentru a maximiza experiența utilizatorului. Am încercat, de asemenea, să aflu cum pot fi folosite unele standarde web moderne, cum ar fi WebSockets, într-o astfel de aplicație. Voi prezenta tehnologiile utilizate în timpul dezvoltării, procesul de proiectare și implementare a părții de utilizator al software-ului și diversele cazuri de utilizare pe care software-ul trebuie să le susțină.

În cele din urmă, rezum experiențele dobândite în timpul dezvoltării și testării, și prezint posibilitățile de dezvoltare ulterioară.

Cuvinte de cheie: aplicație web, contract de vânzare-cumpărare, experiență de utilizator

Abstract

The aim of my thesis is to design and develop a web application that can be used to create, fill out, and finalize sales contracts easily and conveniently over the Internet. My goal was to make the application easy to use and accessible from anywhere, and to provide a user experience that balances comfort and functionality.

Throughout the years, it has become evident that the services offered by the online world play an increasingly important role in our everyday lives. Nowadays many of our activities, whether it is shopping, entertainment, or even work, are done over the Internet, from the comfort of our homes.

One of the most important branches of online services is public administration. If we allow citizens to handle these matters through the Internet, we can significantly ease their burden. With an online state system, there is no need for citizens to be present at a specific time, nor to stand in long queues just to fill out, sign, and submit a few papers.

In this thesis, I developed a software for one of the use cases mentioned above, namely sales contracts. I tried to find answers to the following questions: what technologies should be used to make it easy to develop such an application, and what design frameworks and practices can be used to maximize the user experience. I also tried to find out how modern web standards, such as WebSockets, can be useful in such an application. I present the technologies used during development, the design and implementation process of the end-user part of the software, and the various use cases that the software must support.

Finally, I summarize the experiences gained during development and testing, and present the possibilities for further development.

Keywords: web application, sales contract, user experience

Tartalomjegyzék

1. Bevezető	11
1.1. Célkitűzés	12
2. Elméleti megalapozás	14
2.1. Piacelemzés	14
2.1.1. Dropbox Sign (HelloSign)	14
2.1.3. PandaDoc	15
2.1.5. Singaporean Network Trade Platform (NTP)	16
2.2. Szoftverek	16
2.2.1. Next.js és React	16
2.2.2. TypeORM	17
2.2.3. TailwindCSS	18
2.2.4. WebSocket és Socket.IO	18
3. Szoftverkövetelmények	20
3.1. Felhasználói követelmények	20
3.2. Rendszer követelmények	22
3.3. Funkcionális követelmények	23
3.4. Nem-funkcionális követelmények	23
4. A szoftver tervezése	25
4.1. Fejlesztési eszközök	25
4.2. Projektszerkezet	26
4.3. UI komponensek	29
4.4. Adatmodellek	32
4.5. Diagramok	36
4.5.1. Sequence diagram	36
4.5.2. Activity diagram	37
4.5.3. Osztálydiagram	38
4.6. WebSocket alapú chat működési elve	39
5. A szoftver működésének részletes bemutatása	41
5.1. Landing page és egyéb tartalmi oldalak	41
5.2. Felhasználók hitelesítése és adataik kezelése	41
5.3. Tulajdonok és szerződések kezelése	42
5.4. Szerződés felei közti üzenetküldés	44
5.5. Felhasználói beállítások	45

5.6. Felhasználók által feltöltött állományok eltárolása	46
5.7. Többsnyelvűsítés (I18N)	46
5.8. Akadálymentesítés és mobil optimalizáció	48
6. A rendszer gyakorlati működése	50
Összefoglaló	57
6.1. Következtetések	57
6.2. Továbbfejlesztési lehetőségek	57
Ábrák jegyzéke	59
Táblázatok jegyzéke	60
Irodalomjegyzék	62

1. fejezet

Bevezető

Nem kétséges az, hogy a digitális modernizálódás világában élünk. Az internetezők száma folyamatosan nő, eszközeink egyre több dologra képesek, valamint a szoftverek is egyre összetettebbek és egyre több lehetőséget kínálnak. Folyamatos innovációban részesülnek a mindennapi tevékenységeink is: nagy fellendülésnek indult például az online vásárlás, számlafizetés, szórakozás, kommunikáció, valamint bizonyos körülmények között az online munkavégzés is.

Egy olyan ágazat, ami még nem kapott megfelelő figyelmet, főleg a kisebb európai országokban, az a modernizált állami ügyintézés. Sajnos még mindig sok olyan dolog van, amit csak személyesen lehet elintézni, ami együtt jár a pontos időpontokra való személyes megjelenés, az olykor órákig tartó várakozás, valamint a sokszor feleslegesnek érezhető bürokrácia okozta bosszúságokkal és stresszel.

Egyes országok, mint például Észtország, nagyon komolyan veszik az állami ügyintézés digitalizálását. Észtország egyike az egyetlen olyan európai országoknak, ahol majdnem minden ügyet egy jól kidolgozott internetes portálon keresztül el lehet intézni. [3] Mindezek mellett egyike az egyetlen olyan országoknak az egész világban, ahol bárki, bármilyen más országból tud egy „elektronikus állampolgárságot” igényelni, amelynek segítségével az illető bárhol, bármikor, az internetes portálon és egy fizikai kulcs használatával elérheti és kezelheti adatait, tulajdonait, cégeket alapíthat és kezelhet, adóbevallást tehet, valamint számos más állami ügyet is el tud intézni. [2]

Az internet egyik legfontosabb szellemi összetevője a kommunikáció és az együttműködés. Itt természetesen az első dolog, ami eszünkbe jut, az az interneten keresztül csevegés, levelezés, valamint a közösségi média. Azonban kommunikáció alatt gondolhatunk még olyan dolgokra is, mint például a kollaboratív munkavégzés, ahol több ember együtt dolgozik egy adott feladaton, valamint az információk megosztása, amelynek keretén belül az emberek kisebb vagy nagyobb fontosságú információkat és dokumentumokat oszthatnak meg egymással.

Az egyik olyan állami ügy, ami még nem kapott kellő figyelmet a digitalizálást illetően, az az adásvételi szerződések lebonyolítása. Sokszor hetekig tartó folyamat az, hogy két fél megossza egymás között a szükséges dokumentumokat, számos papírmunkát elvégezzen, ügyvédet és tanúkat fogadjon, ráadásul mindezt személyesen. Ezt a folyamatot tovább tudja befolyásolni a két fél közötti elérhetőség, valamint a távolság.

Ha picit jobban belegondolunk, akkor az adásvételi folyamat összehangolja azokat az elemeket, amik az internet lényegét alkotják, más szóval, az adásvételi szerződések

lebonyolítása egy olyan folyamat, amely teljes mértékben digitalizálható és amely élvezni tudja az internet által nyújtotta lehetőségeket. A két fél, valamint a szerződéshez szükséges tanúk és ügyvédek könnyedén tudnának kommunikálni egymással és meg tudnák osztani egymással a szükséges iratokat és adatokat. Éppen ezért egyértelmű az, hogy egy olyan online platformra van szükség, ami egységesíti a teljes folyamatot, így mindent egy helyen, könnyedén, otthonunk saját kényelméből el tudunk intézni.

A következő fejezetekben bemutatom a Project Raccoon nevű szoftvert, amely egy ilyen jellegű online platform szerepét akarja betölteni. Részletes bemutatásra kerülnek a szoftver fejlesztéséhez használt technológiák, tervezési döntések, valamint a szoftver működésének részletes leírása és bemutatása.

1.1. Célkitűzés

A szoftver fejlesztése során a következő célokat tűztem ki magam elé:

- Tanulmányozni a Next.js keretrendszer által nyújtott lehetőségeket egy full-stack webes platform fejlesztéséhez.
- Tanulmányozni a TailwindCSS könyvtárat, valamint olyan komponenseket tervezni és implementálni, amelyek segítségével biztosítani lehet a felhasználói élményt.
- Kutatást végezni az adásvételi szerződések lebonyolításával kapcsolatos legfontosabb felhasználói esetekről, valamint ezeket támogatni a szoftverben.
- Egy olyan landing page-t készíteni, mely bemutatja és összefoglalja a platform által nyújtott szolgáltatásokat, valamint a felhasználókat arra biztatja, hogy regisztráljanak.
- Egy hitelesítési rendszert kifejleszteni, melyen keresztül a felhasználók regisztrálhatnak, be tudnak jelentkezni, valamint kezelni tudják a személyes adataikat.
- Egy felületet megtervezni és kifejleszteni, amely segítségével a felhasználók kezelhetik tulajdonaikat, valamint az ahhoz tartozó adásvételi szerződéseket.
- Egy WebSocket alapú chatrendszert kifejleszteni, amelyen keresztül egy adott szerződés felei kommunikálni tudnak egymással.
- A szoftvernek támogatnia kell a többnyelvűsítést.
- Kisegítő lehetőségeket fejleszteni a felhasználók számára, mint például a billentyűzet használatának könnyítése, valamint különböző megjelenési témák biztosítása.
- A szoftver felületét úgy megtervezni, hogy mobiltelefonokon is kényelmesen használható legyen.
- Felhasználói dokumentációt készíteni.
- Konfigurálható állománytárolási stratégiákat kivitelezni: lokális fájlrendszer, Amazon S3, Firebase Storage.

- Konfigurálható SQL-alapú adatbázisokat támogatni.
- A projektet publikussá tenni és tesztelni egy serverless környezetben.
- Valós felhasználási eseteket szimulálni, tesztelni, valamint mérni a szoftver teljesítményét.

2. fejezet

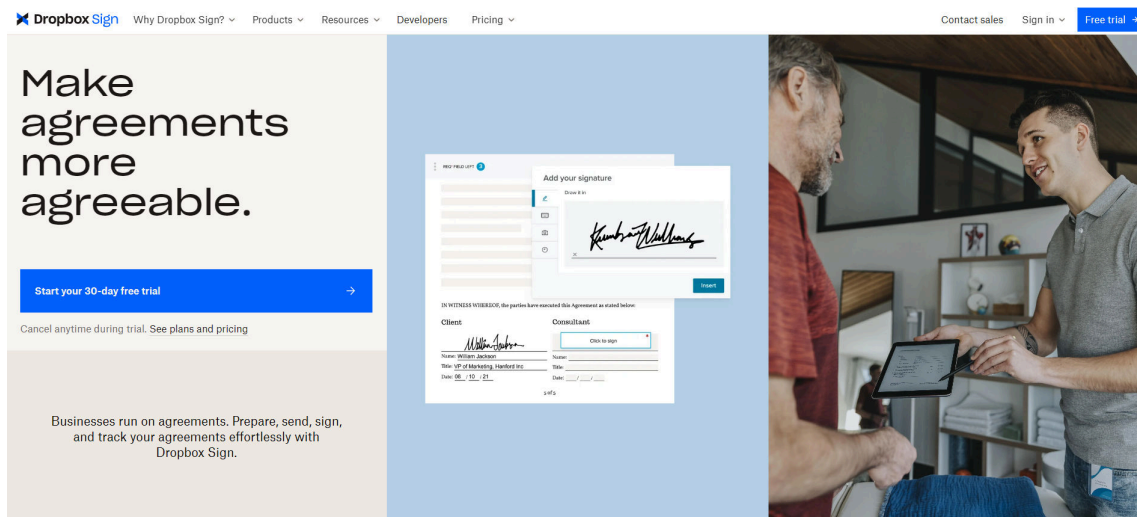
Elméleti megalapozás

2.1. Piacelemzés

A Project Raccoon ötletének megszületésekor utánajártam, hogy milyen hasonló megoldások léteznek a piacon. Olyan szoftvereket kerestem, amelyek a projekt legtöbb funkcióját támogatják, viszont legtöbbször csak olyan megoldásokat sikerült találnom, amik csak egy részét fedték le a Project Raccoon céljának.

Egyes platformok például csak az ingatlanok vagy más tulajdonok eladására szaksodtak, szerződéskötési funkcionalitással pedig nem rendelkeztek. Mások ugyan támogaták a szerződések kitöltését és aláírását, viszont nem volt szoros kapcsolat az eladó, a vevő, illetve az eladott tulajdon között.

2.1.1. Dropbox Sign (HelloSign)



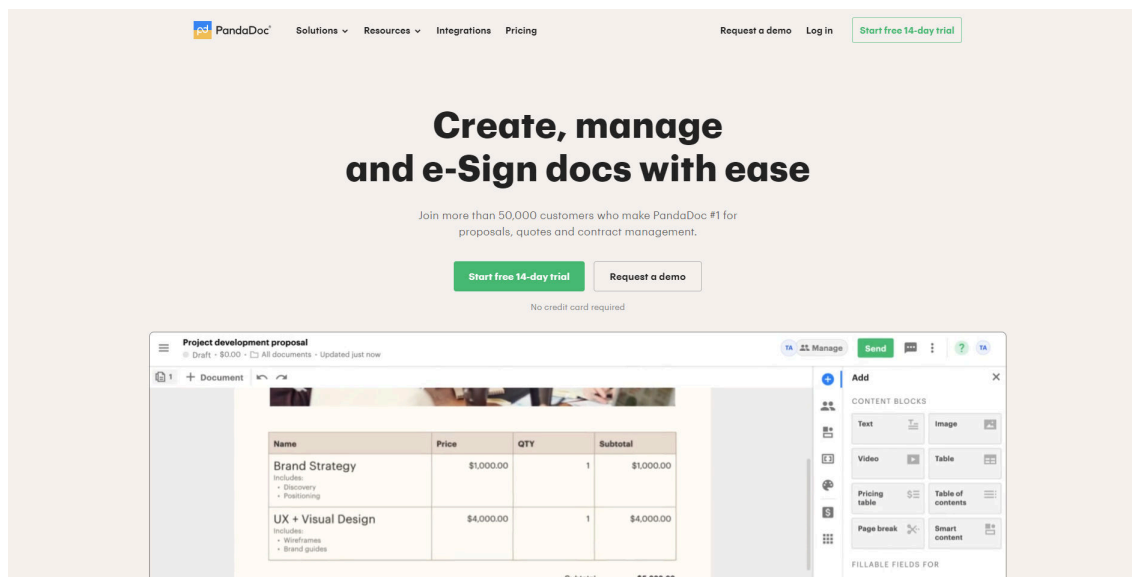
2.1. ábra. Dropbox Sign (HelloSign) főoldala

A Dropbox Sign egy olyan platform, ami szerződések online kitöltését és digitális aláírását biztosítja. Erőssége, hogy lehetőséget nyújt arra, hogy egy szerződéshez több fél tartozzon és mindegyik fél külön-külön alá tudja írni azt.

A Dropbox Sign egy másik jellegzetessége, hogy lehetőséget nyújt a felhasználók számára, hogy sablonokat hozzanak létre, amelyeket később újra fel tudnak majd használni. A sablonokat egy intuitív felületen lehet létrehozni, valamint lehetőség van arra is, hogy olyan mezőket definiáljunk, amik automatikusan ki lesznek bizonyos adatokkal töltve. [12]

Ez a megoldás nagyon hasznos a dokumentumok kitöltését és aláírását illetően, viszont a Project Raccoon-hoz viszonyítva nincs szoros kapcsolatban az adásvételi folyamatok lebonyolítását illető funkciókkal. Ugyanakkor, a Dropbox Sign használata azt is jelenti, hogy a felhasználók feladata lesz az, hogy a sablonokat előkészítsék, így ezzel is több munkát ruház a felhasználókra. A számítógépes ismeretekkel kevésbé rendelkező felhasználók számára ez a platform nem biztos, hogy nagyon optimális, főleg az eladók szemszögéből.

2.1.3. PandaDoc



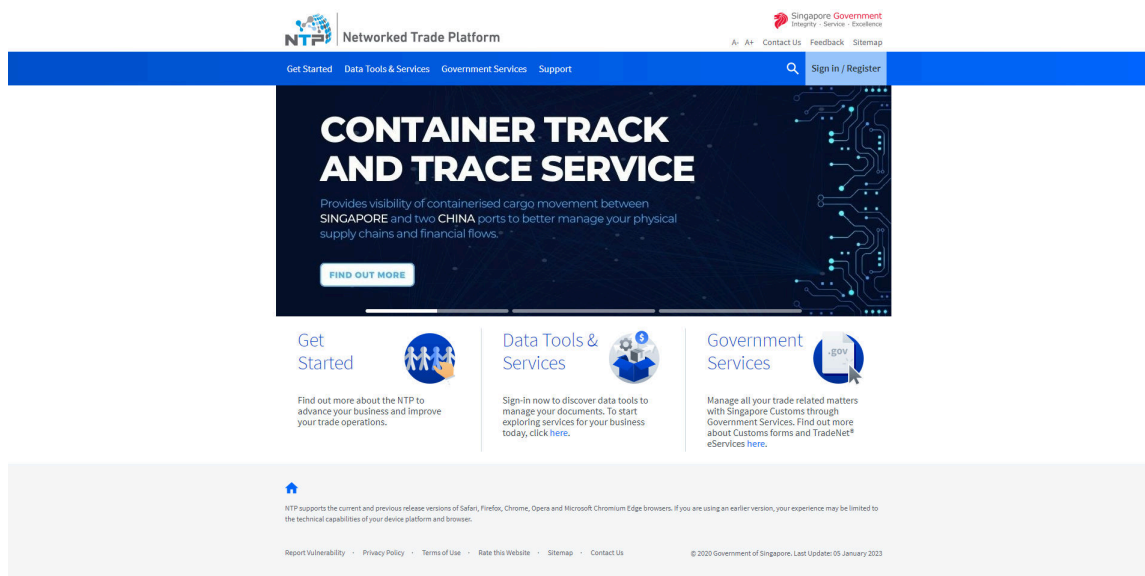
2.2. ábra. PandaDoc főoldala

A PandaDoc nagy mértékben hasonlít az imént említett Dropbox Sign-ra, viszont rendelkezik előre definiált sablonokkal is, amik közé tartozik egy adásvételi szerződés sablon is. [11] Ennek előnye, hogy a szerződéssablonok létrehozásának feladata nem a felhasználókra hárul, tehát egyből neki lehet fogni a szerződések kitöltésének és aláírásának.

A PandaDoc inkább az angol nyelvet beszélők számára készült, mivel maga a felület és a szoftver által felajánlott sablonok is angol nyelvűek. A Project Raccoon ezzel szemben inkább a magyar nyelvű közösség felé irányul. Mindezek mellett a PandaDoc saját sablonokkal rendelkezik, amik nem feltétlenül felelnek meg a jogszabályok által elvárt formátumoknak. Ezekből az okokból kifolyólag a magyar nyelvű közösség számára megint az eladókra hárul a feladat, hogy megfelelő sablonokat hozzanak létre.

A Project Raccoon-nal szemben a PandaDoc nem rendelkezik a tulajdonok kezelését szolgáló funkciókkal sem, mivel ez a szoftver nagyon általános funkciókészletet nyújt a dokumentumok kitöltését és aláírását illetően.

2.1.5. Singaporean Network Trade Platform (NTP)



2.3. ábra. Singaporean Network Trade Platform főoldala

Kormányzati szinten is léteznek hasonló megoldások más országokban, mint például a szingapúri Network Trade Platform (a továbbiakban NTP). Az NTP olyan megoldásokat nyújt, amik bizonyos tulajdonok eladását elősegítik, beleértve a szerződések létrehozását, kitöltését és digitális aláírását is. [10]

Az NTP egyik legnagyobb előnye, hogy hivatalos szerződéssablonokat biztosít, így a felhasználók mindig biztosak lehetnek, hogy a szerződéseik megfelelnek a formai követelményeknek.

Az NTP egyik hátránya viszont, hogy leginkább céges szinten használható, így a magánszemélyek számára nem biztos, hogy egy optimális megoldás. Ezen kívül az NTP csak a szingapúri lakosok számára elérhető.

Ezekből a példákból egyértelmű, hogy van elég érdeklődés olyan platformok iránt, amik a szerződések digitális kitöltését és aláírását egyszerűsítik, valamint a tulajdonok eladását könnyítik meg. A Project Raccoon tehát azzal a céltudattal jött létre, hogy egységesítse a tulajdonok kezelését, a szerződések lebonyolítását, valamint a felek közötti egyszerű kollaborációt.

2.2. Szoftverek

A Project Raccoon több komponensből tevődik össze, mindegyik komponenshez pedig különböző nyílt forráskódú könyvtárakat és keretrendszereket használtam fel.

2.2.1. Next.js és React

A Next.js egy, a Vercel cég által fejlesztett keretrendszer, amelynek segítségével könnyen lehet full-stack alkalmazásokat fejleszteni a React frontend könyvtár segítségével. A Next.js jellegzetességei közé tartozik az, hogy a kliensoldali alkalmazás mellett tudunk

úgynevezett API route-okat is definiálni, amik a szerveroldalon futnak és teljes mértékben kompatibilisek a serverless architektúrával. Ennek köszönhetően ugyanabban a kódbázisban, ugyanazzal a keretrendszerrel tudjuk lefejleszteni és összehangolni a backend-et és a frontend-et egyaránt.

A Next.js keretrendszer alkalmazza a hibrid renderelési technikát, amely azt jelenti, hogy a felhasználástól függően többféle kimenetet tud generálni minden egyes oldalon. [6] Egy fájlrendszer-alapú routing mechanizmust használ, aminek révén minden egyes route-hoz tartozik egy állomány, amelynek a neve megegyezik a route nevével. [7] Komplexebb route-okat alkönyvtárak létrehozásával tudunk definiálni. Minden route át lesz fordítva a következő renderelési típusok egyikére:

- **Static Site Generation (SSG):** a route egy statikus HTML oldal lesz, amelyet a Next.js kompilálás során kigenerál. Ezeket az oldalakat olyankor használjuk, amikor a tartalom nem változik gyakran.
- **Server-side Rendering (SSR):** a route egy dinamikus HTML oldal lesz, amelyet a Next.js minden egyes kérésre újra generál a szerveroldalon. Miután a szerver elküldte a böngészőnek a tartalmat, a React átveszi az irányítást és a kliensoldalon hidrálja a kimenetet és mountolja az általunk definiált komponenseket. Ezeket az oldalakat olyankor használjuk, amikor a tartalom gyakran változik.
- **Client-Side Rendering (CSR):** a route egy minimális HTML oldal lesz, ami nem tartalmaz semmit, hanem JavaScript segítségével tölti be a teljes oldalt a kliensoldalon. Ezeket általában akkor használjuk, amikor nem tudunk SSR-t használni, viszont rendelkezünk dinamikus tartalommal.
- **API Routes:** olyan lambda függvények, amik a szerveroldalon, Node.js környezetben futnak. Ezeket a route-okat olyankor használjuk, amikor backend logikát szeretnénk implementálni, mint például egy REST API alapú rendszert. Mivelhogy a Next.js serverless-first, ezért ezeket a route-okat könnyen tudjuk serverless platformokon használni. Ennek egyik hátulütője, hogy bizonyos körülmények között nem lehet megtartani a különböző route-ok és lekérések között az állapotot (state), mivel minden egyes kérésnél újra inicializálódik a szerveroldali kód.

A Next.js keretrendszer a React könyvtárra épül, amely egy komponens alapú frontend könyvtár. A React jellegzetessége, hogy JavaScript-en belül tudjuk definiálni a felhasználói felületet a JSX szabványnak köszönhetően. A JSX nagy mértékben hasonlít a HTML-hez, viszont lehetőségünk van a React kódban definiált JavaScript függvényeinkhez és state változóinkhoz hozzáférni. [13] React-ben két módon tudunk komponenseket definiálni: osztályok vagy függvények segítségével. Manapság a függvényes megközelítés a felkapottabb, mivel egyszerűbb velük dolgozni és a React Hooks API nagyon sok lehetőséget biztosít számunkra.

2.2.2. TypeORM

Egy ORM (Object-Relational Mapping) segítségével tudunk adatbázismodelleket definiálni, migrációkat készíteni és futtatni, valamint végrehajtani a CRUD műveleteket egy relációs adatbázis (pl. MySQL) felett. Az ORM-eknek több előnye is van, mint például,

hogy nem kell SQL-t írunk, kevesebb eséllyel tudunk SQL injection-t elkövetni, valamint a felhasznált programozási nyelvtől függően a kódunk olvashatóbbá válik és könnyebben tudunk vele dolgozni.

A projekt keretén belül a TypeORM nevű könyvtárat használtam, mivelhogy a projekt a TypeScript programozási nyelvben van írva, a TypeORM pedig pont abból a célból jött létre, hogy jó összhangban működjön a TypeScript-tel. A TypeORM támogatja a dekorátorokat, amelyek segítségével tudjuk annotálni a modelljeinket jellemző osztályokat, valamint azok mezőit. [15] Ennek köszönhetően a modelljeink, valamint az azok felett elvégezhető CRUD műveletek teljesen típusbiztosak.

A TypeORM képes migrációkat generálni, illetve futtatni is. A migrációk olyan műveletsorok, amik egy vagy több adatbázismodell sémáját képesek módosítani. Ezeket nyomon tudjuk követni, valamint vissza tudjuk vonni, így könnyen tudjuk kezelni az adatbázis változásait.

2.2.3. TailwindCSS

A TailwindCSS, más CSS keretrendszerekkel ellentétben nem előre stilizált komponenseket biztosít, hanem alacsony szintű, utility-first CSS osztályokat, amelyeket felhasználhatjuk ahhoz hogy a projektünk saját design nyelvéhez illeszkedő komponenseket hozzunk létre. A TailwindCSS egyik legnagyobb előnye, hogy egységesen tudjuk konfigurálni a projektünkben használt színeket, betűtípusokat, stílusokat, stb. egyetlen konfigurációs fájlban.

A TailwindCSS egyik kritikája, hogy nagyon sok CSS osztályt kell használni egy adott elem stilizálásához, ezáltal a HTML kódunk nagyon nehezen olvashatóvá tud válni. Ez viszont részben jó dolog is, mivel közvetlen módon ösztönzi a fejlesztőket, hogy újrahasznosítható komponenseket hozzanak létre, ezáltal még egységesebbé téve a felhasználói felületet. [4]

Egy másik ok amiért a TailwindCSS használata mellett döntöttem a keretrendszer által használt JIT (Just-In-Time) compiler, ami mindig csak azokat a CSS osztályokat fordítja le, amiket használunk. Ezáltal a CSS állományaink sokkal kisebbek lesznek, mint más CSS keretrendszerek esetében, ahol a teljes keretrendszer stílusait le kell fordítani, függetlenül attól, hogy használjuk-e azokat.

A TailwindCSS továbbá rendelkezik egy nagyon hasznos Visual Studio Code kiegészítővel, amely segítségével a CSS osztályokhoz IntelliSense-t kapunk, ami nagyon megkönnyíti a fejlesztést. Mindez ráadásul teljesen kompatibilis a saját konfigurációnkkal.

2.2.4. WebSocket és Socket.IO

A WebSocket egy olyan web standard technológia, melynek segítségével full-duplex kommunikációra vagyunk képesek egy kliens és egy szerver között. [1] Ez önmagában azt jelenti, hogy a kliens és a szerver egymástól függetlenül tud üzeneteket küldeni egymásnak. Más szavakkal, a kezdetleges TLS handshake után létrejön egy kapcsolat, ami fennáll addig, amíg valamelyik fél meg nem szakítja azt, ez alatt pedig a kliens küldhet üzeneteket a szervernek, a szerver pedig a kliensnek. Ez a technológia sokkal robusztusabb realtime kommunikációra, mint egy polling-on alapuló megoldás, ahol a kliensnek folyamatosan meg kell kérdeznie a szervertől, hogy van-e új üzenet.

A Socket.IO egy nagyon hatékony és production-ready WebSocket implementáció, aminek segítségével könnyen tudunk implementálni WebSocket-alapú valós idejű alkalmazásokat. [14] A Socket.IO két részből áll: egy WebSocket szerverből és egy JavaScript kliensből. A Socket.IO szerver Node.js környezetben fut, míg a kliensoldali kódunkat a böngésző WebSockets API-ját használja fel.

A Socket.IO egyik legnagyobb előnye, hogy nem kell mi magunk leimplementálnunk a két komponens működését, valamint az olyan funkciókat, mint például az automatikus újrapcsolódás, vagy a kliensoldali és szerveroldali események szinkronizálása. A Socket.IO továbbá támogatja a szobákat, amiket nehezebb lenne manuálisan implementálni. A szobák segítségével tudjuk a klienseket csoportokba rendezni, így csak azok a kliensek kapják meg az adott eseményeket, akik egy adott szobában vannak.

3. fejezet

Szoftverkövetelmények

3.1. Felhasználói követelmények

3.1. táblázat. Felhasználói követelmények

Onboarding	<ol style="list-style-type: none">1. Az alkalmazás, kezdőlapja, avagy landing page-je, röviden össze kell foglalja azt, mire is szolgál a szoftver, illetve olyan szövegeket és grafikákat kell tartalmazzon, amik ösztönzik a látogatót arra, hogy létrehozzon egy fiókot.2. A regisztrációs űrlap csak azokat az adatokat tartalmazza, amik a hitelesítéshez szükségesek, mivel az emberek általában nem szeretnek sok időt eltölteni a regisztrációval.3. A felhasználók személyes adatait mihamarabb eltároljuk, mivel olyan szoftverről van szó, ahol hivatalos dokumentumokkal dolgozunk. Ezért minden belépésnél a szoftver leellenőrzi, hogy a személyes adatok megvannak-e adva, ha pedig nem, akkor a felhasználót át kell irányítani egy erre a célra kialakított oldalra.
-------------------	---

Tulajdonok	<ol style="list-style-type: none"> 1. Meg kell adni a felhasználónak a lehetőséget, hogy meghatározza a tulajdon típusát (például: ingatlan, autó, stb.), a tulajdont azonosító adatokat (például egy autó esetében az autó márkája, típusa, színe, technikai adatai, ingatlan esetében pedig az ingatlan elérési címe, stb.), valamint egyéb, egy szerződés megkötéséhez szükséges információkat (például a tulajdon eladási ára, a tulajdon állapota, stb.) 2. A tulajdon adatait úgy kell eltárolni a szoftveren belül, hogy csak azok férjenek hozzá, akik részt vesznek az ügyintézésben (a tulajdonos és egy szerződés létrehozása után a vevő, illetve a szerződéshez hozzáadott harmadik felek).
Szerződések	<ol style="list-style-type: none"> 1. Szerződések létrehozásához a felhasználó meg kell tudja határozni az eladandó tulajdont azáltal, hogy egy listából kiválasztja azt. 2. Az eladó felhasználó meg kell tudja határozni a szerződés másik résztvevőjét, a vevő email címének megadásával. 3. A szerződés létrehozása után az eladót át kell irányítani egy olyan oldalra, ahol megadhatja a szerződéshez szükséges további adatokat. 4. A felhasználóknak meg kell adni a lehetőséget, hogy külön-külön meg tudjanak hívni további résztvevőket, például ügyvédet vagy tanúkat. 5. A szerződés véglegesítése előtt a felhasználók meg kell adni a lehetőséget, hogy áttekinthessék és szerkesszék a szerződés adatait. 6. Miután egy szerződés véglegesítve lett, az adatok módosítását meg kell akadályozni.

Chat	<ol style="list-style-type: none"> 1. Egy szerződésen belül meg kell adni a résztvevőknek a lehetőséget, hogy közvetlenül az alkalmazáson keresztül tudjanak valós időben kommunikálni. 2. Egy ilyen chatszobát minden egyes szerződéshez külön-külön létre kell hozni. 3. Megkülönböztetést kell tenni a szerződés különböző felei között. 4. Új üzenetek érkezéséről hangalapú értesítést kell küldeni az aktív résztvevőknek.
------	--

3.2. Rendszer követelmények

A Project Raccoon egy webes alkalmazás, ami azt jelenti, hogy ugyanúgy használható Windows-on és UNIX-alapú operációs rendszereken is (pl. Linux, macOS, stb.), valamint mobileszközökön (Android, iOS, stb.). Az egyetlen megkötés az egy modern böngésző használata, amely támogatja a HTML5 és CSS3 szabványokat, valamint az ECMAScript 2015-ös szabványt. Az alkalmazás buildelési folyamatán belül több polyfill is használatra kerül, melynek segítségével bizonyos funkciók támogatása régebbi böngészőkben is megoldható.

Böngészőverziókra nincs különösebb megkötés, azonban a következő minimális verziókat ajánlott használni [5]:

- Google Chrome 64+
- Mozilla Firefox 67+
- Microsoft Edge 79+
- Apple Safari 12+

Az alkalmazás nem támogatja az Internet Explorer böngészőt a böngésző elavult állapotának köszönhetően.

A szoftver használatához természetesen szükséges egy aktív internetkapcsolat, valamint a JavaScript engedélyezése az alkalmazás domainjén. A chat funkció használatához a böngészőnek támogatnia kell a WebSocket szabványt is.

Az alkalmazás alkalmazza a code-splitting (bizonyos kódrészletek kisebb darabokra, állományokra való lebontása), valamint a lazy-loading (kisebb kódrészletek csak akkor töltődnek be, ha azokra szükség van) technikákat, így egy nagyobb JavaScript állomány betöltése helyett kisebb állományokat tölt be folyamatosan, az alkalmazást használva. Ennek köszönhetően az alkalmazás betöltése és működése gyorsabb, még a gyenge internetkapcsolattal rendelkező eszközökön is.

3.3. Funkcionális követelmények

A szoftver kezdőlapjának a legfontosabb szerepe az, hogy röviden tudassa az új látogatókkal, hogy mi a szoftver célja, illetve hogyan zajlik le egy adásvételi szerződés véglegesítése. Marketing szempontból fontos, hogy a kezdőlap olyan szlogeneket, grafikákat, illetve szövegeket tartalmazzon, amik felkeltik az érdeklődést és ösztönzik a látogatót arra, hogy létrehozzon egy fiókot. A szerződéskezelési folyamatot minél rövidebben és egyszerűbben kell összefoglalni, nem kell pontos részletekbe belemenni.

A hitelesítési oldalak, avagy a belépési és regisztrációs oldalak, könnyen elérhetőek kell legyenek a landing page fejlécéből, illetve bizonyos call-to-action gombokon keresztül. Sikeres hitelesítés után a felhasználót át kell irányítani az ügyfélkapu oldalára, ahol elkezdhetik használni a szoftvert.

Az ügyfélkapu egyik legfontosabb összetevője a bal oldali menüsáv. Ez a menü egy „kapu” szerepét tölti be, mivel innen el lehet érni a szoftver által kínált összes funkciót. Mobileszközökön nem praktikus állandó jelleggel megjeleníteni a menüt, még akkor is ha az az oldal elején van, így egy lebegő, hamburger ikont tartalmazó gombot kell megjeleníteni a képernyő egyik sarkán, ami szükség esetén előhossa az oldalsávot.

3.4. Nem-funkcionális követelmények

A nem-funkcionális követelményeket több kisebb csoportba lehet szervezni:

3.2. táblázat. Termék követelmények

felhasználhatóság	<ol style="list-style-type: none">1. A legtöbb felhasználó végig kell tudja hajtani bármelyik feladatot anélkül, hogy segítségre szorulna.2. Abban az esetben, ha egy felhasználó mégis segítségre szorul, akkor hozzá kell tudjon férni az alkalmazás által nyújtott dokumentációhoz.3. A felhasználói felület eléggé intuitív kell legyen ahhoz, hogy egy, a számítógépekkel kevésbé jártas felhasználó is könnyedén tudja használni kellemetlenségek nélkül.4. A felhasználói felület nem tartalmazhat olyan gombokat, ikonokat, vagy szövegeket, amik félrevezetőek vagy nem eléggé tiszták.
teljesítmény	<ol style="list-style-type: none">1. Az alkalmazás minden felhasználói interakcióra gyorsan kell reagáljon.2. Ha egy műveletnek hosszabb időre van szüksége, a felhasználói felület egy vizuális indikátorral jelezze ezt a felhasználónak.

biztonság	A platform felhasználói rendszere standard biztonsági eljárásokkal kell rendelkezzen (a jelszavakat hashelni, illetve sózni kell, a bizalmas adatokat titkosítani kell, stb.).
megbízhatóság	Az alkalmazás biztonságosan kell eltárolja a felhasználók által feltöltött, illetve generált állományokat egy külső szerveren, vagy egy felhőalapú tárhelyen, az alkalmazás konfigurációnak megfelelően.

3.3. táblázat. Szervezési követelmények

környezet	Az alkalmazás nyílt forráskódú és platformfüggetlen keretrendszereket és eszközöket használ. Ezeket az eszközöket bármilyen számítógépen, bármelyik operációs rendszeren futtatni lehet. Lehetséges használni egy felhőalapú kódolási környezetet is, mint például a GitHub Codespaces, a könnyű hozzáférés érdekében.
fejlesztés	<ol style="list-style-type: none"> 1. A szoftver architektúrája nem függhet egy bizonyos kódszerkesztőtől vagy fejlesztői környezettől, viszont a legjobb fejlesztői élményt a Visual Studio Code kódszerkesztő nyújtja. 2. Az alkalmazás a TypeScript programozási nyelvben íródott és a Next.js keretrendszert használja. 3. Verziókövetéshez a Git rendszer volt használva, a forráskód pedig a GitHub platformon van tárolva.

3.4. táblázat. Külső követelmények

etikai és törvényi	<ol style="list-style-type: none"> 1. Az alkalmazásnak meg kell felelnie a GDPR (General Data Protection Regulation) szabályozásnak és erről értesítenie kell a felhasználókat. 2. Az alkalmazásnak elérhetővé kell tennie egy könnyen elérhető és átlátható adatvédelmi nyilatkozatot és felhasználási feltételeket.
---------------------------	---

4. fejezet

A szoftver tervezése

4.1. Fejlesztési eszközök

A szoftver kivitelezése során több olyan eszközt használtam, amik nagy mértékben közreműködtek a fejlesztési folyamat felgyorsításában és a fejlesztői élmény javításában.

- Kódszerkesztőnek a Microsoft által fejlesztett **Visual Studio Code** szoftvert használtam, ami egy TypeScript-ben írt Electron alapú kódszerkesztő. A Visual Studio Code előnyei közé sorolható a rengeteg kiegészítő (amik segítségével a fejlesztők szinte bármilyen programozási nyelvben tudnak fejleszteni), a beépített Git támogatás, az egyszerűség és a robusztus kezelőfelület.
- A szoftver verziókövetéséhez a **Git** rendszert használtam, ami nyílt forráskódú és nagyon elterjedt. A Git segítségével könnyen lehet nyomon követni a szoftver változásait és a különböző verziókat, ha pedig valami balul sült el, akkor könnyen vissza lehet állni egy korábbi verzióra.
- A forráskód tárolására a Microsoft által felvásárolt **GitHub** platformot használtam, ami egy nagyon népszerű és ingyenes Git tárhely, ráadásul nagyon sok hasznos funkcióval rendelkezik, mint például a GitHub Actions, amivel automatizálni lehet a szoftverfejlesztés bizonyos folyamatait.
- A forráskód statikus analízisére az **ESLint** nevű lintert használtam, amit a Next.js keretrendszer alapértelmezetten lefuttat minden produkciós build során. Az ESLint nem csak abban segít, hogy a kódunk egységes és elegáns legyen, hanem abban is, hogy a kódunk ne tartalmazzon potenciális hibákat, mint például a nem használt változók, vagy a rosszul végzett összehasonlítások.
- A forráskód stílusának meghatározásához a **Prettier** nevű csomagot használtam, amivel a kódunkat automatikusan formázni tudjuk bizonyos szabályok alapján. A Prettier-t ugyanakkor beépítettem az ESLint konfigurációba is, tehát linteléskor is tudunk formázási hibákat kiszemelni és javítani. A Visual Studio Code-ra írt Prettier kiegészítő segítségével minden mentéskor automatikusan formázódik a kódunk.

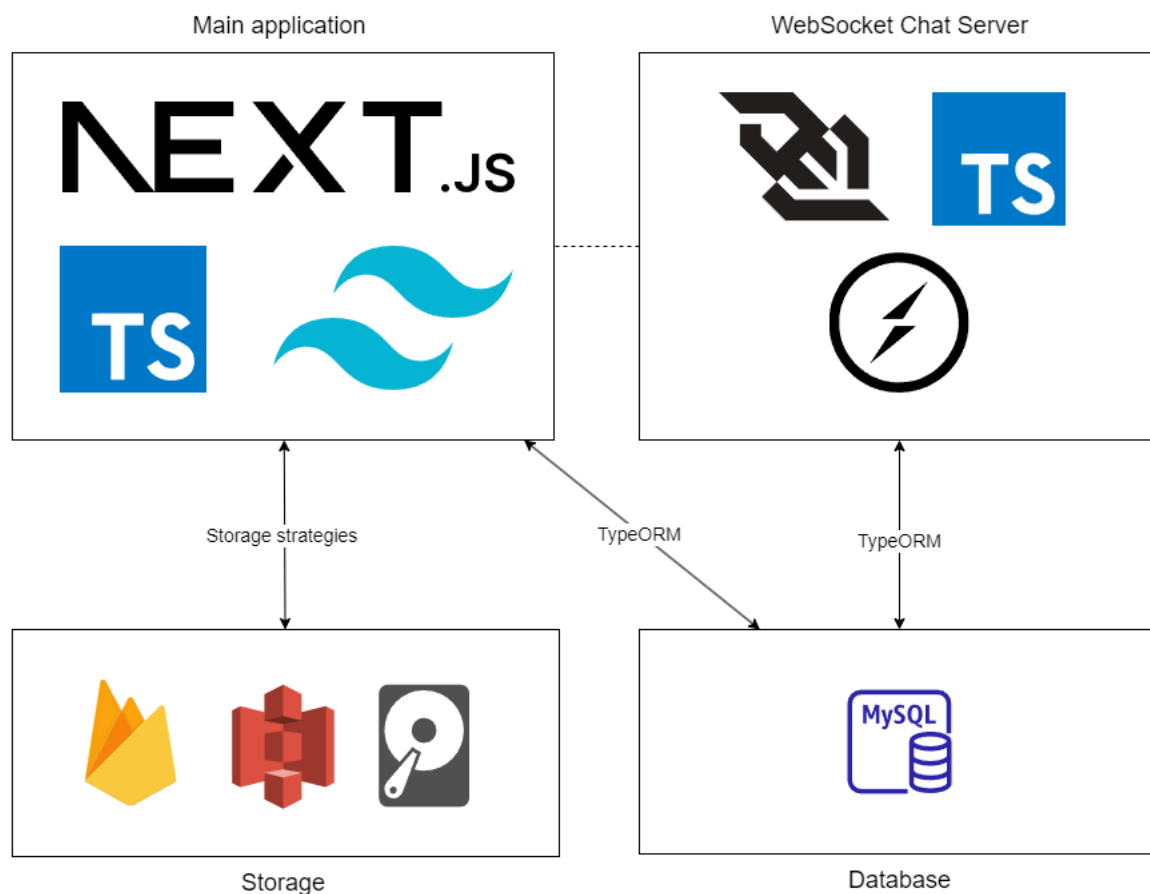
A projektet egy Arch Linux alapú Windows Subsystem for Linux (WSL) környezetben fejlesztettem. Bár a fejlesztést közvetlenül a Windows operációs rendszeren is lehetett

volna végezni, a WSL használatával sokkal egyszerűbb és kényelmesebb volt. A WSL egy olyan kompatibilitási réteg, amely lehetővé teszi a Linux alkalmazások futtatását Windows alatt.

4.2. Projektszerkezet

A szoftver két nagyobb komponensből tevődik össze: egy Next.js alapú webes alkalmazás és egy Socket.IO alapú WebSocket szerver. Mindkét komponens a TypeScript programozási nyelvben íródott, ugyanahhoz az adatbázishoz csatlakozik és ugyanabban a Git repository-ban van tárolva.

Két külső komponens is szerepet kapott a szoftveren belül: egy relációs adatbázis (ami lehet bármilyen, a fejlesztés során viszont MySQL-t használtam), valamint egy állománytárolási stratégia (ami lehet lokális, Amazon S3 alapú, vagy Firebase Storage alapú). A szoftver architektúráját a 4.1 ábra szemlélteti.



4.1. ábra. A szoftver architektúrája

A Next.js egyik jellegzetessége, hogy az összes route-ot a **pages** nevű könyvtárban tudjuk elhelyezni, ahol minden JSX (vagy a Project Raccoon esetében TSX) állomány egy-egy route-nak felel meg. Továbbá, a **pages/api** könyvtárban van lehetőségünk API végpontokat létrehozni egyes JS vagy TS állományokban.

A főprogram szerveroldali része egy REST API, ami azt jelenti, hogy az egyes modellekhez tartozik egy vagy több API végpont, amikre a kliensek HTTP kéréseket küldhetnek, a szerver pedig egy JSON választ ad vissza. A lekérés eredménye nagy mértékben függ a kliens által küldött HTTP kérés metódusától, például egy GET lekérés esetén a szerver egy adott modell elemeit adja vissza, egy POST lekérés létrehoz egy új modell elemet, egy PUT lekérés módosít, egy DELETE lekérés pedig töröl. Ennek előnye, hogy ugyanazt a végpontot tudjuk használni többféle műveletre és egy modellhez tartozó összes logika egy helyen lehet.

A legtöbb API endpoint tehát a következő formában néz ki:

- `/api/<modell>` - GET kérés esetén a modell összes elemét adja vissza, POST kérés esetén pedig létrehoz egy új modell elemet.
- `/api/<modell>/<id>` - GET kérés esetén a modell egy adott elemét adja vissza, PUT kérés esetén módosítja, DELETE kérés esetén pedig törli.
- `/api/<modell>/<id>/<action>` - olykor szükségünk van pontosabb műveletekre, például egy szerződéshez elfogadása.

A főprogram az MVC (model-view-controller) architektúrát követi. Az adatbázison végrehajtott műveleteket a TypeORM könyvtár segítségével valósítottam meg, a modellek definíciója pedig a `db/models` könyvtárban található. Ezeken a modelleken bizonyos kontroller függvények végeznek el CRUD műveleteket. Minden egyes modellhez tartozik a `controllers` könyvtárban egy-egy könyvtár, azokon belül pedig a kontrollerek implementációját tartalmazó állományok. Minden kontroller egy-egy, a fájlból exportált TypeScript függvény, amit majd az API route-ok definíciójában tudunk meghívni.

A 4.1 kódrészletben található példában egy kontroller sablonja látható.

4.1. kódrészlet. Kontroller sablon

```
import { NextApiRequest, NextApiResponse } from 'next';

export const index = async (req: NextApiRequest, res: NextApiResponse) => {
  // req alapján lekérni az összes elemet ebből a modellből es visszaadni
  // oket a res objektumon keresztül
};

export const create = async (req: NextApiRequest, res: NextApiResponse) => {
  // req alapján létrehozni egy új elemet ebből a modellből es visszaadni
  // azt a res objektumon keresztül
};

// ...
```

Egy API route állományában az általam fejlesztett `next-bar` JavaScript könyvtár segítségével könnyen tudjuk összekapcsolni a különböző HTTP metódusokat a hozzájuk tartozó kontrollerekhez.

A 4.2 kódrészletben látható egy példa a két komponens összehangolására.

4.2. kódrészlet. API route definíciója

```
import bar from 'next-bar';

import * as sampleController from '@/controllers/sample/sampleController';

export default bar({
  get: sampleController.index,
  post: sampleController.create,
  // ...
});
```

Az API route-ok által visszaküldött válaszok JSON formátumban érkeznek. Ezek a JSON objektumok rendszerint a 4.1 táblázatban látható mezőket tartalmazzák.

4.1. táblázat. API válasz mezői

mező	típus	leírás
ok	boolean	Jelzi, hogy a lekérés sikeres-e (ez az információ viszont a HTTP státuszkódban is át van adva).
error	string?	Sikertelen lekérések esetén ez a mező egy hibaüzenet kódját tartalmazza. A hibaüzenet kódja egy egyedi string (pl.: CONTRACT_NOT_FOUND), amit a frontend majd feldolgoz egy olvasható hibaüzenetté.
...rest	multiple	Sikeres lekérések esetén a további mezők tartalmazzák az endpoint által elvárt adatokat.

A szoftver frontend része React-alapú, a **pages** könyvtárban található TSX kiterjesztésű állományok tartalmazzák az egyes oldalak implementációját. Az oldalak rendszerint az Axios nevű HTTP kliens könyvtár segítségével hajtják végre a lekéréseket az előbb említett API route-ok felé.

Az Axios egy olyan könyvtár, ami a böngésző natív XMLHttpRequest API-ját használja fel a HTTP lekérések végrehajtására és a válaszok feldolgozására. Részben azért robusztusabb a natív Fetch API-nál [9], mivel jobb a hibakezelése (a fetch csak akkor eredményez kivételt, ha a HTTP lekérés 500-as hibakóddal tér vissza, az Axios pedig minden nem 200-as státuszkódot hibának tekint), könnyebben lehet az adatokat felküldeni, illetve a fogadott adatokat feldolgozni, valamint támogatja az interceptorokat és a kérések megszakítását.

Az Axios lekérések végrehajtását több API service építi össze. Gyakorlatilag minden egyes kontrollernek van egy megfelelő API service-je a frontenden, ami olyan metódusokat tartalmaz, amik képesek a kontroller különböző feladatainak végrehajtásához szükséges API lekéréseket kezdeményezni, illetve az API route-ok által visszaadott válaszokat feldolgozni és visszaadni a metódust meghívó kódrészletnek. Az API service-ek egyszer jönnek létre és egy **APIService** nevű wrapper osztályban találhatóak, amiből mindig csak egy példány létezik az alkalmazásban. A 4.7 ábrán látható a különféle API service-ekben implementált metódusok osztálydiagramja.

A frontend a következő főbb könyvtárakból tevődik össze:

- **components:** tartalmazza az újra felhasználható, illetve a felhasználás-specifikus komponenseket, ezekből épülnek fel az oldal elemei.
- **layouts:** különböző oldalakhoz tartozó sémát tartalmazza, ezek önmagukban olyan React komponensek, amik egy-egy oldalt „átkarolnak”, az oldalak mindig ezekben a komponensekbe vannak beágyazva. Külön layout van például a főoldalra, a tartalomoldalakra (pl. a felhasználási feltételek), az ügyfélkapura, illetve a dokumentációra.
- **styles:** SCSS fájlokat tartalmaz, amik meghatározzák a felhasználói felület által használt színeket, betűtípusokat, valamint egyes globális elemek stílusait. Ezekben a fájlokban lehetőség van a Tailwind direktíváit is használni, így még jobban tudjuk egységessé tenni a felület kinézetét.
- **assets:** olyan képeket, ikonokat és más erőforrásokat tartalmaz, amiket buildeléskor a Next.js automatikusan optimalizál és tömörít.
- **public:** a Next.js által használt statikus fájlokat tartalmazó könyvtár. Ezeket a fájlokat elérhetjük, ha az alkalmazás domainje utáni részben beírjuk a relatív címüket.

4.3. UI komponensek

Amint az előzőekben említettem, a Tailwind úgy van megtervezve, hogy ösztönözze a fejlesztőket arra, hogy újrahasznosítható komponenseket hozzanak létre az alkalmazásaik keretén belül, ezáltal megszabadulván a felesleges kódismétlődésektől.

A projekt keretén belül több olyan komponens is van, ami több helyen is fel van használva, mint például egy saját gomb komponens, betöltésjelző, toaster, illusztrációk, valamint a layout tervezését elősegítő komponensek (dobozok, oszlopok, stb.).

Ezek mellett vannak kontextus-specifikus komponensek is, melyek egy bizonyos célt szolgálnak, egy bizonyos feladat elvégzésére vannak felhasználva. Ilyenek például a különféle űrlapok, amiket a Formik nevű Reacthez írt könyvtárnak köszönhetően könnyedén meg lehetett tervezni.

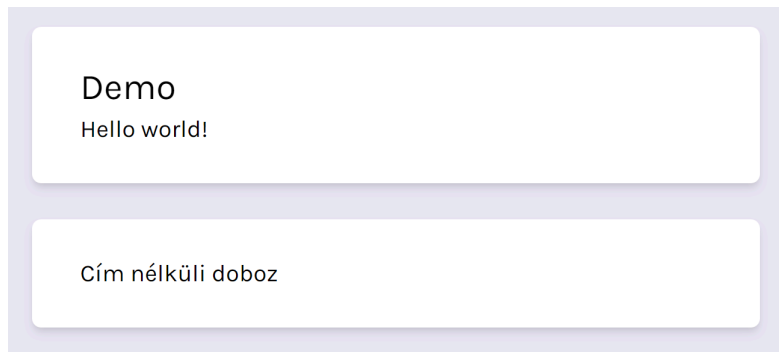
A következőkben bemutatok két common komponens, amiket a projekt keretén belül sok helyen felhasználtam.

Layout komponensek

Ezek a komponensek segítenek abban, hogy egy egységes szerkezetet tudjunk megteremteni az oldalainkban, legfőképpen az ügyfélkapuban.

A layout komponensek közé sorolható például a **Box** komponens, ami paraméterül kap egy opcionális címet, valamint egy tartalmat, amit meg kell jelenítsen. A komponens a 4.2 ábrán látható módon jeleníti meg a renderelni kívánt tartalmat.

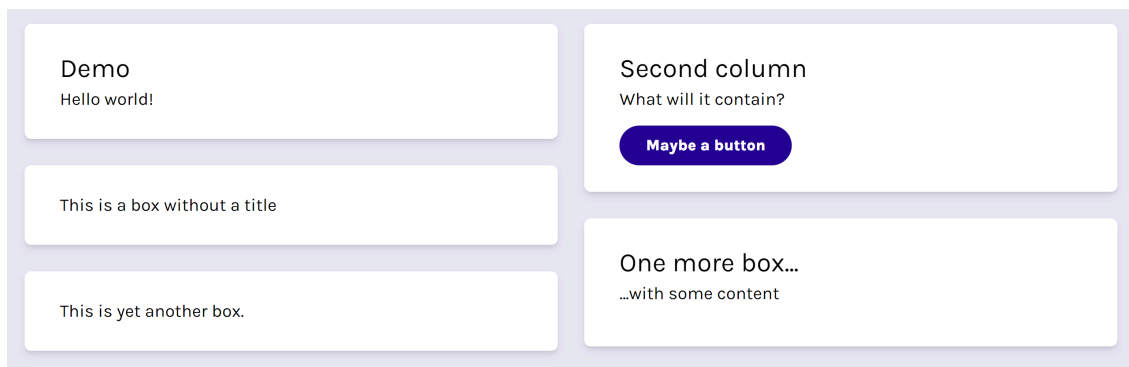
A **Columns**, illetve a **Column** komponensek segítenek abban, hogy a tartalmat több oszlopban jelenítsük meg. Ez legfőképpen az ügyfélkapuban hasznos, mivel sok információ két oszlopban van elhelyezve. Az oszlopokat a CSS Flexbox technológiával valósítottam meg.



4.2. ábra. Box komponens

Ez a komponens mobileszközökre úgy van optimalizálva, hogy az egyes oszlopok tartalmi egymás alatt helyezkedjenek el, így a tartalom megfelelően adaptálódik bármilyen képernyőmérethez.

Az oszlopkomponenseket a leglátványosabb a **Box** komponenssel együtt használni, ahogy ezt a 4.3 ábrán is láthatjuk.



4.3. ábra. Oszlopok és dobozok

Toaster

A toaster feladata az, hogy a felhasználói felületen különböző üzeneteket jelenítsen meg toast-ok formájában. A toast-ok egy rövid, általában egymondatos üzenetek, amiket a felhasználók számára fontos információk közlése céljából használunk. A projekten belül ezeket a toast-okat egy saját implementációval valósítottam meg, nem használva külső könyvtárat.

Egy toast tartalmazhat egy címet, egy leírást, valamint egy gombot, amivel el lehet tüntetni azt. A toast-okat a szoftver a képernyő alsó részén jeleníti meg középen, valamint négy különböző típust különböztet meg:

- **info**: informatív toast, általános információt közöl.
- **success**: sikeres műveletet közlő toast.
- **warning**: figyelmeztető toast, akkor jelenik meg amikor egy művelet nem járt sikerrel, viszont nem sikerült az elvárásoknak megfelelően végrehajtani.

- **danger:** hibát jelző toast, akkor jelenik meg amikor egy művelet végrehajtása során hiba történt.

A toaster-re lehet bárholnan hivatkozni a `lib/toaster` állományon keresztül. Az állomány alapértelmezett exportja egy `Toaster` osztály példánya, ami olyan metódusokat tartalmaz, amivel egy toast-okat tartalmazó stack-be be lehet illeszteni újabb toast-okat, illetve azokat el lehet távolítani.

A 4.4 ábrán látható egy példa a Toast komponensekre.



4.4. ábra. Toast komponensek

4.4. Adatmodellek

Az alkalmazásban számos modell van, amik egymással szoros kapcsolatban állnak. Ezek közül a két legfontosabb modell, ami a felhasználói részt illeti, az a **FilledContract** (kitöltött szerződés) és a **User** (felhasználó) modell.

Megjegyzések

- Minden modell rendelkezik egy **datetime** típusú, nullable **deletedAt** mezővel, ami azt jelzi, hogy a modell törölve van-e. Ez azt jelenti, hogy a projekt keretén belül a modellek soha nem kerülnek végleges törlésre, hanem soft-delete logikát követve vannak törölve. Ezt a TypeORM könyvtár könnyedén meg tudja oldani nekünk.
- A modellek között nincsenek foreign key kapcsolatok. Ez azért van, mert a foreign key-ek sokszor felesleges komplexitást okozhatnak, illetve az adatbázis teljesítménye is romolhat tőlük. Továbbá, a foreign key-ek kihagyása jobba teszi az adatbázis kompatibilitását is különféle stratégiákkal és platformokkal. A PlanetScale on-demand MySQL szolgáltatása például nem támogatja a foreign key-eket, az adatbázisuk alatt használt Vitess technológia miatt.
- A modellek típusában a **?** azt jelenti, hogy az a mező lehet null értékű. A **string?** típus tehát azt jelenti, hogy a mező string típusú, de lehet null.

Az alábbiakban bemutatom a felhasználói részhez használt legfontosabb adatbázis modelleket.

User

Egy felhasználó adatait tartalmazza, beleértve egyaránt a bejelentkezéshez szükséges adatokat, illetve a személyes adatokat, amik szerződéskötéskor fel lesznek használva.

A **User** modell a 4.2 táblázatban látható mezőket tartalmazza.

Mező	Típus	Leírás
id	number, PK	A felhasználó egyedi azonosítója.
createdAt	datetime	A felhasználó regisztrációjának dátuma.
updatedAt	datetime	A felhasználó adatainak utolsó módosítási dátuma.
name	string	A felhasználó teljes neve, beleértve a vezetéknévét, keresztnévét, és édesapa nevének kezdőbetűjét.
email	string	A felhasználó e-mail címe. Ez lesz felhasználva a belépéshez, illetve a szerződésekhez való meghíváshoz.
password	string	A felhasználó jelszava bcrypt algoritmussal hashelve.
image	string?	A felhasználó profilképének relatív URL-je. Amennyiben a felhasználó nem rendelkezik profilképpel, egy alapértelmezett profilképet fog kapni.
isAdmin	boolean	Jelzi, hogy a felhasználó rendelkezik-e adminisztrátori jogosultsággal.
motherName	string?	A felhasználó édesanyjának lánykori neve. Szerződések kitöltésénél használatos.
motherBirthDate	datetime?	A felhasználó édesanyjának születési dátuma. Szerződések kitöltésénél használatos.
nationality	string?	A felhasználó nemzetisége. Szerződések kitöltésénél használatos.
personalIdentifierType	ld. (4.3)	A dokumentum típusa, amellyel a felhasználó igazolni szeretné a személyazonosságát. Ezt ugyanakkor a szerződés kitöltésekor is felhasználjuk.
personalIdentifier	string?	A felhasználó által megadott személyazonosító dokumentum száma. Szerződések kitöltésénél használatos.
phoneNumber	string?	A felhasználó telefonszáma. Szerződések kitöltésénél használatos.
birthDate	datetime?	A felhasználó születési dátuma. Szerződések kitöltésénél használatos.
birthPlace	string?	A felhasználó születési helye. Szerződések kitöltésénél használatos.

4.2. táblázat. User modell

A `PersonalIdentifierType` típus egy enum érték, ami a 4.3 táblázatban látható értékek egyikét veheti fel.

Érték	Leírás
IDENTITY_CARD	Egy egyszerű személyazonossági igazolvány.
PASSPORT	Egy kormányzati szerv által kiadott útlevél.
DRIVING_LICENSE	Egy gépjárművezetői engedély, ami a felhasználót azonosítja.

4.3. táblázat. `PersonalIdentifierType` enum típus

FilledContract

A `FilledContract` olyan szerződés, ami két `felhasználó` között jön létre. Ez a szerződés adatokat tartalmaz az eladott tulajdonról, illetve személyes adatokat úgy az eladóról, mint a vevőről. A személyes adatok a felhasználók által megadott személyes jellegű információkból lesznek átmásolva, a tulajdonra vonatkozó adatok pedig a tulajdon objektumhoz hozzárendelt adatokból lesznek származtatva.

A `FilledContract` modell a 4.4 táblázatban található mezőket tartalmazza.

Mező	Típus	Leírás
id	number, PK	A szerződés egyedi azonosítója.
createdAt	datetime	A szerződés létrehozásának dátuma.
updatedAt	datetime	A szerződés adatainak utolsó módosítási dátuma.
friendlyName	string	Egy megnevezés, amit az eladó ad meg a szerződésnek, annak érdekében, hogy azt könnyebben lehessen azonosítani.
filename	string?	A templating rendszer által kigenerált állomány relatív címe. Ennek a mezőnek csak akkor lesz értéke, ha a szerződés véglegesítve van, ami azt jelenti, hogy az összes fél aláírta azt.
contract	Contract	A szerződéshez tartozó sablon. Ezeket a sablonokat az adminisztrációs részben lehet kezelni.
options	ld. (4.5)	A szerződéshez tartozó mezők értékei.
userId	number, FK	A szerződés eladójának azonosítója.
buyerId	number, FK	A szerződés vevőjének azonosítója.
accepted	boolean	Jelzi, hogy a szerződéshez meghívott vevő elfogadta-e a szerződést.
sellerSignedAt	datetime?	Amennyiben nem null, meghatározza, mikor írta alá a szerződést az eladó.
buyerSignedAt	datetime?	Amennyiben nem null, meghatározza, mikor írta alá a szerződést a vevő.

4.4. táblázat. `FilledContract` modell

FilledContractOption

A `FilledContractOption` modell a `FilledContract` modellhez tartozó mezők értékeit tartalmazza. A szerződéssablonok úgy vannak megtervezve, hogy azokhoz többféle mezőt lehessen hozzárendelni, különböző mezőtípusokkal. Ez a modell magában foglalja egyetlen egy mező értékét egy adott szerződésben.

A `FilledContractOption` modell a 4.5 táblázatban látható mezőket tartalmazza.

Mező	Típus	Leírás
id	number, PK	A mezőérték egyedi azonosítója.
createdAt	datetime	A mezőérték létrehozásának dátuma. Ez legtöbbször megegyezik a szerződés létrehozásának dátumával.
updatedAt	datetime	A mezőérték adatainak utolsó módosítási dátuma.
filledContract	ld. (4.4)	A szerződés, amihez ez a mezőérték tartozik.
option	ContractOption	A mezőértékhez tartozó mező definíciója. Ezek a definíciók az adminisztrátor által vannak megadva egy <code>Contract</code> objektumon belül.
value	string?	A mezőérték stringbeli reprezentációja. Amennyiben null, a mező még nem volt kitöltve. Olykor a mezők értékeit át kell konvertálni string formátumba mentés előtt, például a számokat vagy a dátumokat.

4.5. táblázat. `FilledContractOption` modell

Egy `FilledContractOption` értéke olykor automatikusan generálódhat a szerződés létrehozásakor, illetve egy szerződés vevő általi elfogadásakor. Ilyen például a felhasználók személyes adatai, illetve a tulajdon adatai.

ChatMessage

A `ChatMessage` modell egy adott szerződéshez tartozó chat üzenet adatait tartalmazza. Önmagában csak egy reláció egy felhasználóhoz, illetve egy szerződéshez, ami magában foglalja az üzenet tartalmát is.

A `ChatMessage` modell a 4.6 táblázatban látható mezőket tartalmazza.

Mező	Típus	Leírás
uuid	string, PK	A chat üzenet egyedi azonosítója UUID formátumban.
createdAt	datetime	A chat üzenet létrehozásának dátuma.
user	ld. (4.2)	A felhasználó, aki elküldte az üzenetet.
filledContract	ld. (4.4)	A szerződés, amihez ez az üzenet tartozik.

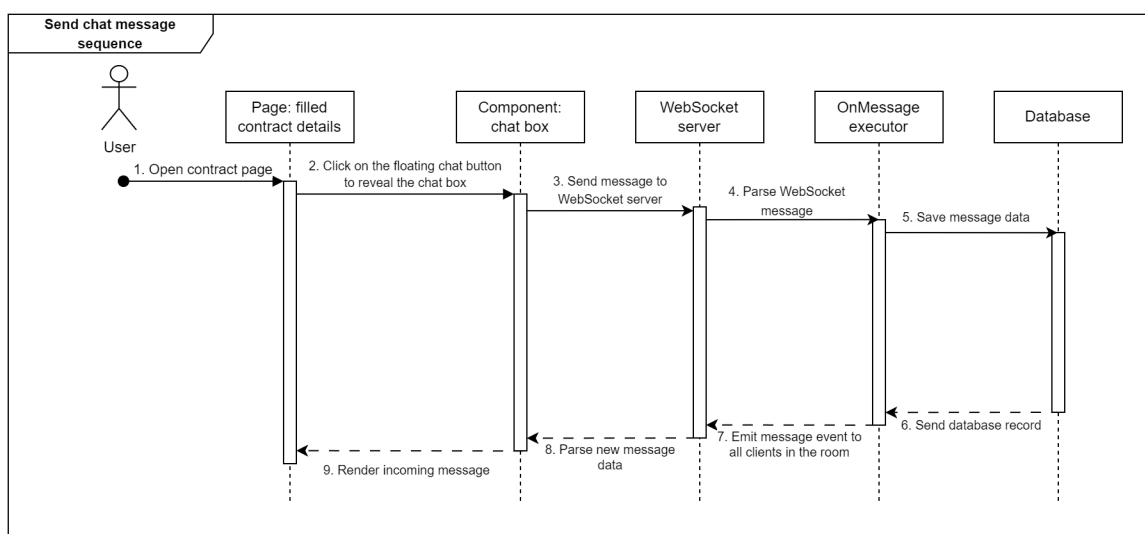
message	string	Az üzenet tartalma.
---------	--------	---------------------

4.6. táblázat. ChatMessage modell

4.5. Diagramok

4.5.1. Sequence diagram

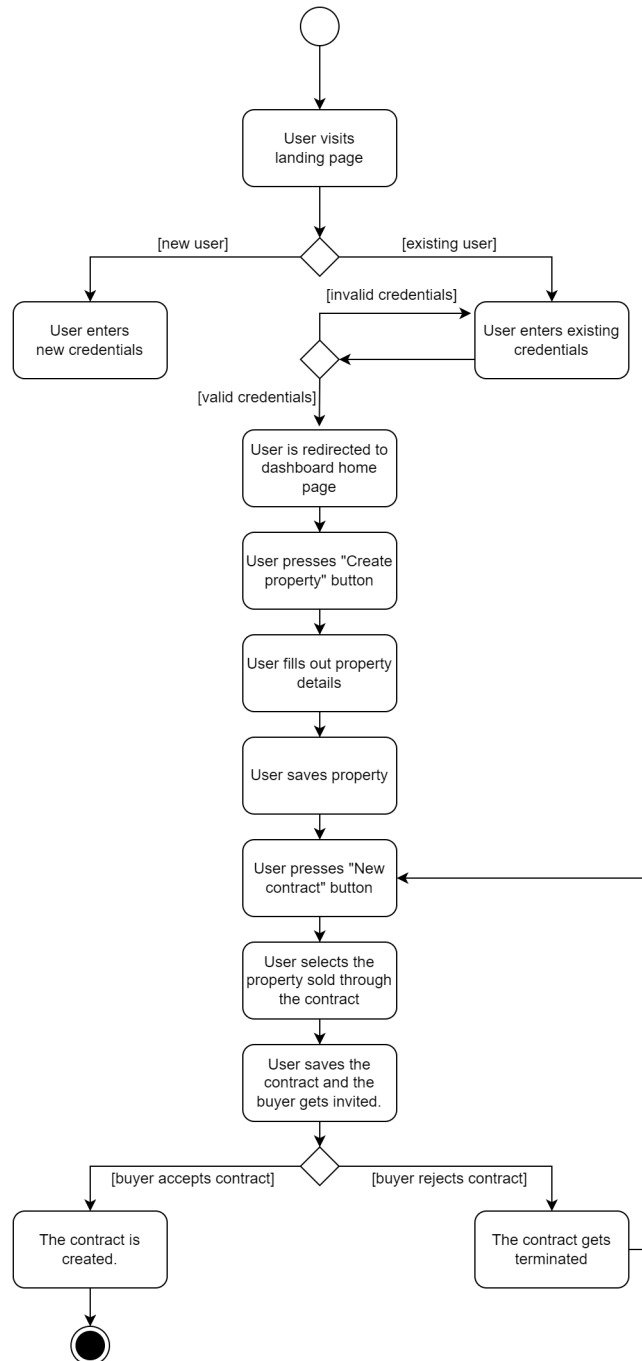
A 4.5 diagram bemutatja az üzenetküldés folyamatát a szerződéseken belül található chat funkción keresztül.



4.5. ábra. Üzenetküldés

4.5.2. Activity diagram

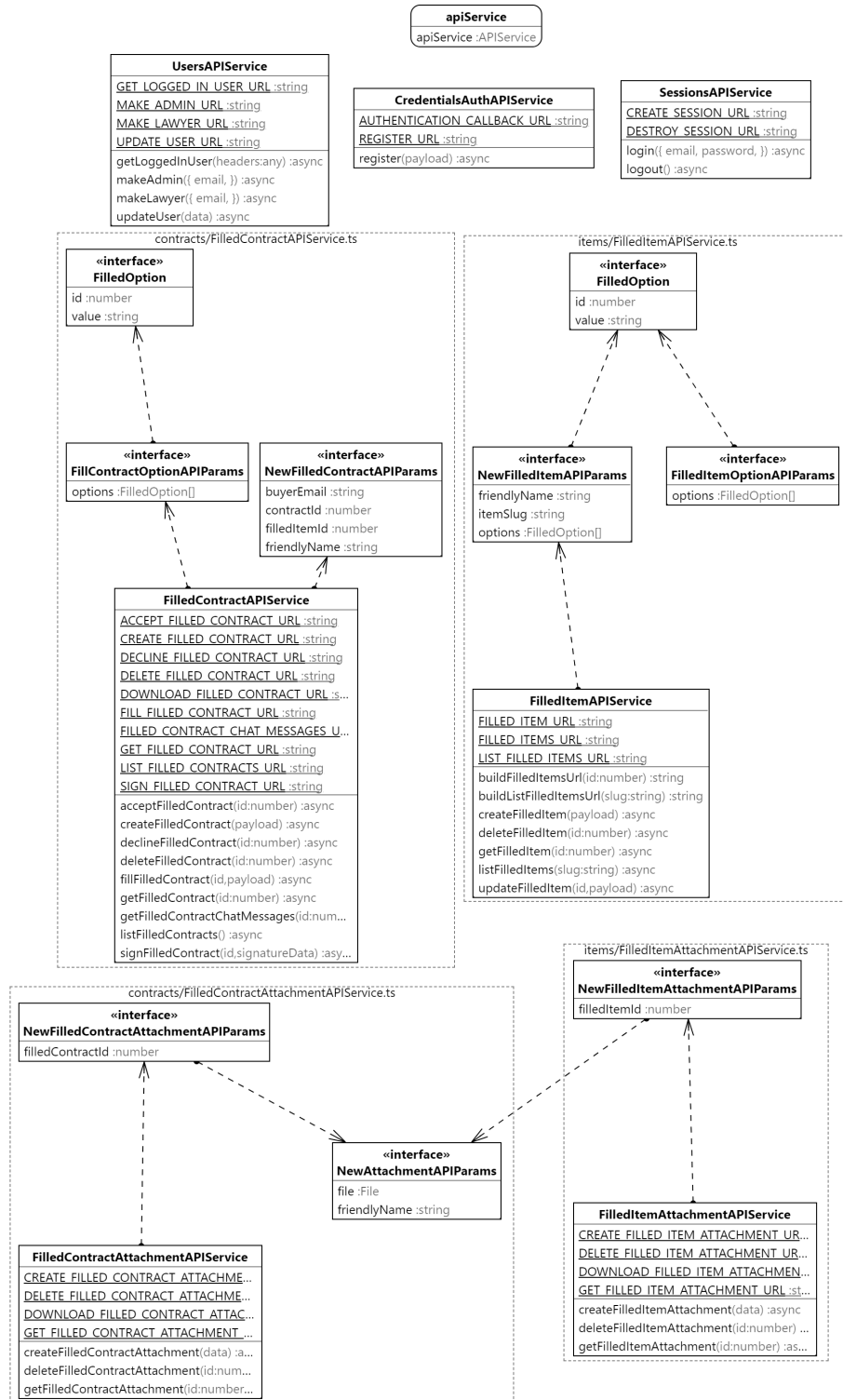
A 4.6 diagram bemutatja a szerződés létrehozásának folyamatát az eladó szemszögéből.



4.6. ábra. Szerződés létrehozása

4.5.3. Osztálydiagram

A 4.7 diagram bemutatja a frontend által felhasznált API service-eket, illetve azok metódusait.



4.7. ábra. API service-ek

4.6. WebSocket alapú chat működési elve

A Project Raccoon-ban az egyes szerződésekhez tartozó chat funkcionalitás a WebSocket technológia segítségével van megvalósítva, a Socket.IO könyvtár által nyújtott absztrakciós rétegen keresztül. A projekten belül van egy különálló WebSocket szerver, aminek a forráskódját a **chat** nevű könyvtárban lehet megtalálni. Ennek a szervernek szoros kapcsolata van a projekt egészére kiható TypeORM-es adatbázisműveleteket végző állományokkal, valamint az alkalmazás fő konfigurációjával.

A Socket.IO könyvtár képes bizonyos eseményeket kezelni, amiket mi tudunk definiálni a szerveren belül. Az eseményeknek van egy tetszőleges neve, valamint egy adatokat tartalmazó objektum paramétere. A kliens ezeket az eseményeket tudja elküldeni a szerver felé, a szerver pedig minden eseményre egy eseménykezelő függvényt tud meghívni. Ez fordítva is igaz, a szerver is ugyanúgy tud eseményeket küldeni a kliens felé, amikre a kliens hallgatózhat.

A különálló WebSocket szerver egy Express szerveren keresztül fut egy dedikált porton. A kliens majd ezen a porton keresztül tud csatlakozni amikor megnyitjuk egy szerződés oldalát.

Tervezési szempontból minden olyan eseményre, amit a WebSocket szerver kezelni tud, létezik egy **executor** osztály van, aminek az implementációja a **chat/executors** könyvtárban található. Az **executor**-ok a következő absztrakt osztályból származtatnak:

4.3. kódrészlet. Executor osztály alapja

```
import { Server, Socket } from 'socket.io';

export abstract class EventExecutor<T = any, R = BaseExecutorResponse> {
  public abstract execute(io: Server, socket: Socket, data: T): Promise<R>;
}
```

Tehát minden **executor**-nak van egy **execute** metódusa, ami paraméterül kapja a Socket.IO szerver példányt, a kliens által létrejött WebSocket kapcsolatot, valamint az eseményben kapott adatokat. A létrehozott **executor**-okat egy **ExectuorRepository** nevű osztályban hozzuk létre és regisztráljuk különféle eseményekhez.

Az **ExectuorRepository**-nak van egy **handle** nevű metódusa, ami paraméterül kapja az eseményt, amire le kell egy callback-et futtatni, valamint az **executor**-ok által elvárt adatokat. A metódus megvizsgálja, hogy van-e az adott eseményhez hozzárendelve bármilyen **executor**, ha pedig van, továbbítja a lekérést annak **execute** metódusához.

A kérésekre jövő válaszokat az **executor**-ok fogják emitálni a kliens felé. Az **execute** metódusok egy objektumot térítenek vissza, ami a következő mezőket tartalmazza:

- **ok**: boolean, jelzi, hogy a lekérés sikeres volt-e.
- **error**: string vagy null, amennyiben egy hiba történt, itt lehet azt pontosabban meghatározni.

Amennyiben a lekérés sikeres volt, az **ok** mező értéke **true** lesz és a **handle** metódus futása véget ér. Ha viszont **false** értéket kapunk, akkor a **handle** metódus egy **error** eseményt fog emitálni a kliens felé, ami tartalmazza a hiba kódját (az **error**), mezőből, vagy annak hiányában az **INTERNAL_SERVER_ERROR** hibakódot.

A WebSocket szerver a következő eseményeket képes kezelni:

- `join` - egy kliens csatlakozott egy adott szerződéshez tartozó chatszobához.
- `leave` - egy kliens lecsatlakozott egy adott szerződéshez tartozó chatszobából.
- `chat-message` - egy kliens elküldött egy üzenetet egy adott szerződéshez tartozó chatszobában.

Mindegyik eseménytípus megkap paraméterül egy `filledContractId` mezőt, ami azonosítja azt a szobát, ahol az esemény lefut. A `chat-message` esemény továbbá tartalmaz egy `message` mezőt is, ami magát az üzenetet tartalmazza.

5. fejezet

A szoftver működésének részletes bemutatása

5.1. Landing page és egyéb tartalmi oldalak

A frontend keretén belül megtalálható egy külön layout, amit a tartalmi oldalak esetében használtam fel. Ezek a tartalmi oldalak közé sorolhatóak maga az alkalmazás főoldala (landing page), a jogi oldalak (adatvédelmi nyilatkozat, felhasználási feltételek), a kapcsolatfelvételi űrlap, valamint a felhasználói dokumentáció.

A landing page egy egyszerű, de letisztult megjelenésű oldal, amelyen a legelső információ ami megjelenik, az a projekt szlogenje, egy rövid leírás arról, mire szolgál a szoftver, majd egy call-to-action gomb, ami elvezet a regisztrációs oldalra. A landing page-en továbbá fel van vázolva három egyszerű lépésben az, hogyan lehet létrehozni és véglegesíteni a szoftver keretén belül egy szerződést.

A landing page-en belül számos olyan szabadon használható kép található, amelyek az Undraw nevű oldalról származnak. Ezek a grafikák nagyon modern és letisztult kinézzel rendelkeznek, valamint SVG formátumban érhetőek el, ami nem csak azt jelenti, hogy gyorsabban lehet őket betölteni és renderelni, de még szabadon módosíthatóak is, így a szoftver színeihez és stílusához igazíthatóak.

A felhasználási feltételek és az adatvédelmi nyilatkozat oldalakon ugyanúgy egy Undraw által biztosított grafika látható a cím alatt, majd maga az oldal tartalma. Az oldal tartalmát Markdown formátumban tároljuk, a MarkdownX (MDX) technológiának köszönhetően pedig könnyedén be tudjuk ágyazni ezeket a tartalmakat a React komponenseinkbe, ebben az esetben a jogi oldalakba.

A dokumentáció oldalak szintén MDX formátumban léteznek, viszont ezek már rendelkeznek bizonyos metaadatokkal is, amiket a **front-matter** nevű technológiának köszönhetően ki tudunk nyerni. Ezek az oldalak egy catch-all route-on keresztül vannak létrehozva, ami azt jelenti, hogy a `/docs/*` útvonalon belül bármilyen útvonalat megadunk, azt kikeresi a `content` könyvtárból, a felhasználó által használt nyelv függvényében.

5.2. Felhasználók hitelesítése és adataik kezelése

Két oldal található a szoftverben, ahol a hitelesítést lehet kezelni. Az első a regisztrációs oldal, ami a `/register` útvonalon érhető el. Az űrlapot kitöltve, majd a regisztrációs

gombra kattintva egy POST lekérés lesz elküldve a szerver felé, ami egy JSON objektummal rendelkezik a következő adatokkal:

```
export interface CredentialsRegisterAPIRequest {  
  name: string;  
  email: string;  
  password: string;  
  password2: string;  
}
```

A szerver számos validációt végez el a beérkező adatokon, például ellenőrzi, hogy a megadott e-mail címmel már regisztráltak-e, vagy hogy a két jelszó egyezik-e. Azokat a validációs lépéseket amik nem szükségeltetik az adatbázisban való keresést (például a jelszavak egyezősége), azokat a frontend oldalon is elvégezzük, így a felhasználó azonnal értesülhet arról, hogy valami nem stimmel az általa megadott adatokkal. Ezt a kliensoldali validációt a Formik nevű könyvtár által végezzük el.

Sikeres regisztráció esetén a felhasználó át lesz irányítva a második oldalra, ami a hitelesítéshez kötődik: a bejelentkezés oldalra, amit a `/login` útvonalon érhetünk el. Itt egy hasonló űrlapot találunk, mint a regisztrációs űrlap, viszont itt csak az e-mail címet és a jelszót kell megadni. Az űrlapot elküldve létrejön egy POST lekérés az `/api/sessions` végpontra, ami hitelesíteni próbálja a felhasználót.

A felhasználók hitelesítését és a session-ok létrehozását az `iron-session` könyvtárral valósítjuk meg [17]. Az `iron-session` egy olyan nyílt forráskódú könyvtár, aminek segítségével könnyedén tudunk egy hitelesítéssel rendszert implementálni az alkalmazásunkban. First-class támogatást nyújt a Next.js keretrendszerhez, így tökéletes választás a mi céljainkra.

A session-ök létrehozása egy titkosított cookie segítségével történik („seals”), amit a felhasználó böngészőjében tárolunk. Ez a cookie HTTP-only, ami azt jelenti, hogy a böngésző nem engedi meg a JavaScript kódnak hogy hozzáférjen ahhoz, hanem csupán a kezdetleges HTTP lekérésekben elérhető. Ezen kívül a cookie olyan módon van titkosítva egy általunk megadott titkosítási kulccsal, hogy azt csak a szerver tudja dekódolni. Ez a hitelesítési stratégia nagy mértékben hasonlít arra, amit az olyan keretrendszerek is használnak, mint a Ruby on Rails.

A frontenden továbbá létrehoztunk egy `useCurrentUser` hook-ot, ami két dolgot térít vissza: egy `currentUser` objektumot, ami ha null értékű, akkor nincs bejelentkezett felhasználó, ha pedig nem null, akkor tartalmazza a felhasználó adatait, valamint egy `setCurrentUser` függvényt, amivel be lehet állítani a bejelentkezett felhasználót. Ez a hook továbbá az oldal betöltésekor egy GET lekérést küld a szerver felé, amivel a jelenlegi felhasználó adatait lekéri (amennyiben létezik). Ezt a hook-ot a frontend-en belül bárhol tudjuk használni, hogy hozzáférjünk a felhasználó adataihoz, például a jogainak megtekintéséhez.

5.3. Tulajdonok és szerződések kezelése

A tulajdonok és szerződések kezelése a szoftver legfontosabb komponensét jelenti. A felhasználók itt leltározhatják és kezelhetik a tulajdonjaikat, valamint később azokat felhasználhatják egy adásvételi szerződésben.

A tulajdonok kategóriákra vannak osztva, amit a szoftver üzemeltetői hoznak létre. A felhasználók minden egyes kategóriára lekérhetik egy GET request-en keresztül a tulajdonjaikat. A kategóriákhoz tartozó tulajdonokat a szerver JSON formátumban téríti vissza egy tömbben, amik tartalmazzák a tulajdon alapvető információit.

A felhasználóknak ugyanakkor lehetőségük van ebben a kategóriában létrehozni egy új tulajdont. A tulajdon létrehozásakor minden esetben meg kell határozni egy tetszőleges nevet, amit a tulajdon azonosításához fog tudni majd használni a felhasználó. Ugyanakkor a tulajdonhoz társított mezők értékeit is meg kell adja a felhasználó a tulajdon létrehozási űrlapjában.

A meglévő tulajdonokra kattintva a felhasználók módosíthatják a tulajdonhoz rendelt információk mezők értékét, valamint csatolmányokat rendelhetnek hozzá. Csatolmányokat csak akkor lehet hozzárendelni egy tulajdonhoz, ha az létre lett már hozva.

A szerződések a tulajdonokhoz hasonló módon működnek, viszont itt több funkció válik elérhetővé. A frontend-en megkülönböztetünk három Box komponensben három különböző szerződéstípust: olyan szerződések, ahol mi vagyunk az eladók, olyan szerződések ahol mi vagyunk a vevők, valamint olyan szerződések, ahol ügyvédi vagy tanúi szerepet játszunk. A szerződések listájában kisebb dobozokban alapszolgáltatásokat kaphatunk a szerződésről, mint például azt, hogy ki a vevő, milyen szerződéstípusról van szó, melyik tulajdonra érvényes ez a szerződés, valamint, hogy alá volt-e írva már a szerződés az egyes felek által.

Szerződés létrehozásakor meg kell adnunk a szerződés típusát. Ezeket a típusokat ugyancsak a szoftver üzemeltetői fogják kezelni és minden felhasználó számára használhatóak lesznek. Bizonyos szerződéstípus megkövetelheti azt is, hogy a felhasználó határozzon meg egy tulajdont, amire érvényes lesz a szerződés. A kategória és az esetleges tulajdon mellett az eladó felhasználó meg kell adja a vevő e-mail címét is, hogy meghívja azt a szerződéshez.

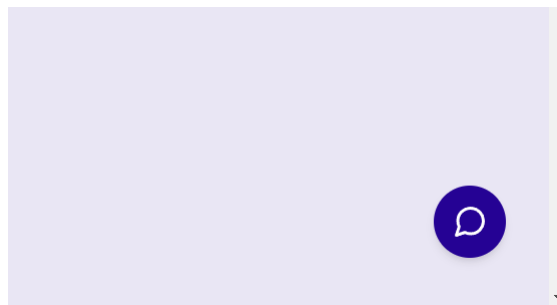
Bejövő szerződéseket el lehet fogadni, illetve vissza lehet utasítani. Egy szerződés elfogadása azt jelenti, hogy az elfogadó személy személyes adatai át lesznek másolva a szerződés vevővel kapcsolatos mezőibe, a vevő szabadon módosíthatja a hozzá tartozó mezők értékeit, tanút vagy ügyvédet hívhat meg, illetve természetesen aláírhatja a szerződést. Egy szerződés elutasítása a szerződés azonnali megsemmisülését vonja maga után.

Minden szerződéstípus tartalmaz egy sablont is, ami meghatározza, hogy a végleges szerződés hogy néz ki, illetve milyen módon lesznek az információk behelyettesítve a dokumentumban. A sablonokban moustache-hez hasonló szintaxis által vannak megadva azok a mezőazonosítók, amiket majd a rendszer behelyettesíti a felhasználók által megadott információkkal.

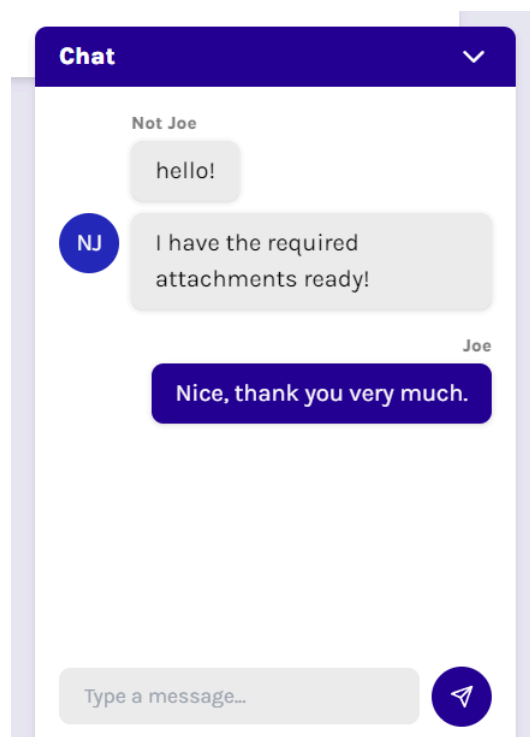
5.4. Szerződés felei közti üzenetküldés

Egy szerződés oldalának megnyitásakor, amennyiben a szoftver helyesen van konfigurálva és a WebSocket szerver elérhető, a frontend a Socket.IO könyvtár segítségével létre próbál hozni egy kapcsolatot a chat szerverrel, amit a `process.env.NEXT_PUBLIC_WS_SERVER_URL` környezeti változóban határoztunk meg. A frontend ugyanakkor próbál egy GET lekérést is küldeni az `/api/filled-contracts/:id/chat` végpontra, amivel lekéri a szerződéshez tartozó eddigi chat üzeneteket.

Amennyiben sikeresen le lehetett kérni az eddigi chat üzeneteket, valamint a Socket.IO kapcsolat is sikeresen létrejött, a képernyő jobb alsó sarkában látni fogunk egy chat gombot (5.1), amire rákattintva megnyílik a chat ablak (5.2).



5.1. ábra. A chat gomb a szerződés oldalán



5.2. ábra. A chat ablak néhány üzenettel

Amikor létrejön a WebSocket kapcsolat, a kliens regisztrál a szerver által küldött `chat-message` eseményre és hozzárendel egy függvényt, ami le fog futni akárhányszor a szervertől egy ilyen eseményt kap. A függvénynek két feladata van: egyrészt hozzá kell adnia az új üzenetet a már meglévő üzenetekhez, ezáltal hidrálva a chat ablak tartalmát, másrészt egy hangjelzést fog küldeni azoknak a klienseknek, akiknél nincs fókuszban a chat ablak (amennyiben az üzenet létrehozója nem egyezik meg a bejelentkezett felhasználóval). Éppen ezért az regisztrálunk egy eseményt a chat ablakban levő mező `onFocus` és `onBlur` eseményeire, majd egy state változó értékét beállítjuk attól függően, hogy a mező fókuszban van-e vagy sem. A hangjelzés küldéséhez a Howler JavaScript könyvtárat használtam, mivel egy egyszerű API-val rendelkeznek.

Development módban rálátást kaphatunk a szerver által kapott és küldött eseményekre:

```
[SOCKET] join (uid 1) -> { filledContractId: 1 }
[SOCKET] join (uid 1) <- { ok: true }
[SOCKET] join (uid 1) -> { filledContractId: 3 }
[SOCKET] join (uid 1) <- { ok: true }
[SOCKET] join (uid 2) -> { filledContractId: 3 }
[SOCKET] join (uid 2) <- { ok: true }
[SOCKET] chat-message (uid 2) -> { filledContractId: 3, message: 'hello!' }
[SOCKET] chat-message (uid 2) <- { ok: true }
[SOCKET] chat-message (uid 2) -> {
  filledContractId: 3,
  message: 'I have the required attachments ready!'
}
[SOCKET] chat-message (uid 2) <- { ok: true }
[SOCKET] chat-message (uid 1) -> { filledContractId: 3, message: 'Nice, thank you very much.' }
[SOCKET] chat-message (uid 1) <- { ok: true }
user disconnected
```

5.3. ábra. Konzol kimenet a chat szerver eseményeiről

5.5. Felhasználói beállítások

A Felhasználói beállítások oldalon a jelenleg bejelentkezett felhasználó három műveletet tud elvégezni:

- Módosítani tudja a nevét, illetve a profilképét. Amennyiben a felhasználó egy profilképet is feltölt, azt a szerver eltárolja a használt és bekonfigurált állománytárolási stratégiának megfelelően.
- Módosítani tudja a jelszavát azáltal, hogy megadja és megerősíti az új jelszavát, valamint megadja a régi jelszót. A jelszó módosításához a szerveren is validáljuk a régi jelszót, így biztosítva, hogy csak akkor tudjuk megváltoztatni a jelszavunkat, ha ismerjük a régit is.
- Módosítani tudja az személyes azonosítási információit, mint például az édesanya lánykori nevét, a születési helyét, hitelesítési dokumentumának számát, stb. Ezeket az adatokat a szerződések létrehozásakor mindig felhasználjuk és automatikusan ki lesznek töltve a megfelelő mezők.

Design szempontjából a három feladat három külön dobozban van elhelyezve, amik két oszlopban jelennek meg.

5.6. Felhasználók által feltöltött állományok eltárolása

A Project Raccoon-t igyekeztem úgy megtervezni, hogy az üzemeltetők meg tudják határozni, hol szeretnék lementeni a szoftveren belül a felhasználók által feltöltött vagy generált állományokat. Éppen ezért biztosítunk három különböző állománytárolási stratégiát:

- **lokális** - ebben a stratégiában az állományok a lokális lemezen, egy megadott könyvtáron belül (alapértelmezetten `storage`) lesznek lementve.
- **Amazon S3** - ebben a stratégiában egy bekonfigurált S3 bucket-ben lesznek eltárolva az állományok. Ehhez a stratégiához felhasználtuk az AWS JavaScript SDK-ját. Ennek egyik előnye, hogy a szoftvert olyan helyekre is tudjuk deployolni, ahol nincs lokális lemez, vagy nem lehet a lokális lemezre írni, például a Vercel platformra.
- **Firebase Storage** - ebben a stratégiában egy bekonfigurált Firebase instancia által biztosított tárolóban lesznek eltárolva az állományok. Ehhez a stratégiához felhasználtuk a `firebase-admin`, valamint a `@google-cloud/storage` könyvtárakat. Az S3-hoz hasonlóan itt is az egyik előny, hogy serverless platformokon is könnyedén tudjuk deployolni a szoftvert.

A három stratégia közül a lokális a leggyorsabb, mivel nem szükséges egy külső szerverre kapcsolódni az állományok leolvasásához vagy feltöltéséhez, viszont ez jár a legtöbb felelősséggel is, hiszen ilyenkor a szoftver üzemeltetőjének magának kell gondoskodnia arról, hogy a lokális lemez ne teljen meg, illetve a rajta eltárolt adatok ne vesszenek el. Az S3 és a Firebase Storage stratégiák esetében ezeket a feladatokat a szolgáltató végzi el, valamint ezek a platformok végtelenül skálázhatóak, így nem kell aggódni arról, hogy sok felhasználó esetén nem lesz elég tárhelyünk. Ugyanakkor az utóbbi két stratégia esetében egy fokkal lassúbbak lesznek az állományműveletek, valamint egy idő múlva lehet, hogy már több pénzbe fog kerülni a szolgáltatás.

Stratégiától függetlenül, a szoftver a következő könyvtárakban tárol adatokat:

- `attachments` - a szerződésekhez csatolt állományok
- `avatars` - a felhasználók által feltöltött profilképek
- `documents` - a véglegesített szerződések dokumentumai PDF formátumban
- `signatures` - a szerződések digitális aláírásai

5.7. Többnyelvűsítés (I18N)

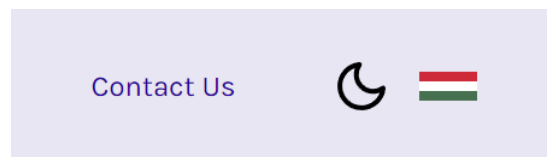
Fontosnak tekintettem azt, hogy a felhasználói felület legyen elérhető több nyelven is. Legfőképpen azért gondoltunk így, mivel megtörténhet az is, hogy a szoftvert egy Magyarországra költöző külföldi személy is igénybe akarja venni. Ezekben az esetekben hasznos, ha a felhasználók legalább angolul, vagy akár a saját anyanyelvükön is használhatják a szoftvert.

A lokalizációt az `i18next` könyvtár segítségével valósítottam meg, amit könnyedén be tudtunk integrálni a Next.js keretrendszerbe a `next-i18next` könyvtár segítségével.

Az `i18next` egy olyan könyvtár, amivel JSON formátumban tudunk tárolni szövegeket, amikre a felhasználói felületen bizonyos kulcsok megadásával tudunk hivatkozni. Így például a `'common:log-in'` szövegre hivatkozva a szoftver automatikusan a bejelentkezésnek megfelelő szöveget fogja renderelni, a felhasználó által kiválasztott nyelv függvényében.

A Next.js egyik jellegzetessége, hogy nagyon jól támogatja a többnyelvűsítést [8]. Éppen ezért nem kell sokat bajlódni azzal, hogy megtudjuk, milyen nyelvet választott a felhasználó, vagy hogy mikor melyik nyelvet jelenítsük meg.

A felhasználói felületen található egy nyelvválasztó is, ami a jelenlegi nyelvtől eltérő nyelvekhez tartozó ország zászlóit jeleníti meg. Valamely zászló ikonra való rákattintással a felhasználó nyelve át lesz állítva a megfelelő nyelvre.



5.4. ábra. A nyelvválasztó a landing page alján

A nyelvi fájlok egyszerűbb kezelésének érdekében a lokalizációs string-eket több namespace-re osztottam fel. A namespace-ek erőssége, hogy jobban lehet kategorizálni a szövegeket, valamint lehetőségünk van arra is, hogy csak egy bizonyos namespace-hez tartozó szövegeket töltsük be, ezáltal is csökkentve a szükséges lekérések számát és a betöltési időt. A Project Raccoon keretén belül a következő namespace-ek találhatóak:

- **common** - a legáltalánosabb szövegek, amiket rendszerint az összes oldal/komponens betölt, valamint erre a namespace-re fallback-el az `i18next` ha nem sikerült betöltenie egy másik namespace-t, vagy nem határoztuk meg, hogy melyik namespace-ből szeretnénk egy string-et kiragadni
- **auth** - a hitelesítési oldalakon használt szövegek (belépés és regisztráció)
- **home** - a főoldalon használt szövegek
- **contact** - a kapcsolatfelvételi űrlapon használt szövegek
- **docs** - a dokumentációs oldalakon használt szövegek
- **privacy** - az adatvédelmi nyilatkozat oldalon használt szövegek
- **terms** - a felhasználási feltételek oldalon használt szövegek
- **dashboard** - az ügyfélkapuban használt szövegek
- **errors** - a bizonyos hibakódoknak megfelelő hibaüzenetek

A könnyebb lokalizáció, valamint az újrahaználhatóság érdekében az API soha nem térít vissza egy szöveget, hanem mindig egy szöveges kulcsot, ami egy hibakódot jelképez. A frontend-en belül ezeket a hibakódokat az **errors** namespace alatt tároljuk, illetve innen olvassuk ki őket.

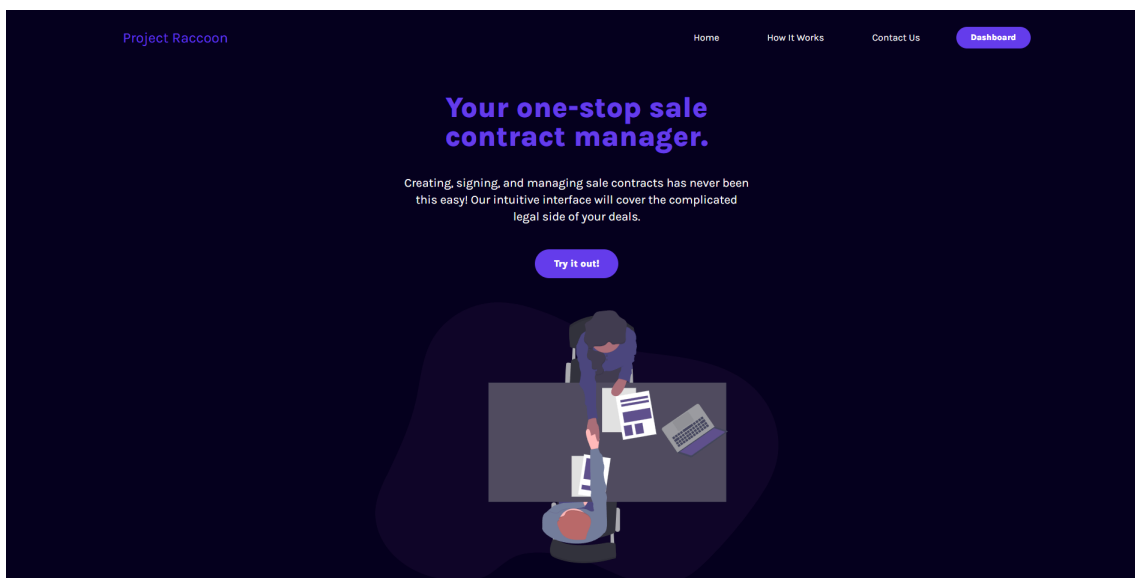
A `next-i18next` csomag biztosít számunkra egy `useTranslation` hook-ot, aminek megadhatjuk azt, hogy milyen namespace-eket szeretnénk egy komponensen belül használni. Ez a hook egy `t` nevű függvényt térít vissza, amiben megadhatjuk, milyen kulcsú szöveget szeretnénk lekérni a lokalizációs fájlokból. A kulcsokat a következő formában adjuk meg:

5.1. kódrészlet. A kulcsok formátuma

```
<namespace>:<key>.<subkey1>.<subkey2>. . .
```

5.8. Akadálymentesítés és mobil optimalizáció

A Project Raccoon-t úgy terveztem meg, hogy a lehető legtöbb felhasználó által kényelmesen használható legyen. Éppen ezért a szoftver felhasználói felületét témázható módon terveztem meg, pontosabban szólva implementáltam egy sötét témát is. A sötét téma egyik előnye, hogy kevésbé fárasztja a szemet, valamint az OLED kijelzők esetében kevesebb energiát fogyaszt, így a telefonok akkumulátorára is kevésbé lesz terhelő hatással. A sötét téma bekapcsolásához egy kapcsolót helyeztünk el, rendszerint a nyelvválasztó mellett.



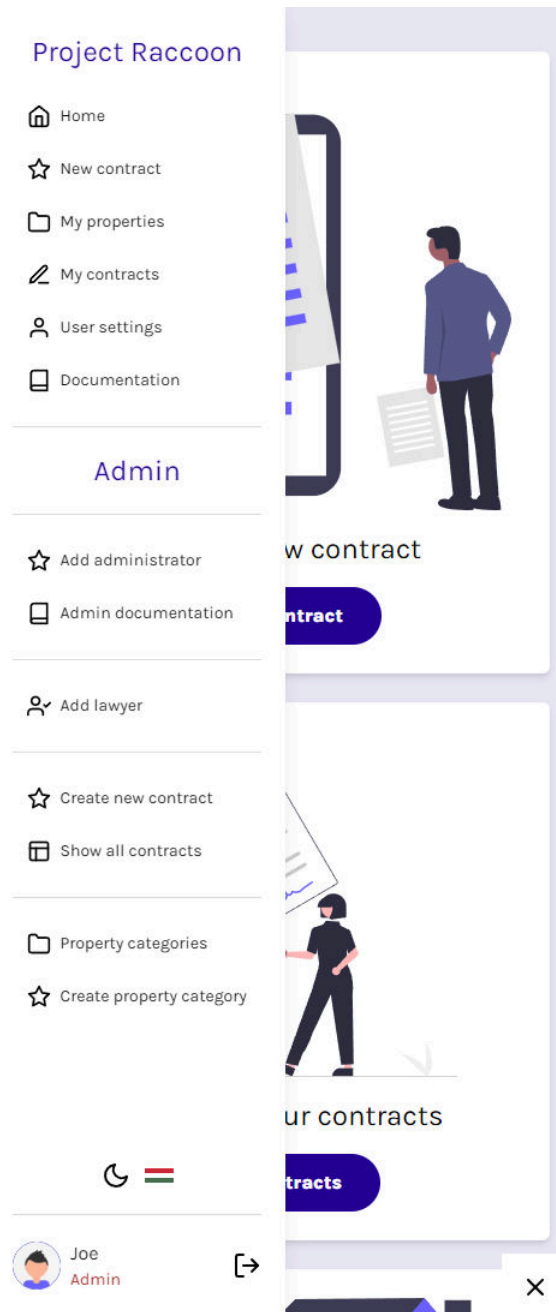
5.5. ábra. A szoftver kezdőlapja sötét témában

A szoftvert ugyanakkor igyekeztem olyanok számára is könnyebben elérhetővé tenni, akik inkább billentyűzetet használnak, mint például a mozgássérültek. Ezeknek a felhasználóknak biztosítunk egy „Skip to content” nevű gombot, ami a Tab billentyű lenyomásával elérhetővé válik és az Enter billentyű lenyomásával egyből a legfontosabb tartalomhoz fog ugrani a tab index. Ezután minden egyes Tab billentyű lenyomásával a következő elemre fog ugrani a felhasználó, így megkönnyítve a billentyűzettel való navigációt [16].

A Project Raccoon-t szerettem volna mobilbaráttá is tenni, így a felhasználói felületet mobiltelefonokra is optimalizáltam. A mobilos felhasználói felület kialakításában nagy segítségünkre volt a Tailwind mobile-first megközelítése. Ez azt jelenti, hogy a szoftver

felhasználói felületét úgy terveztem meg, hogy az elsődleges célom a mobilos felhasználói élmény legyen, majd ezt követően jöhetnek a nagyobb kijelzőkön való optimalizációk.

A mobilos felhasználói felület keretén belül a kétszlopos elrendezést felváltja az egyoszlopos, valamint a szoftver funkciói, a navigációs menü, valamint a nyelv- és téma-választó egy hamburger menübe került. Azt, hogy a hamburger menü meg legyen-e nyitva egy `useState` hook-al követtem nyomon.

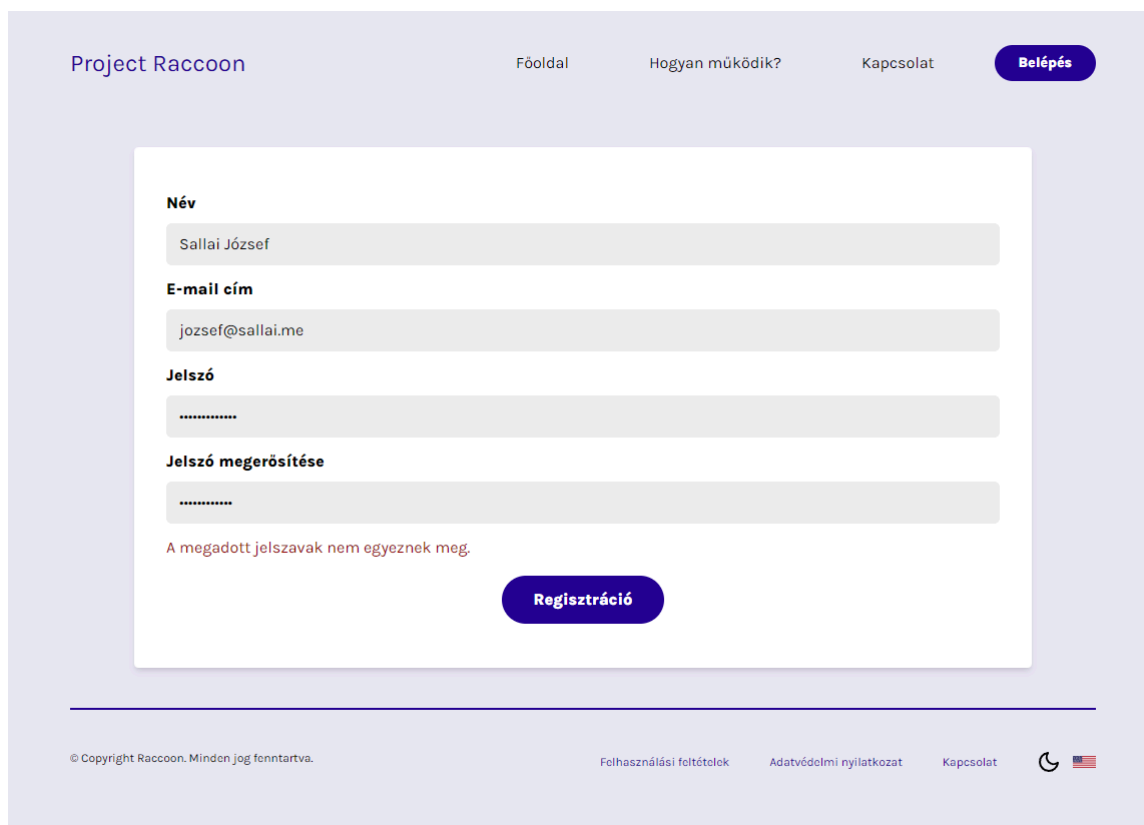


5.6. ábra. Megnyitott hamburger menü az ügyfélkapuban

6. fejezet

A rendszer gyakorlati működése

A Project Raccoon szoftver elérési címét megnyitva megjelenik a szoftver landing page-je, ami rövid áttekintést nyújt a látogatónak a szoftver főbb funkcionálisáról. Innen a call-to-action gombra vagy a fejléc jobb felső sarkában levő gombra kattintva a látogató át lesz irányítva a szoftver regisztrációs űrlapjára.



The image shows a web browser window displaying the Project Raccoon registration page. The page has a light purple header with the 'Project Raccoon' logo on the left and navigation links 'Főoldal', 'Hogyan működik?', 'Kapcsolat', and a 'Belépés' button on the right. The main content area is a white registration form with the following fields: 'Név' (Name) with the value 'Sallai József', 'E-mail cím' (Email address) with the value 'jozsef@sallai.me', 'Jelszó' (Password) with masked characters, and 'Jelszó megerősítése' (Confirm password) with masked characters. Below these fields is a red error message: 'A megadott jelszavak nem egyeznek meg.' (The provided passwords do not match). At the bottom of the form is a dark blue 'Regisztráció' (Registration) button. The footer of the page contains copyright information '© Copyright Raccoon. Minden jog fenntartva.', links to 'Felhasználási feltételek', 'Adatvédelmi nyilatkozat', and 'Kapcsolat', along with a refresh icon and a flag icon.

6.1. ábra. A regisztrációs űrlap

Sikeres regisztráció és belépés után a felhasználó át lesz irányítva az ügyfélkapuba, azon belül pedig a személyes információk módosítását szolgáló űrlapra. A felhasználó meg kell adja a személyes adatait ahhoz, hogy szerződések lebonyolítását tudja véghezvinni.

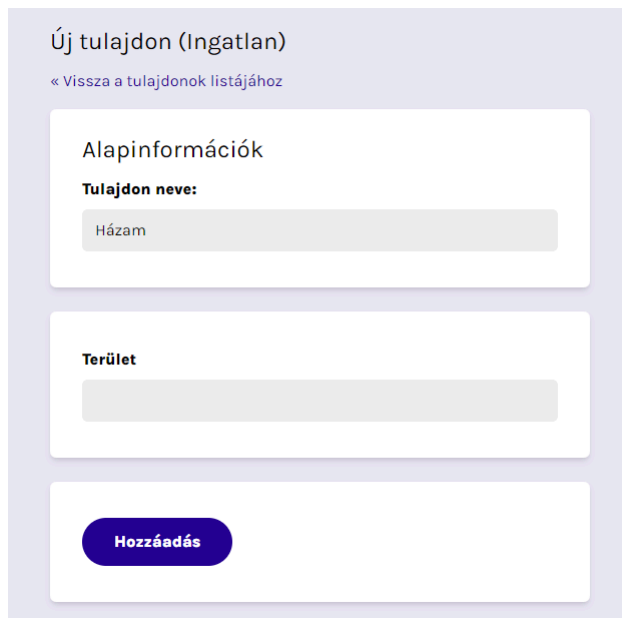
6.2. ábra. A felhasználói beállítások oldala első belépés után

A beállítások mentése után a felhasználó át lesz irányítva az ügyfélkapu főoldalára, ahol a platform főbb funkcióiról kap gyors elérési útvonalakat.

A bal oldali menüben levő **Tulajdonaim** menüelemre kattintva, a felhasználó megtekintheti a szoftveren belül elérhető tulajdonkategóriákat.

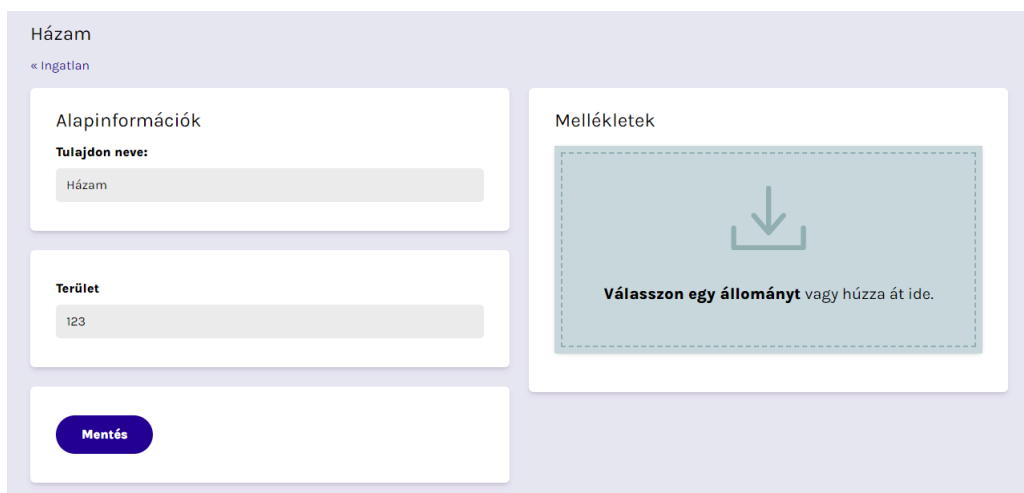
6.3. ábra. Tulajdonkategóriák listája

Egy tulajdonkategóriát kiválasztva a felhasználó megtekintheti azon tulajdonait, amik a kiválasztott kategóriához tartoznak. A kategória neve mellett lévő **Új** gombra kattintva a felhasználó létrehozhat egy új tulajdont. Az űrlapban egy nevet kell adnia a tulajdonnak, ami alapján azt a későbbiekben azonosítani tudja, valamint ki kell töltenie a tulajdonkategóriához tartozó mezőket.



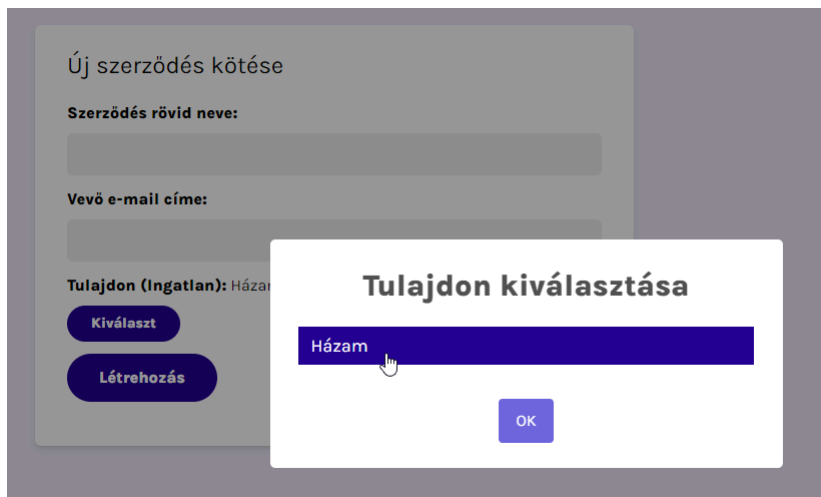
6.4. ábra. Tulajdon létrehozása űrlap

A tulajdon létrehozása után a felhasználónak lehetősége van módosítani annak adatait, illetve dokumentumokat feltölteni a tulajdonhoz. A dokumentumokat fel tudja úgy tölteni, hogy az oldal jobb oldalán levő dobozba húzza az állományt vagy egyszerűen rákattint és kiválasztja a megjelenő ablakból a csatolni kívánt dokumentumot.



6.5. ábra. Tulajdon részletei oldal

Az oldalsávban levő **Új szerződés kötése** menüelemre kattintva létre lehet hozni egy új adásvételi szerződést. A megjelenő űrlapon meg kell adni egy tetszőleges nevet, amit minden fél látni fog, a vevő e-mail címét, valamint a ki kell választani a tulajdont, amire a szerződés vonatkozik. A tulajdon kiválasztását egy pop-up ablak segíti elő.



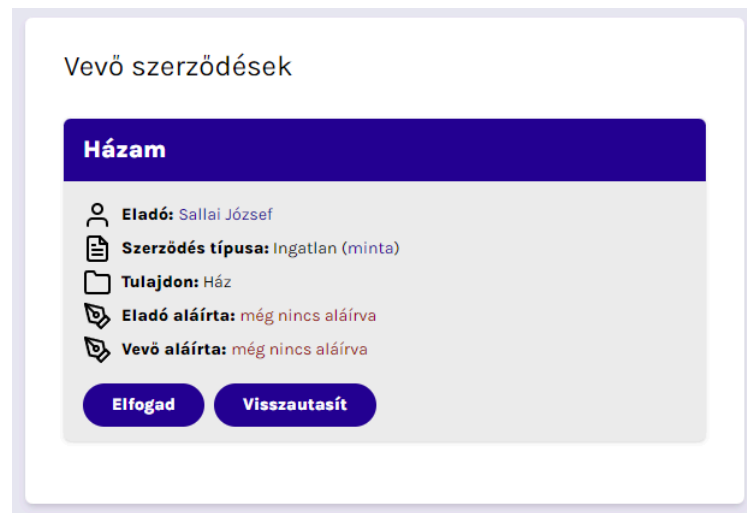
6.6. ábra. Tulajdon kiválasztása a szerződés létrehozásának oldalán

Miután létrejött a szerződés, meg kell várni, amíg a vevő elfogadja azt. A szerződést létrehozó felhasználó ilyenkor át lesz irányítva a szerződések listájának oldalára, ahol megtekintheti az újonnan létrehozott szerződés állapotát.



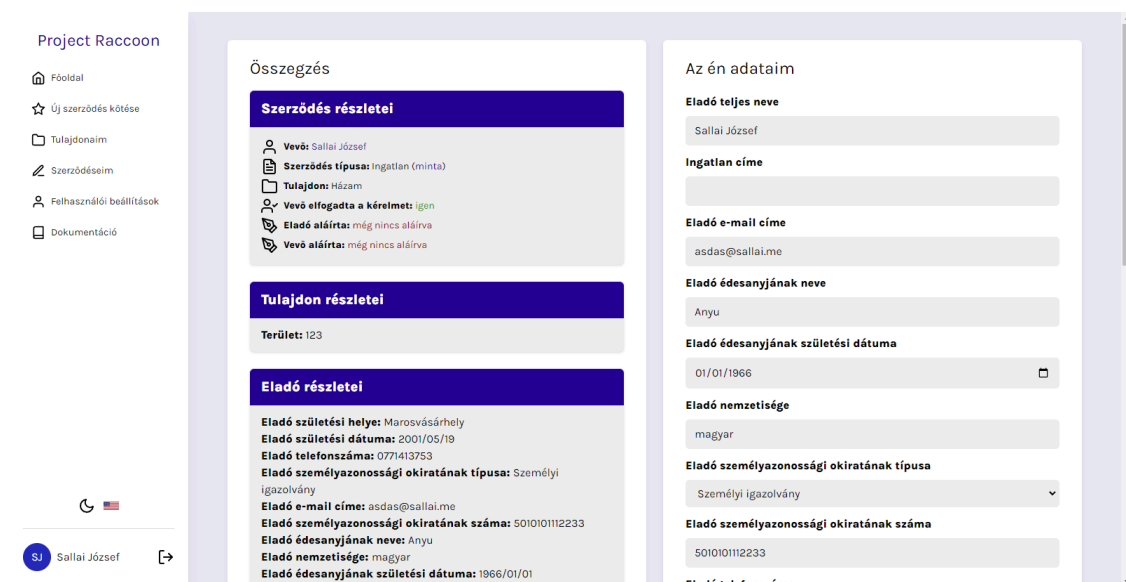
6.7. ábra. Eladói szerződések egy új szerződés létrehozása után

A vevő szemszögéből nézve a szerződések listájában egy új szerződés fog megjelenni a vevő szerződések listájában, ahol két gomb fog megjelenni: az első gombbal a vevő elfogadja a szerződést, a másodikkal pedig elutasítja azt.



6.8. ábra. Vevő szerződések egy függőben levő szerződéssel

A szerződés címére való rákattintással megtekinthető a szerződéshez kapcsolódó összes információ, beleértve az eladó személyes adatait, a vevő személyes adatait (amennyiben az elfogadta a szerződést), a tulajdon adatait, valamint egyéb, a szerződés által kért információkat.



6.9. ábra. Egy szerződés részleteinek oldala

A szerződés oldalán mindkét fél megtekintheti a szerződés tárgyát képező adatokat, módosíthatja az egyes adatokat, valamint csatolmányokat tölthet fel és ügyvédet vagy tanúkat hívhat meg a szerződéshez. Az ügyvédek olyan felhasználók, akiket az alkalmazás üzemeltetői hitelesítettek és a szerződésekhez hozzáadva ügyvédi státuszt kapnak. Amennyiben egy szerződéshez hozzá van adva egy ügyvéd vagy egy tanú, addig nem véglegesíthető a szerződés amíg ők is alá nem írták azt.

A mellékletek csatolása a tulajdonokhoz hasonló módon működik. Az eladó, a fent említett műveletek mellett képes arra is, hogy a szerződést teljes mértékben törölje.

Amennyiben egy fél alá szeretné írni a szerződést, a szoftver két lehetőséget nyújt számára egy pop-up ablakon keresztül: az egyik lehetőség, hogy manuálisan, az aláírást egy dobozba rajzolva írja alá, a másik pedig, hogy simán a saját nevével történjen meg az aláírás.



6.10. ábra. Szerződés aláírása

Miután minden fél aláírta a szerződést, a szerződés véglegesítve lesz és minden fél számára elérhető lesz a szerződés PDF formátumban. A kitöltött szerződés PDF másolatát a szerződés oldalán levő **Letöltés** gombra kattintva lehet elérni, a felhasználói műveletek dobozában.



6.11. ábra. Egy véglegesített szerződés aláírásai és műveletei

Összefoglaló

6.1. Következtetések

Dolgozatomban igyekeztem utánajárni annak, hogy milyen megoldások léteznek a tulajdonok gyors és hatékony kezelésére, az adásvételi szerződések megkötésére és véglegesítésére, valamint az eladók és vevők közötti kommunikációra. Véleményem szerint egy ilyen rendszer megvalósítása csak egy lépés a jogi és közigazgatási folyamatok teljes digitalizációja felé, amelynek eredményeképpen egyszerűsíteni lehet úgy az állampolgárok, mint a kormányzati szervek munkáját.

A dolgozatomban bemutatott rendszer ugyanakkor bebizonyítja azt is, hogy a projekt által használt technológiákkal könnyedén lehet megvalósítani egy komplex rendszert, amit egyszerűen és olcsón lehet üzembe helyezni akár serverless architektúrán is. Ennek előnye az, hogy nem kell stresszelni a szoftver skálázhatóságát illetően, valamint a használat függvényében kell fizetni.

Egy optimista jövőkép szerint a Project Raccoon egy olyan rendszerré válik, ami több ország kormánya által akkreditált, így a felhasználók egy hivatalos és biztonságos rendszeren keresztül fogják majd tudni az adásvételi szerződéseiket lebonyolítani.

6.2. Továbbfejlesztési lehetőségek

A Project Raccoon fejlesztése során igyekeztünk nagy figyelmet fordítani a legkisebb részletekre is, azonban még mindig vannak olyan területek, amelyek esetleges továbbfejlesztésre szorulhatnak.

- Tartalmi szempontból a szerződéseket lehetne lokalizálni is, mivel jelenleg csak a felhasználói felület van többnyelvűsítve. A szerződések lokalizálásával több felhasználónak is lehetőséget tudnánk biztosítani a rendszer használatára, országtól függetlenül.
- Biztonsági szempontból be lehetne építeni az alkalmazásba egy kétlépcsős azonosítást, amivel megakadályozhatjuk, hogy illetéktelen személyek hozzáférjenek az egyes felhasználók fiókjaihoz. Ez a funkció leginkább olyan felhasználók számára lehetne hasznos, akik nem fordítanak kellő figyelmet a jelszavuk biztonságára.
- Ugyancsak biztonsági szempontból lehetne implementálni egy CAPTCHA rendszert is a robotok elleni védelem érdekében.
- Akadálymentesítési szempontból több olyan funkciót lehetne implementálni, ami elősegíti a platform használatát mindenki számára, mint például a kontrasztos té-

mák implementálása, valamint a betűméret megváltoztatásának biztosítása a felhasználók számára.

GitHub projekt linkje

A projekt forráskódja a következő linken érhető el: <https://github.com/InfraPatch/raccoon>.

Ábrák jegyzéke

2.1. Dropbox Sign (HelloSign) főoldala	14
2.2. PandaDoc főoldala	15
2.3. Singaporean Network Trade Platform főoldala	16
4.1. A szoftver architektúrája	26
4.2. Box komponens	30
4.3. Oszlopok és dobozok	30
4.4. Toast komponensek	32
4.5. Üzenetküldés	36
4.6. Szerződés létrehozása	37
4.7. API service-ek	38
5.1. A chat gomb a szerződés oldalán	44
5.2. A chat ablak néhány üzenettel	44
5.3. Konzol kimenet a chat szerver eseményeiről	45
5.4. A nyelvválasztó a landing page alján	47
5.5. A szoftver kezdőlapja sötét témában	48
5.6. Megnyitott hamburger menü az ügyfélkapuban	49
6.1. A regisztrációs űrlap	50
6.2. A felhasználói beállítások oldala első belépés után	51
6.3. Tulajdonkategóriák listája	51
6.4. Tulajdon létrehozása űrlap	52
6.5. Tulajdon részletei oldal	52
6.6. Tulajdon kiválasztása a szerződés létrehozásának oldalán	53
6.7. Eladói szerződések egy új szerződés létrehozása után	53
6.8. Vevő szerződések egy függőben levő szerződéssel	54
6.9. Egy szerződés részleteinek oldala	54
6.10. Szerződés aláírása	55
6.11. Egy véglegesített szerződés aláírásai és műveletei	56

Táblázatok jegyzéke

3.1. Felhasználói követelmények	20
3.2. Termék követelmények	23
3.3. Szervezési követelmények	24
3.4. Külső követelmények	24
4.1. API válasz mezői	28
4.2. User modell	33
4.3. PersonalIdentifierType enum típus	34
4.4. FilledContract modell	34
4.5. FilledContractOption modell	35
4.6. ChatMessage modell	36

Irodalomjegyzék

- [1] MDN Web Docs. Websockets API - Web APIs. https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API.
- [2] e-Estonia. e-Residency - e-Estonia. <https://e-estonia.com/solutions/e-identity/e-residency/>.
- [3] e-Estonia. Story - e-Estonia. <https://e-estonia.com/story/>.
- [4] Josh Hicks. Is Tailwind CSS Worth It? *Medium*, 2022.
- [5] Vercel Inc. and contributors. Architecture: Supported Browsers - Next.js Documentation. <https://nextjs.org/docs/architecture/supported-browsers>.
- [6] Vercel Inc. and contributors. Building Your Application: Rendering - Next.js Documentation. <https://nextjs.org/docs/pages/building-your-application/rendering/>.
- [7] Vercel Inc. and contributors. Building Your Application: Routing - Next.js Documentation. <https://nextjs.org/docs/pages/building-your-application/routing>.
- [8] Vercel Inc. and contributors. Routing: Internationalization - Next.js Documentation. <https://nextjs.org/docs/pages/building-your-application/routing/internationalization>.
- [9] Faraz Kelhini. Axios vs. fetch(): Which is best for making http requests. *LogRocket Blog*, 'un, 2022.
- [10] Singapore NTP. About NTP. <https://www.ntp.gov.sg/public/faqs/introduction-to-ntp/about-the-ntp>.
- [11] PandaDoc. Sales Contract Template. <https://www.pandadoc.com/sales-contract-template/>.
- [12] Dropbox Sign. Dropbox Sign Templates. <https://www.hellosign.com/features/templates>.
- [13] React Core Team and contributors. Writing Markup with JSX - React Documentation. <https://react.dev/learn/writing-markup-with-jsx>.

- [14] Socket.IO Team and contributors. Socket.IO Documentation. <https://socket.io/docs/v4/>.
- [15] TypeORM Team and contributors. Decorator Reference - TypeORM Documentation. <https://typeorm.io/decorator-reference>.
- [16] North Carolina State University. Skip to Main Content - IT Accessibility. <https://accessibility.oit.ncsu.edu/it-accessibility-at-nc-state/developers/accessibility-handbook/mouse-and-keyboard-events/skip-to-main-content/>.
- [17] Vincent Voyer and contributors. vvo/iron-session README. <https://github.com/vvo/iron-session#readme>.