

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



SAPIENTIA
ERDÉLYI MAGYAR
TUDOMÁNYEGYETEM

Project Raccoon - Szerződéskötések és ingatlanértékesítések
korszerűsítése internetes platformon keresztül

DIPLOMADOLGOZAT

Témavezető:
Győrfi Ágnes, Tanársegéd
Kátai Zoltán, Egyetemi docens

Végzős hallgató:
Derzsi Dániel

2023

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



**UNIVERSITATEA
SAPIENTIA**

Proiect Raccoon - Gestionarea unei platforme pentru
eficientizarea finalizării de contracte și a vânzărilor de proprietăți
prin intermediul internetului

LUCRARE DE DIPLOMĂ

Coordonator științific:

Győrfi Ágnes, Asistent universitar
Kátai Zoltán, Conferențiar universitar

Absolvent:

Derzsi Dániel

2023

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



SAPIENTIA
HUNGARIAN UNIVERSITY
OF TRANSYLVANIA

Project Raccoon - Management of a Platform to Streamline
Contract Creation and Property Sales Through the Web

BACHELOR THESIS

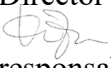
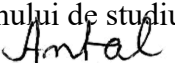


Scientific advisor:

Győrfi Ágnes, Assistant lecturer
Kátai Zoltán, Associate professor

Student:

Derzsi Dániel

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. Kátai Zoltán Îndrumător: Györfi Ágnes | Candidat: Derzsi Dániel Anul absolvirii: 2023 | |
| <p>a) Tema lucrării de licență: Proiect Raccoon - Gestionarea unei platforme pentru eficientizarea finalizării de contracte și a vânzărilor de proprietăți prin intermediul internetului</p> <p>b) Problemele principale tratate:</p> <ul style="list-style-type: none"> - Studiu bibliografic privind dezvoltarea interfețelor de administrator pentru o aplicație web folosind React și TailwindCSS. - Studiu bibliografic privind instalarea în viața reală a site-urilor NextJS folosind Docker Compose și/sau tehnologii serverless. - Studiu bibliografic privind semnalizarea digitală a documentelor PDF și atașarea de documente la fișiere PDF. - Dezvoltarea și planificarea unui sistem de șabloane folosind facilitățile oferite de platforma Google Docs. - Instalarea proiectului în viața reală. Măsurări de latență, resursele utilizate și analiza costurilor. <p>c) Desene obligatorii:</p> <ul style="list-style-type: none"> - Diagrame de proiectare pentru aplicația software realizată. - Diagrame de implementare și instalare în condiții reale de viață. <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 - Arhiva de instalare finalizată, bazată pe sistemul Docker Compose. <p>e) Bibliografia recomandată:</p> <ul style="list-style-type: none"> - Deploying Next.JS - https://nextjs.org/docs/pages/building-your-application/deploying - Docker Compose file version 3 reference - https://docs.docker.com/compose/compose-file/compose-file-v3/ <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: 20.06.2022 Termen de predare: 02.07.2023</p> | | |
| Semnătura Director Departament  Semnătura responsabilului programului de studii  | Semnătura coordonatorului  Semnătura candidatului  | |

Declarație

Subsemnatul/a DERESI DANIEL, 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 Sapientia 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, TÂRGU MUREȘ
Data: 26.06.2023

Absolvent

Semnătura Daniel.....

Kivonat

A mai rohanó világban az ingatlanügyletekben részt vevő magánszemélyek és vállalkozások gyakran jelentős kihívásokkal szembesülnek az ingatlan- és tulajdonadásvételi szerződések elkészítésekor. Ezen szerződések létrehozása az ingatlanügyletek szerves részét képezi, összetett jogi szabályok böngészésével és aprólékos dokumentáció írását köteleztetve. Szükség van egy olyan intuitív eszköz kitalálására, amely enyhíti ezeket a bonyolult folyamatok végigjárását, és megkönnyíti a jogszabályoknak megfelelő, pontos szerződések létrehozását.

Dolgozatom témája kettős. Először is, meg kívánom teremteni egy olyan online platform alapjait, amely lehetővé teszi a szerződések automatikus létrehozását a biztonságos ingatlanértékesítések megkönnyítése érdekében. Ez magában foglalja a szükséges szoftverarchitektúra megtervezését, egy könnyen kezelhető adminisztrációs felület létrehozását, valamint PDF dokumentumok szó szerinti generálását és kitöltését az adminisztrátorok által létrehozott sablonok segítségével. Másodszor, szándékomban áll dokumentálni a végső kitelepítés különböző szakaszait (és lehetőségeit), valós körülmények között mérve.

A javasolt weboldal átfogó platformként szolgál, barátságos felületet kínálva a felhasználóknak a saját igényeikre szabott, jogilag teljes bizonyító erejű szerződések létrehozására. Azáltal, hogy nincs szükség kiterjedt jogi ismeretekre vagy költséges ügyvédek fizetésére, a szerződések létrehozásához szükséges idő és erőfeszítés jelentősen csökken.

Összehasonlításképpen, világszerte hasonló szoftvermegoldásokat vezetett be számos kormány az adásvételi szerződésekkel kapcsolatos kihívások kezelésére. Ezeket a kormányok által támogatott törekvéseket azonban gyakran jelentősen hátráltatják a meggon-
dolatlan tervezési döntéseik és az eredendő homályosságuk. Platformunk célja, hogy a felhasználók számára olyan egyedi leegyszerűsített élményt nyújtson, amely felülmúlja a meglévő kormányzati kezdeményezéseket.

A szakdolgozat elején a javasolt szoftveres megoldás létrehozásához alkalmazott technológiákat tárgyaljuk. Ezt követően leírjuk, hogy mit fog a szoftver szolgáltatni, és hogy milyen teljesítményt várunk el tőle egy szoftverkövetelményspecifikáció részeként. A lehetséges szoftverarchitektúra és telepítési bemutatását követően részletesen bemutatjuk a platform leendő felhasználói felületét. Végezetül bemutatunk néhány mérést a szoftverarchitektúra tervezési szakaszában hozott döntéseink racionalizálása érdekében.

Kulcsszavak: *Tulajdonszerződések, Dokumentumsablonok, PDF dokumentumgenerálás, Dockeres kitelepítés, Serverless technológiák.*

Rezumat

În ziua de azi, persoanele fizice și juridice angajate în tranzacții imobiliare se confruntă adesea cu provocări semnificative atunci când redactează contracte de vânzare de proprietăți. Crearea acestor contracte de vânzare a proprietății este partea integrală a tranzacțiilor imobiliare, implicând legalități complexe și necesitatea unei documentații meticuloasă. Există o nevoie pentru un instrument intuitiv care să atenueze aceste complexități și să faciliteze crearea unor contracte precise și conforme cu legea.

Scopul tezei mele este dublu. În primul rând, intenționez să creez bazele unei platforme online care să permită crearea automată de contracte pentru a facilita vânzarea sigură a proprietăților. Aceasta include planificarea unei arhitecturii software necesare, crearea unei interfețe de administrare ușor de utilizat, precum și generarea și completarea propriu-zisă a documentelor PDF, folosind șabloanele create de către administratori. În al doilea rând, intenționez să documentez diferitele etape (și posibilități) ale instalării finale, măsurate în condiții reale.

Site-ul web propus servește ca o platformă cuprinzătoare, oferind utilizatorilor o interfață prietenoasă pentru a genera contracte obligatorii din punct de vedere juridic, adaptate la cerințele lor specifice. Prin eliminarea necesității unor cunoștințe juridice extinse sau a asistenței unor avocați costisitori, timpul și efortul necesar pentru a crea un contract sunt mult reduse.

Comparativ, guvernele din întreaga lume au implementat soluții software asemănătoare pentru a aborda provocările asociate cu contractele de vânzare de proprietăți. Cu toate acestea, aceste eforturi susținute de guverne sunt adesea dezavantajate în mod semnificativ de obscuritatea lor percepută. Această platformă își propune să ofere utilizatorilor o experiență unică și simplificată, care depășește inițiativele guvernamentale existente.

La începutul tezei vom discuta diferitele tehnologii utilizate pentru a crea soluția software propusă. Ulterior, vom descrie ceea ce va face software-ul și cum se așteaptă să funcționeze ca parte a unei specificații a cerințelor software (*SRS*). O interfață de utilizator prospectivă pentru platformă va fi detaliată în urma propunerii unei arhitecturi software. În cele din urmă, vor fi prezentate câteva măsurători pentru a raționaliza deciziile luate în timpul etapelor de planificare a arhitecturii software.

Cuvinte cheie: *Contracte de proprietate, Șabloane, Generarea de documente PDF, Instalare folosind Docker, Tehnologii serverless.*

Abstract

In today's fast-paced world, individuals and businesses engaged in real estate transactions often encounter significant challenges when drafting property sale contracts. The creation of these property sale contracts is an integral part of real estate transactions, involving complex legalities and meticulous documentation. There is a pressing need for an intuitive tool that can alleviate these complexities and facilitate the creation of accurate and legally compliant contracts.

The aim of my thesis is two-fold. Firstly, I intend to create the basis of an online platform to allow the automatic creation of contracts to facilitate secure property sales. This includes the creation and planification of the necessary software architecture, the creation of an easy to use administration interface, as well as the verbatim generation and filling out of PDF documents using templates created by the administrators. Secondly, I intend to document the various stages (and possibilities) of the final deployment, measured within real-world conditions.

The proposed website serves as a comprehensive platform, offering users a friendly interface to generate legally binding contracts tailored to their specific requirements. By eliminating the need for extensive legal knowledge or the assistance of costly attorneys, the time and effort required to create a contract is greatly reduced.

Comparatively, governments worldwide have implemented similar software solutions to address the challenges associated with property sale contracts. However, these government-backed endeavours are often significantly handicapped by their perceived obscurity and their convoluted design choices. This platform aims to provide users with a unique and simplified experience that surpasses existing government initiatives.

In the beginning of the thesis we will discuss the various technologies employed to create the proposed software solution. Afterwards, we will describe what the software will do and how it will be expected to perform as part of a Software Requirements Specification. A prospective user interface for the platform will be detailed following the proposal of a potential software architecture and deployment layout. Finally, some measurements will be presented to rationalize the decisions taken during the planning stages of the software architecture.

Keywords: *Property contracts, Document templates, PDF document generation, Docker deployment, Serverless technologies.*

Tartalomjegyzék

| | |
|--------------------------------------|-----------|
| 1. Bevezető | 11 |
| 2. Elméleti megalapozás | 13 |
| 2.1. Használt szoftverek, könyvtárak | 13 |
| 2.1.1. Next.js | 13 |
| 2.1.2. Node.js | 15 |
| 2.1.3. React | 16 |
| 2.1.4. Tailwind CSS | 16 |
| 2.1.5. TypeScript | 19 |
| 2.1.6. Docker és Docker Desktop | 20 |
| 2.1.7. Cloudflare R2 | 21 |
| 2.1.8. Google Cloud Storage | 22 |
| 2.1.9. TypeORM | 22 |
| 2.1.10. MariaDB | 23 |
| 2.1.11. PostgreSQL | 24 |
| 2.1.12. Google Docs | 25 |
| 2.1.13. MDX | 25 |
| 2.1.14. Iron Session | 27 |
| 2.1.15. bcrypt.js | 27 |
| 2.2. Piacelemzés | 28 |
| 2.2.1. Zillow | 29 |
| 2.2.2. LawDepot | 30 |
| 2.2.3. Kormányzati kezdeményezések | 31 |
| 3. A rendszer specifikációja | 32 |
| 3.1. Felhasználói követelmények | 32 |
| 3.1.1. Vendég felhasználók | 32 |
| 3.1.2. Hitelesített felhasználók | 34 |
| 3.1.3. Ügyvéd felhasználók | 37 |

| | |
|--|-----------|
| 3.1.4. Rendszergazda felhasználók | 37 |
| 3.2. Rendszerkövetelmények | 39 |
| 3.2.1. Funkcionális követelmények | 39 |
| 3.2.2. Nem funkcionális követelmények | 46 |
| 4. Tervezés | 49 |
| 4.1. A szoftver architektúrája | 49 |
| 4.2. A projekt mappaszerkezete | 51 |
| 4.3. Kliens-szerver architektúra | 54 |
| 4.4. Serverless architektúra | 59 |
| 4.5. MVC - Modell-Nézet-Vezérlő | 61 |
| 4.5.1. Modellek | 61 |
| 4.5.2. Nézetek | 62 |
| 4.5.3. Vezérlők | 63 |
| 4.6. Szolgáltatási réteg | 64 |
| 4.7. Szerződéssablonok létrehozása | 65 |
| 4.8. Végleges szerződések kigenerálása | 67 |
| 4.9. Felhasználói felület tervezése | 68 |
| 4.10. Felhasználói use case-ek | 71 |
| 5. Kivitelezés | 75 |
| 6. Mérések | 81 |
| 6.1. Késleltetési teszt: hidegindítás esetén | 82 |
| 6.2. Késleltetési teszt: melegindítás esetén | 83 |
| 6.3. Telepítési idő frissítéskor | 84 |
| 6.4. Költségelemzés havonta: alacsony használati előrejelzés | 85 |
| 6.5. Költségelemzés havonta: magas használati előrejelzés | 86 |
| Összefoglaló | 87 |
| Ábrák jegyzéke | 89 |
| Táblázatok jegyzéke | 90 |
| Irodalomjegyzék | 92 |

1. fejezet

Bevezető

Az évek során a technológia robbanásszerű fejlődésen ment keresztül, számos területen átalakította életünket. Számítalan ügy, amelyet azelőtt személyesen kellett lebonyolítani, átkerült az online világba, az internetre. Ez az online ügyintézési folyamat a modern technológia által kínált előnyök és lehetőségek következménye, és gyökeresen megváltoztatta mindennapi életünk dinamikáját.

Az interneten történő ügyintézés számos előnnyel és kényelemmel jár a mai ember számára. Az egyik legnyilvánvalóbb az idő megtakarítása. Ezelőtt, ha valamilyen ügyet szeretnénk volna intézni, a megfelelő hivatalt személyesen kellett felkeresnünk, amely órákba vagy egyes esetekben akár napokba is telhetett. Nem kell időpontot előzetesen személyesen egyeztetni, nem kell hosszú sorokat kivárni. Nem szükséges szabadnapot kivenni a munkahelyről az ügyek elintézése végett, illetve bármikor, akár este is végezhetjük ügyeink intézését, a technológia révén most már otthonunk kényelméből tudjuk ügyesbajos dolgainkat intézni, vagy bárhol, ahol internet-hozzáférésünk van - legyen ez akár a mindennapi tömegközlekedés során mobiltelefonunk segítségével. Egyszerűen bejelentkezünk az illetékes weboldalra, kitöltjük a szükséges űrlapokat vagy elvégezzük a szükséges tranzakciókat, és néhány kattintással elintézzük az ügyet. Könnyen belátható tehát, hogy az online ügyintézés igénybevételével sokkal nagyobb szabadságfokot és rugalmat élvezhetünk saját időbeosztásunk tekintetében.

Ezzel párhuzamosan nemcsak számunkra, hanem az intézmények és vállalatok számára is hasznosak ezen internetes platformok: csökkenthetik az adminisztrációs terheket, a munkaerőigényt (hiszen az ügyfelek maguk tudják elvégezni ügyeiket), gyorsabb kiszolgálást tesz lehetővé a különböző folyamatok automatizálásával, az adatok online tárolásával és kezelésével.

Az online ügyintézés rohamos terjedése a technológiai fejlődés mértékét tükrözi, és az életünk szerves részévé vált. Az internet lehetőséget ad számunkra, hogy gyorsan és

hatékonyan intézzük az ügyeinket, miközben megtakarítjuk az értékes időnket és növeljük a kényelmünket.

Néhány ország annyira élenjár a technológiai fejlődésben, hogy állampolgáraik képesek majdnem minden ügyeiket az internet segítségével intézni. Sajnos kevés ország tartozik jelenleg ezen országok közé. Magyarország vagy Románia esetében számos ügyet csak személyesen lehet továbbra is lebonyolítani. Ilyen kategóriába tartoznak az adásvételi szerződések. Amennyiben az említett országokban szeretnénk eladni vagy épp venni ingóságot vagy épp ingatlant, számos jogi lépéseken kell keresztül menni, amelyek esetenként napokat is felöllelhetnek. Nem vitatott tehát, hogy egy hosszadalmas és stresszes procedúráról van szó.

Az üzleti világban a szerződések elengedhetetlenek az adásvételi folyamatok lebonyolításához és a jogi biztonság fenntartásához. Azonban a hagyományos módszerekkel járó papír alapú szerződéskötés és az azt követő adminisztráció gyakran időigényes és bonyolult folyamatot jelentett. Azonban most, a Project Raccoon platform bemutatkozásával, egy teljesen új megközelítés jelentkezik az adásvételi szerződések kezelésében.

A Project Raccoon egy forradalmi platformot kínál, amely lehetővé teszi minden szerződéses fél számára, hogy egyszerűen és hatékonyan kezelje és aláírja az adásvételi szerződéseket. Ez a platform megszabadítja őket az időrabló és körülményes adminisztratív feladatoktól, és lehetővé teszi a jogi formulációk gyors és könnyed elvégzését.

A Raccoon platformja új dimenziót ad az adásvételi szerződések generálásához és kezeléséhez. Az alkalmazás egyszerűen használható felhasználói felülete és intuitív navigációja révén minden fél - akár eladók, vevők, vállalkozások vagy magánszemélyek - egyszerűen és hatékonyan tudja kitölteni és aláírni a szerződéseket.

A platform úttörő módon automatizálja a szerződések létrehozását a kormány által kiadott hivatalos sablonok alapján. Ez jelentősen megkönnyíti a szerződéskötés folyamatát, hiszen a feleknek már nincs szükségük szakértői jogi ismeretekre vagy ügyvédi segítségre ahhoz, hogy létrehozzanak egy pontos és jogilag érvényes szerződést. A Raccoon platformja gondoskodik arról, hogy minden szükséges jogi információ és kitöltendő mező rendelkezésre álljon, így a felhasználók egyszerűen csak kitölthetik az adott részeket, és a platform automatikusan generálja a szerződést.

Az adásvételi szerződések digitalizálása és azok kezelésének egyszerűsítése egyértelműen egy hatalmas előrelépést jelent a Project Raccoon platformja által kínált innovatív megoldással. A Raccoon platformja segítségével a szerződések elkészítése és aláírása könnyebbé válik, a folyamat pedig gyorsabbá és hatékonyabbá válik minden érintett számára.

2. fejezet

Elméleti megalapozás

2.1. Használt szoftverek, könyvtárak

A dolgozat ezen részében megvizsgáljuk azokat a szoftvereket, amelyeket felhasználtunk a platform készítéséhez. A felhasznált szoftvereket két részre bontjuk: a végtermék által használt függőségek, valamint a tervezés és a fejlesztés során használt egyéb szoftverek.

2.1.1. Next.js

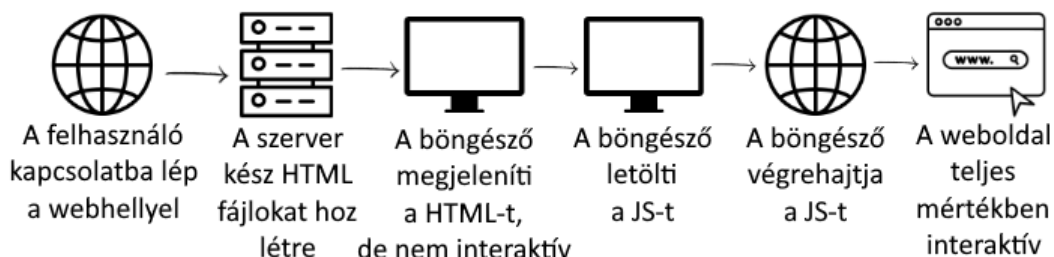


2.1. ábra. A Next.js logója

A Next.js (2.1. ábra) egy nagy teljesítményű [1] JavaScript keretrendszer (*framework*), amelyet modern webes alkalmazások készítésére használnak. A felhasználói felületek építésére széles körben elfogadott JavaScript-könyvtár, mely a React rugalmasságát szerveroldali rendereléssel és több optimalizálási technikával ötvözi. Elsősorban optimalizált és skálázható webes alkalmazások létrehozására használják, mivel akár szerver nélküli (*serverless*) függvényekre is képes felosztani a fejlesztők által írt kódokat.

A Next.js tökéletesen illeszkedik a mi felhasználási esetünkhöz, mivel nemcsak azt teszi lehetővé, hogy a teljes webalkalmazás JavaScriptben íródjon, hanem mindezt teljesítményhatékony módon teszi, lehetőséget adva arra, hogy a frontendet és a backendet egy egységes projekt részeként valósítsuk meg.

A Next.js-t különösen hasznossá teszi az a tény, hogy az alkalmazás igényeitől függően zökkenőmentesen tud váltani a szerveroldali renderelés (SSR) és a kliensoldali renderelés (CSR) között. Ez számunkra nagyszerű, hiszen így mindkét megközelítés legjobb



2.2. ábra. A szerveroldali renderelés magyarázata

tulajdonságait élvezhetjük a kezdeti oldalbetöltési sebesség és az interaktivitás szempontjából. A szerveroldali renderelés és a kliensoldali renderelés ötvözése elősegíti a projekt keresőmotor optimalizálását (*search engine optimization*) is.

A 2.2. ábrán látható a két megközelítés ötvözése. Legelső lépésben a felhasználó kapcsolatba lép a webhellyel, és elkéri a weboldalt jellemző HTML fájlt a szervertől. A szerver egy kész HTML fájlt hoz létre, melyet visszaküld a felhasználónak. A felhasználó böngészője megjeleníti a HTML oldalt, amely még ebben a stádiumban nem interaktív, csupán előre legyártott tartalmakat jelenít meg. Mivel ezek az előre legyártott tartalmak nem szükségeltetik a JavaScript használatát, ezért az egyes keresőmotorok, mint például a Google Search vagy a Yandex Search, könnyedén indexelhetik a weboldalunkon található tartalmakat. Ez nagyszerű, hiszen nem minden keresőmotor támogatja a JavaScript végrehajtását. A böngésző a HTML megjelenítése után letölti a kliensoldali renderelést végrehajtó JavaScript modulokat a webszerverről, majd végrehajtja őket. A végeredmény egy olyan weboldal megjelenítése a felhasználó számára, amely előre elkészített tartalmakat is tartalmaz, viszont teljes mértékben interaktív abból a szempontból, hogy a weboldalon látható tartalmak bármikor frissíthetőek az egész weboldal újratöltése nélkül.

Emellett a Next.js kiváló fejlesztői élményt nyújt olyan funkciókkal, mint a *hot module replacement* (automatikusan megjeleníti a kód által renderelt eredményt, miközben a programozó írja a kódot), az automatikus kódfeosztás (*code splitting*) és a *serverless* telepítés beépített támogatása (tekintettel arra, hogy a *Vercel*¹ a Next.js projekt vezető fejlesztője). A kiterjedt ökoszisztéma és a közösségi támogatás tovább növelte a vonzerejét, így a Next.js meggyőző választás volt a *Project Raccoon* kivitelezésére. Tökéletes egyensúlyt képvisel a szolgáltatások mennyisége és az általunk elképzelt platform létrehozásához szükséges fejlesztési idő között.



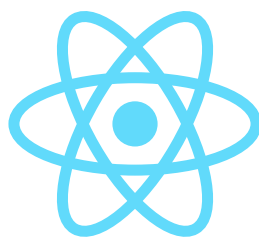
2.3. ábra. A Node.js logója

2.1.2. Node.js

A Node.js (2.3. ábra) egy nyílt forráskódú, szerveroldali JavaScript futtatási környezet, amely képes futtatni a Next.js szolgáltatásait. Lehetővé teszi a JavaScript kód böngészőn kívüli végrehajtását, így nem szükséges külön nyelvet választani a szerveroldali kódok megvalósítására. Tekintettel arra, hogy a szerveroldali kód így JavaScriptben is írható, a különféle modulok, például az adatmodellek újrafelhasználhatóakká válnak a kliens és a szerveroldal között, kevesebb kódDuplikációt eredményezve.

A Node.js egyik meggyőző aspektusa az eseményvezérelt, nem blokkoló (*non-blocking*) I/O modellje, amely lehetővé teszi az egyidejű kapcsolatok hatékony kezelését. Ez a tulajdonsága különösen alkalmassá teszi weboldalak készítésére, hiszen egyszerre több felhasználót kell kiszolgálnak. A Node.js emellett hatalmas ökoszisztémával rendelkezik, mely magába foglalja a fejlesztést egyszerűsítő könyvtárakat és keretrendszereket, így népszerű választás a szerveroldali alkalmazások és kliensoldali alkalmazások készítéséhez egyaránt.

Csomagkezelői, az npm és a yarn értékes eszközzé teszik a nagyméretű alkalmazások felépítéséhez. A Node.js sokoldalúságával, eseményalapú megközelítésével, Next.js támogatásával és közösségi támogatásával vonzó választást jelentett a *Project Raccoon* fejlesztése számára. Egy másik szoftver használatát is fontolóra vettük: a *Deno JavaScript Runtime*-t², egy másik JavaScript szerveroldali környezetet. Azonban sajnos ezt az ötletet elvetettük, mivel a választott keretrendszer, a Next.js támogatására még nem alkalmas a dolgozat írásakor.



2.4. ábra. A React logója

2.1.3. React

A React (2.4. ábra) egy felhasználói felületek építésére szolgáló JavaScript-könyvtár, amely elsősorban interaktív és újrafelhasználható UI-komponensek létrehozására összpontosít. A Reactot általában egyoldalas alkalmazások (*SPA-k*) és mobilalkalmazások készítésére használják. A React bevezeti a virtuális DOM fogalmát, amely lehetővé teszi a komponensek hatékony frissítését és renderelését, jobb teljesítményt eredményezve. A React komponensek a virtuális DOM-ban megfelelnek egy-egy igazi HTML elemnek a böngésző DOM-jában. Egy frissítés során a React felépíti a változtatások részfáját a virtuális DOM újraépítése során, és csupán azokat a HTML elemeket frissíti a böngésző DOM-jából, amelyek módosultak, optimalizálva a frissítéseket. A React komponensalapú architektúrát követ, lehetővé téve a fejlesztők számára, hogy moduláris és újrafelhasználható felhasználói felület elemeket hozzanak létre, ami könnyebb karbantartást jelent, a kód újrafelhasználhatóságát eredményezve. A React-ot használva a fejlesztők valós idejű frissítéseket láthatnak a fejlesztési folyamat során. Az erős közösségi támogatás, a komponensek újrafelhasználhatóságának, a hatékony megjelenítésnek illetve a kiterjedt dokumentáció, egyaránt hivatalos és nem hivatalos forrásokból, megerősítette döntésünket a Next.js keretrendszer és a React könyvtár használata mellett.

2.1.4. Tailwind CSS



2.5. ábra. A Tailwind CSS logója

A hagyományos webhelyprojektekben szoros kapcsolat van a CSS és a HTML között, mivel a CSS struktúrája a HTML-struktúrát tükrözi. [2] Általában a programozók egy-

¹A Vercel egy felhő szolgáltató, mely lehetővé teszi a fejlesztők számára a webes projektek egyszerű telepítését és skálázását *serverless* technológiákkal.

²A Deno egy biztonságos TypeScript futási környezet, amely a V8-ra, a Google JavaScript futtató-motorjára épül.

szerre írják a weboldalaik HTML és CSS részeit, és emiatt jön létre ez a szoros kapcsolat a HTML és a CSS szerkezete között (lásd 2.2. és 2.3. kódrészlet). Ez azt eredményezi, hogy a megírt CSS osztályok nem újrafelhasználhatóak, mindig mikor egy új HTML elemet hozunk létre, vele együtt egy új CSS osztályt is létre kell hoznunk. Ritkaság, hogy a programozók létrehozzák a HTML megírása előtt tudatosan a CSS osztályokat. Emiatt találták ki a Tailwind CSS-et (2.5. ábra), amely segítségével a CSS osztályok megírása a múlt mulatságává válik. A Tailwind CSS új paradigmát vezet be: saját CSS osztályok írása helyett a HTML tagokat előre megírt, általános CSS osztályok ötvöztetésével stílusozzuk (lásd 2.1. kódrészlet). A Tailwind CSS által nyújtott CSS osztályok annyira általánosak, hogy egyetlen osztály a legtöbb esetben csupán egy-egy CSS stílussal (értékpárral) egyezik meg. A Tailwind CSS csökkenti a függőséget a HTML és a CSS között, és kikerüli a komplikált CSS-szelektorok szükségességét.

```
<div class="max-w-sm rounded-lg border border-gray-200 bg-white p-6 shadow">
  <a href="https://meow.com">
    <h5 class="mb-2 text-2xl font-bold tracking-tight text-gray-900">Miert
      kelnek fel a cicak?</h5>
  </a>
  <p class="mb-3 font-normal text-gray-700">Mert megkattannak.</p>
  <a href="https://meow.com" class="inline-flex items-center rounded-lg
    bg-blue-700 px-3 py-2 text-center text-sm font-medium text-white
    hover:bg-blue-800 focus:outline-none focus:ring-4
    focus:ring-blue-300">Olvass tovább</a>
</div>
```

2.1. kódrészlet. Példa megvalósítása egy HTML komponensnek Tailwind CSS segítségével

```
<div class="card">
  <a href="https://meow.com" class="card-link">
    <h5 class="card-title">Miert kelnek fel a cicak?</h5>
  </a>
  <p class="card-description">Mert megkattannak.</p>
  <a href="https://meow.com" class="card-button">Olvass tovább</a>
</div>
```

2.2. kódrészlet. Példa megvalósítása egy HTML komponensnek sima HTML és CSS kombinációval: HTML komponens

2.3. kódrészlet. Példa megvalósítása egy HTML komponensnek sima HTML és CSS kombinációval: CSS komponens

```
.card {  
  max-width: 20rem;  
  border-radius: 0.5rem;  
  border-width: 1px;  
  border-color: #edf2f7;  
  background-color: #fff;  
  padding: 1.5rem;  
  box-shadow: 0 1px 3px 0 rgba(0, 0, 0, 0.1), 0 1px 2px 0 rgba(0, 0, 0, 0.06);  
}  
.card-title {  
  margin-bottom: 0.5rem;  
  font-size: 1.5rem;  
  font-weight: 700;  
  color: #1a202c;  
}  
.card-description {  
  margin-bottom: 0.75rem;  
  font-weight: 400;  
  color: #4a5568;  
}  
.card-button {  
  display: inline-flex;  
  align-items: center;  
  border-radius: 0.25rem;  
  background-color: #4299e1;  
  padding: 0.5rem;  
  text-align: center;  
  font-size: 0.875rem;  
  color: #fff;  
  text-decoration: none;  
}  
.card-button:hover {  
  background-color: #2b6cb0;  
}  
.card-button:focus {  
  outline: 2px solid transparent;  
}
```

2.1.5. TypeScript



2.6. ábra. A TypeScript logója

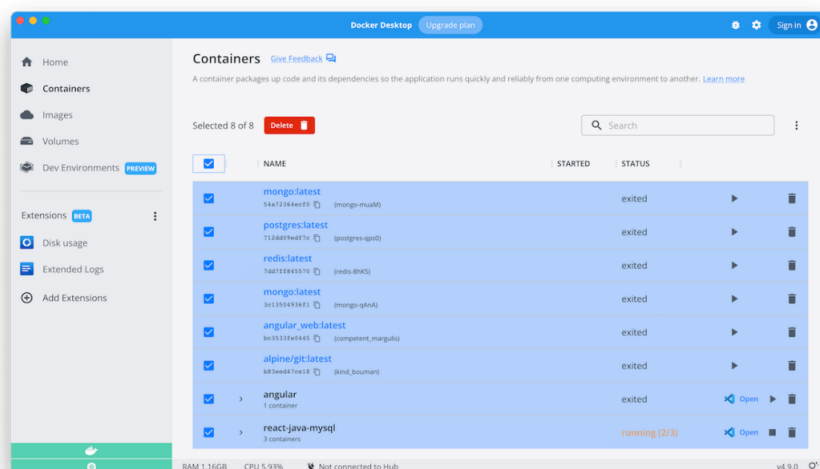
A TypeScript (2.6. ábra) a JavaScript statikusan típusos továbbbővítése, amely kibővíti statikus típusokkal bővíti ki a JavaScript nyelvet. A statikus típusok bevezetésével a TypeScript lehetővé teszi a fejlesztők számára, hogy a fejlesztés során minél hamarabb észrevegyék a hibákat, megőrizve a kód minőségét és javítva a végeredmény karbantartóságát. A TypeScript képes kijavítani a JavaScript projektekben fellelhető hibák akár 15%-át is, csupán a típusellenőrzések végett. [3]

Magába foglalja az objektumorientált programozást (*OOP*), lehetővé téve a fejlesztők számára, hogy interfészeket, osztályokat és modulokat definiáljanak, és felhasználják az objektumorientált programozásból az öröklődést, az egységbezárást és a polimorfizmust. A TypeScript ötvözése a Visual Studio Code integrált fejlesztői környezettel lehetővé teszi a kód automatikus kitöltését és refaktorálását, jelentősen elősegítve minket, fejlesztőket a tiszta és hibamentes kód megírásában.

Emellett olyan fejlett funkciókat is kínál, mint a generikusok és dekorátorok, amelyek kibővítik a kódabsztrakció és a kódbázis-szervezés lehetőségeit. A mi esetünkben a dekorátorokat teljes mértékben kihasználja a TypeORM rendszer, lehetővé téve az adatbázis modelleink leképezését tényleges adatbázis sémára. Mivel a TypeScript kompatibilitás a meglévő JavaScript-ben írt projektekkel, és sima JavaScriptre le is fordítható, ezért könnyen integrálható a Next.js keretrendszerbe is.

A TypeScript programozási nyelv önmagában nem futtatható a Node.js környezettel. Jelenleg csupán a Deno környezet engedélyezi a TypeScript tényleges láthatatlan futtatását, viszont a Deno környezet nem támogatja az általunk kiválasztott web keretrendszert, a Next.js-t. Mivel önmagában nem futtatható, a TypeScript nyelv nem csupán a megfelelő linterekkel és szintaxisfa elemző programokkal telepítendő, hanem a TypeScript kompilátorral is, amely a TypeScript forráskód állományokat képes átalakítani JavaScript forráskód állományokká.

2.1.6. Docker és Docker Desktop



2.7. ábra. A Docker Desktop kinézete

A Docker (2.7. ábra) egy nyílt forráskódú platform, amely megkönnyíti az alkalmazások telepítését, skálázását és kezelését elszigetelt konténerekben. A konténerek elindított Docker *image*eknek felelnek meg, amelyek mögött mindig egy virtuális fájlrendszer és futó folyamatok állnak. A Docker imagek olyan önálló, hordozható környezetek, amelyek egy alkalmazást és függőségeit - beleértve az operációs rendszert és a könyvtárakat - foglalják magukba.

Annak ellenére, hogy megosztják a gazdagép operációs rendszerének kernelét, a konténerek hasonló biztonsági szintet szolgáltatnak, mint a hagyományos virtuális gépek. [4] Az alkalmazás és a függőségei egységbe vannak zárva, csökkentve a konfliktusokat és biztosítva a környezet reprodukálhatóságát. Tekintettel arra, hogy a konténerek el vannak szigetelve a rendszer többi részétől. A konténerek kevesebb rendszererőforrást fogyasztanak, mint a tradicionális virtuális gépek és gyorsan elindulnak, a hardver hatékony kihasználását és az alkalmazások gyorsabb skálázódását eredményezve.

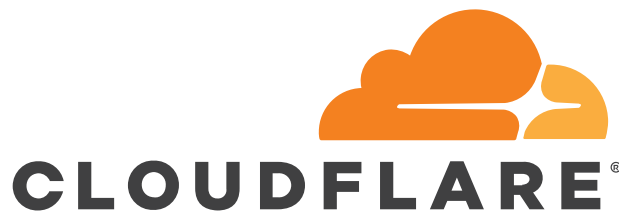
A Docker leegyszerűsíti a függőségek kezelését azáltal, hogy az alkalmazást az adott *runtime*-al, könyvtárakkal és függőségekkel együtt egy image-be csomagolja. Így a fejlesztőknek nem kell manuálisan konfigurálniuk és kezelniük a függőségeket minden egyes célrendszeren. A kompatibilitási problémákat ezáltal kiszűri a Docker és megbízhatóbbá teszi a telepítéseket.

A nagy webes alkalmazások kontextusában a Docker nagyon hasznos, mivel az olyan segédprogramok, mint a Docker Compose (mely a dolgozat írásakor már alapértelmezésként a Docker része) lehetővé teszik, hogy a konténerek konfigurációját Compose fájlként - rendszerleírófájlként mentse el. Ezek a Compose fájlok YAML "markup" nyelven íródnak,

melyek még azok számára is nagyon könnyen manipulálhatóak, akik nem rendelkeznek programozási ismeretekkel. A konténerek közötti kapcsolatokat ezekben a Compose fájlokban írhatjuk le.

Mind a Dockert, mind a Docker Compose-t széles körben használjuk a kitelepített környezetünkben és a fejlesztési környezetünkben is. A Docker Compose segítségével létrehozott szolgáltatásokat gyakran külön Docker Compose projektekbe csoportosítják, amelyek atomi módon a rendszerben esetleg meglévő más Docker Compose projektektől függetlenül indíthatók, újraindíthatók és leállíthatók, így a karbantartás gyerekjátékká válik.

2.1.7. Cloudflare R2



2.8. ábra. A Cloudflare logója

A Cloudflare R2 (2.8. ábra) a Cloudflare felhő alapú szolgáltató 2022-ben bevezetett új ajánlata, amely az Amazon S3 platformjának leváltására szolgál. A Cloudflare R2 célja, hogy az S3-hoz hasonló funkciókat biztosítson, lehetővé téve a felhasználók számára a fájlok és adatok tárolását és visszakeresését.

Paradigmaváltást vezet be azonban azzal, hogy megszünteti a régió megadásának szükségességét. Az Amazon S3 használatához mindenképp szükséges megadni annak a régióknak a nevét, ahol a feltöltött adatok élni fognak. Ehelyett a Cloudflare globális számítógép-hálózata automatikusan biztosítja, hogy a fájlok a kérést benyújtó felhasználóhoz legközelebbi adatközpontban elérhetőek legyenek. Az adattárolás ily módon decentralizált megközelítése jobb teljesítményt és kisebb késleltetést (*latency-t*) eredményez a végfelhasználók számára.

A Cloudflare R2-t különösen vonzóvá teszi a Cloudflare globális hálózatának a legközelebbi adatközpontból történő adatkiszolgálásra való kihasználása. A felhasználók a gyorsabb adathozzáférés és az optimalizált felhasználói élmény előnyeit élvezhetik.

Abban az esetben, ha részletesebb adattárolási törvények vannak érvényben, például az európai adatvédelmi törvények, amelyek előírják, hogy az illékony adatokat európai régiókon belül kell tárolni, ez az automatikus régió konfigurálható úgy, hogy csak egy adott joghatóságon belül engedélyezze a tárolást. [5] Mint ilyen, a Cloudflare R2 kompatibilis

a használati célunkkal, és megfelelő létesítményeket biztosít az Amazon S3-nál olcsóbb, biztonságosabb és gyorsabb adattárolásért.

2.1.8. Google Cloud Storage

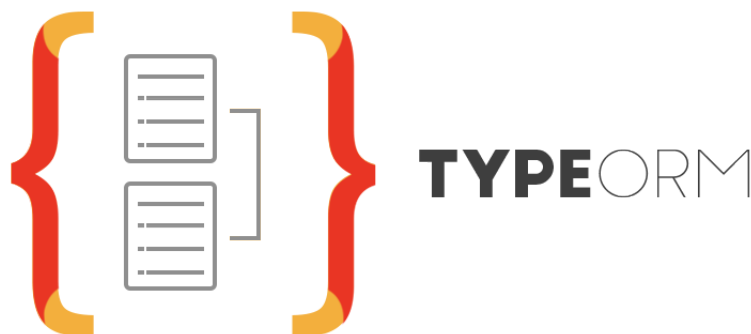


2.9. ábra. A Google Cloud logója

Hasonlóan az Amazon-hoz és a Cloudflare-hez, a Google is szolgáltat saját objektum- és fájl tároló rendszert: a Google Cloud Storage-t (2.9. ábra). A Google Cloud Storage hasonlít az S3 alapú megoldásokhoz, de egy teljesen más API használatát kéri, bár létezik egy már nem karbantartott S3-kompatibilis API-ja, amely a Google Cloud Storage XML API-ját használja. [6]

Mivel a Firebase keretén belül használható a Google Cloud Storage is, ezért akkor ajánlatos a Google Cloud Storage használata, mikor az egész projektünk Firebase alapú. A mi esetünkben nincs különösebb értelme, hogy Google Cloud Storage alapú fájl tárolást használjunk, viszont a lehetőség adott. Szerettük volna bebizonyítani azt a tényt, hogy a választott architektúra megengedi a különféle fájl tárolási stratégiák egyszerű implementálását és kicserélését, ezért a *Project Raccoon* képes Google Cloud Storage-ra is menteni a fájljait.

2.1.9. TypeORM



2.10. ábra. A TypeORM logója

A TypeORM (2.10. ábra) egy objektum-relációs leképező (*Object-Relational Mapping* - *ORM*) könyvtár a TypeScript számára, amelyet az adatbázisokkal való interakciók egy-

szerűsítésére terveztek. A TypeORM használatával leképezhetjük TypeScript osztályainkat adatbázis táblákká, lehetővé téve számunkra, hogy objektumorientált paradigmák segítségével dolgozzunk a relációs és nemrelációs adatbázisokkal egyaránt.

A TypeORM több adatbázis-rendszert is támogat, többek között a MySQL, PostgreSQL, SQLite és Microsoft SQL Server adatbázisokat is. Nyers SQL-lekérdezések írása helyett TypeScript szintaxissal írhatunk adatbázis-lekérdezéseket. Emellett a TypeORM olyan funkciókat is biztosít, mint például az entitáskapcsolatok (one-to-one, one-to-many, many-to-many), a validáció, tranzakciókezelés és automatikus adatmigráció. Jelen projektünkben a TypeORM rendszer segítségével építettük fel a modelleinket, és MySQL, illetve PostgreSQL adatbázist használva tároljuk az adatainkat.

2.1.10. MariaDB



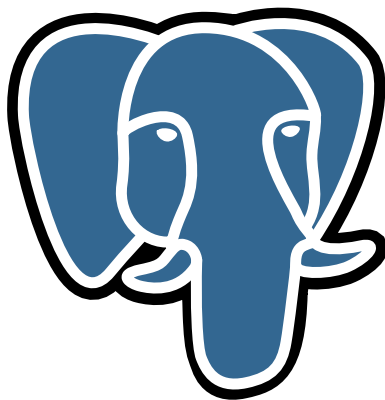
2.11. ábra. A MariaDB logója

A MariaDB (2.11. ábra) egy nyílt forráskódú relációs adatbázis-kezelő rendszer, amely az eredeti MySQL adatbázis-kezelőrendszer kódbázisán alapszik. A MariaDB-t készítő vállalatot, Monty Program AB-t, Michael „Monty” Widenius, a MySQL egyik eredeti fejlesztője alapította, miután az Oracle felvásárolta a MySQL vállalatot 2010-ben. Az új adatbázis-kezelő rendszert pedig második lánya után, Maria után nevezte el. A MariaDB alapítványt 2012 decemberében hozták létre, hogy elkerüljék a MariaDB vállalati felvásárlását, mint ami a MySQL-lel is történt. A MariaDB Alapítvány kezdeményezései a MariaDB folytonos fejlesztésének támogatása és a nyílt együttműködés globális kapcsolati pontjaként szolgálnak.

A MariaDB nagy hangsúlyt fektet az adatbázis-kezelés teljesítményére, skálázhatóságára és megbízhatóságára. Támogatja a szerver-kliens architektúrát, amely lehetővé teszi a többfelhasználós környezetek használatát, és számos funkciót biztosít az adatok hatékony tárolására, lekérdezésére és kezelésére. Számos kiterjesztéssel és bővített funkcióval rendelkezik, amelyek lehetővé teszik az adatok magasabb fokú biztonságát, a tranzakciók kezelését, a replikációt és a szerverkülső hozzáférést. Emellett számos eszközt és interfészt biztosít a fejlesztők és rendszergazdák számára, hogy könnyen dolgozhassanak az adatbázissal.

Mivel alapjául a MySQL szolgál, így a legtöbb esetben a MariaDB kompatibilis vele, ami azt jelenti, hogy a meglévő MySQL alkalmazások és kódok általában gond nélkül futtathatók a MariaDB környezetben is.

2.1.11. PostgreSQL



2.12. ábra. A PostgreSQL logója

A PostgreSQL (2.12) egy nyílt forráskódú relációs adatbázis-kezelő rendszer. Az elnevezése a „Postgres” (utalva a POSTGRES projektre, amely egy másik adatbázis-kezelő rendszer, az Ingres hibáit hivatott kijavítani, innen ered a projekt neve: *Post-Ingres*, azaz „Ingres utáni”) és a „SQL” (*Structured Query Language*, strukturált lekérdezési nyelv) szavak összetételéből származik. A PostgreSQL eredete 1986-ig nyúlik vissza, a Berkeley-i Kaliforniai Egyetem POSTGRES projektjének részeként, és több mint 35 évnyi aktív fejlesztéssel rendelkezik az alapplatformon. Azóta a világ egyik legnépszerűbb adatbázis-kezelő rendszerévé nőtte ki magát.

A PostgreSQL erős hangsúlyt fektet az adatintegritásra, a megbízhatóságra és adataink biztonságára. Támogatja az ACID (*atomicity*, azaz atomikus, *consistency*, azaz konzisztens, *isolation*, azaz elszigetelt és *durability*, azaz tartós) műveleteket, amelyek biztosítják az adatbázis tranzakcióinak megbízhatóságát és konzisztenciáját.

Képes kezelni a hagyományos relációs adatbázisok által nyújtott lehetőségeket, mint például a táblák és az SQL használata, és emellett további funkciókat és adattípusokat is kínál az objektumok tárolására és kezelésére. A PostgreSQL sok kiterjesztést és funkciót nyújt a felhasználók számára, mint például az indexelés, nézetek, tárolt eljárások, triggerek és sok más. Emellett a rendszer nagy skálázhatóságot biztosít, így nagy adathalmazokat is hatékonyan kezelhet.

A PostgreSQL széles körben használatos webes alkalmazásokban, vállalati rendszerekben és kutatási projekteken is. A fejlesztői közösség aktív és folyamatosan fejleszti a rendszert, biztosítva a stabilitást, biztonságot és a funkcionális bővítményeket.

2.1.12. Google Docs



2.13. ábra. A Google Docs logója

A Google Docs (avagy magyarul Google Dokumentumok) (2.13. ábra) egy ingyenes, felhőalapú dokumentumkezelő szolgáltatás, amelyet a Google fejlesztett ki és tárt a nyilvánosság elé 2006-ban. A Google lehetővé teszi, hogy online létrehozassunk dokumentumokat, szerkeszthessük ezeket, illetve megoszthassuk másokkal őket. Ezen dokumentumok között szerepelhet szöveges állomány, de szerkeszthetünk táblázatokat (Google Sheets, vagyis Google Táblázatok), létrehozhatunk prezentációkat (Google Slides, vagyis Google Diák), készíthetünk űrlapokat (Google Forms, vagyis Google Űrlapok) és egyéb fájlokat is.

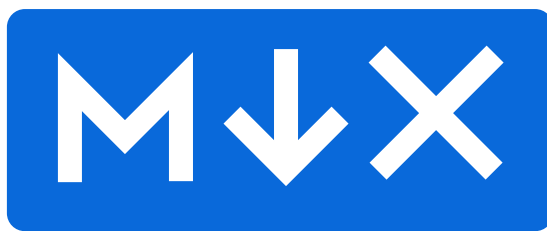
A Google Docs online kezelőfelülete nagyon hasonló a hagyományos szövegszerkesztőkhöz, de az előnye, hogy az adatokat felhőalapú tárhelyen tárolja, így a dokumentumaikhoz a felhasználók bárholnan és bármilyen eszközről (legyen ez akár számítógép, mobiltelefon vagy táblagép) hozzáférhetnek, csupán internetkapcsolattal kell rendelkezniük.

Másik nagy előnye, hogy a Google Docs lehetővé teszi a valós idejű együttműködést is. Ez azt jelenti, hogy több felhasználó egyszerre szerkesztheti ugyanazt a dokumentumot, és azonnal láthatják egymás változtatásait. Emellett lehetőség van a dokumentumok megosztására másokkal csak megtekintési joggal is. Egy másik hasznos funkciója a szerkesztési előzmények megtartása, így bármikor visszaállítható egy dokumentum az adott időpontban elmentett verziójára. Az adatok automatikusan mentésre kerülnek az adott felhasználó Google Drive-ján (a Google felhőalapú adattároló szolgáltatása), így nem kell aggódni az elveszett munka miatt.

A Google Docs összekapcsolható más Google szolgáltatásokkal is, például a már említett Google Drive-val, ahol az összes dokumentumot tárolni lehet, vagy a Google Classroommal, amely lehetővé teszi adott dokumentumok megosztását a virtuális osztályteremben lévő személyekkel.

2.1.13. MDX

Az MDX (Markdown eXtended) (2.14. ábra) egy formázási nyelv és fájlformátum, amely Markdown-alapú szöveges tartalomhoz kiterjesztett lehetőségeket biztosít.



2.14. ábra. Az MDX logója

A Markdown egy egyszerű és könnyen értelmezhető szöveges formázási nyelv, amely lehetővé teszi a formázott tartalom egyszerű és gyors létrehozását. Az MDX ennek a Markdownnak a szintaxisát használja alapul, de kiegészíti azzal, hogy lehetőséget ad React komponensek használatára a szövegben.

Az MDX fájlok kiterjesztése általában „.mdx”. Tartalmukat egyszerű szöveges formátumban írják, de lehetőséget adnak a JSX (JavaScript XML) szintaxis használatára is. Ez azt jelenti, hogy az MDX fájlokban React komponenseket is be lehet illeszteni, amelyek összetett funkciókat adhatnak a tartalomhoz. Ez különösen hasznos lehet például dokumentációk, blogbejegyzések stb. esetén, ahol az egyszerű szöveg mellett egyúttal reaktív, interaktív vagy testreszabott elemeket is szeretnénk hozzáadni (gondolhatunk itt például interaktív diagrammokra).

A Next-MDX egy olyan függvénykönyvtár, amely a Next.js keretrendszerhez készült és a React-alapú MDX tartalmak kezelését és renderelését hivatott kezelni - a Next.js és az MDX kombinációjával teszi lehetővé a dinamikus és interaktív MDX tartalmak könnyű kezelését a Next.js projektjeinkben.

A 2.4. kódrészletben egy egyszerű .mdx fájl megvalósítására adunk példát.

```
# Cím
## Alcím
Az MDX Markdown lehetővé teszi a szöveg formázását is, például:
  - *Dőlt betű* vagy _Dőlt betű_
  - **Félkövér betű** vagy __Félkövér betű__
  - 'Kódstílusú szöveg'
MDX Markdownban könnyedén hozhatunk létre hiperlinkeket is, például:
  [Google](https://www.google.com).
Az MDX Markdown lehetővé teszi React komponensek beágyazását is:
<MyComponent prop1="érték1" prop2="érték2" />
A képek beillesztése is lehetséges:
![Alt szöveg a képhez](/path/to/image.jpg)
```

2.4. kódrészlet. Példa megvalósítása egy .mdx fájlnak

2.1.14. Iron Session

Az Iron Session egy munkamenet-kezelő (*session manager*) middleware³ webes alkalmazásoknak. A munkamenetek biztonságos kezelésére és tárolására használandó. A munkamenetadatok tárolására digitálisan aláírt és titkosított cookie-kat használ („pecsétek” - *seals*). A digitális aláírásokhoz az Iron nevű kriptográfiai könyvtárat használja. A pecséteket csak a szerver tudja dekódolni. Nem léteznek munkamenet azonosítók, így a munkamenetek a szerver szempontjából állapot nélküliek (*stateless-ek*). Az Iron Session biztosítja, hogy a munkamenetadatok hamisíthatatlanok legyenek, és védi a munkamenetek állapotainak integritását.

A hagyományos alkalmazásokban, például a régi PHP alkalmazásokban a munkamenetadatok nyomon követése munkamenetazonosítók segítségével történt. A munkamenet adatait egy relációs adatbázisba mentették. Amikor a felhasználó egy munkamenethez szeretne volna hozzáférni kérését, a munkamenetazonosítót mutatta be a szervernek, amely ezután megkereste a munkamenetet az adatbázisában.

Az Iron Session használatával egyáltalán nem szükséges a munkamenetadatokat az adatbázisban tárolni. Ez a megközelítés csökkenti a szerver memóriaigényét, és lehetővé teszi, hogy a munkamenetek akkor is fennmaradjanak, ha a szerver újraindul, vagy esetleg elveszti kapcsolatát az adatbázissal. Minden munkamenetadat a kliensoldalon kerül elmentésre és tárolásra. Ezeket a munkamenetadatokat egy olyan titkos kulcs segítségével írja alá az Iron Session rendszer, amelyet csak a webszolgáltatás ismer. Így csak a kiszolgáló webservert képes érvényes munkameneteket létrehozni, amelyeket teljes egészében átküld a kliensnek. Ezeket a digitálisan aláírt munkafolyamatokat majd később a kliens a szervernek elküldi. A bemutatott érvénytelen munkameneteket a rendszer elveti.

Mivel a munkamenetadatokat nem tárolja a webservert, a munkamenet keresésének ideje jelentősen csökken. Az Iron Session használatával elkerülhető az adatbázis-szoftverrel történő hiábavaló kommunikáció, ami az alkalmazás érzékelhető gyorsulását eredményezheti. Tekintettel e megközelítés használatának előnyeire, úgy döntöttünk, hogy az Iron Session-t a *Project Raccoon* részeként alkalmazzuk.

2.1.15. bcrypt.js

A bcrypt.js egy kriptográfiai hashelési függvénykönyvtár, amely a bcrypt hashelési algoritmus implementációját nyújtja JavaScriptben. A bcrypt algoritmus [7] egy (akaratlagosan) lassú és erős jelszóhashelő, amely tervezésétől fogva célja a jelszavak biztonságos átalakítása.

³A middleware-ek lehetővé teszik egy webes lekérés válaszáinak módosítását az eredeti kérés átírásával, átirányításával, a kérés vagy a válasz fejlécének módosításával vagy közvetlen válaszadással.

Amikor a felhasználó elküld egy jelszót, a jelszó hashelésre kerül, majd tárolásra az adott alkalmazás adatbázisában. A mi alkalmazásunkon belül a felhasználóink és adminisztrátoraink jelszavait hasheljük. Később, amikor a felhasználó hitelesíteni akarja magát, a webszervernek össze kell hasonlítania a felhasználó által beírt jelszót az adatbázisban tárolt hash értékkel, hogy megegyezik-e. A jelszavak „megbízható” hashfüggvényekkel való hashelésére biztonsági szempontból van szükség, hogy egy esetleges adatszivárgás vagy hackertámadás esetén a feltört jelszavak továbbra is érthetetlenek, használhatatlannak maradjanak a támadók számára.

A bcrypt.js segítségével a fejlesztők JavaScript alkalmazásokban használhatják a bcrypt algoritmust a felhasználók jelszavainak hashelésére és ellenőrzésére. A bcrypt hashelése egy egyirányú folyamat, ami azt jelenti, hogy a hashelt értékből nem lehet visszafejteni az eredeti jelszót. Ez a tulajdonság növeli a jelszavak biztonságát, mivel még akkor sem lenne lehetséges a jelszavak helyreállítása, ha a hash értékek valahogy illetéktelen kezekbe kerülnének.

A bcrypt-et Niels Provos és David Mazières tervezte a Blowfish titkosító alapján, nevét is innen kapta: a „b”-t a Blowfishtől, a „crypt” részét pedig az UNIX jelszórendszere által használt hash-függvény nevéből.

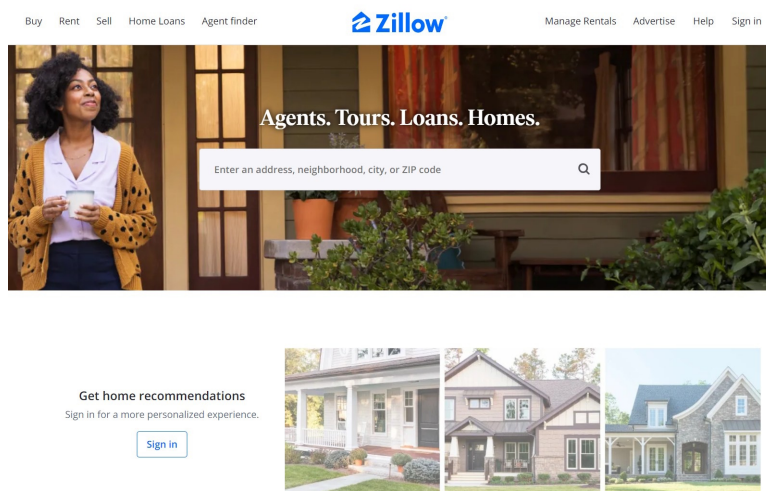
A bcrypt algoritmus emellett beépített biztonsági funkciókat is tartalmaz, például sózást (*salting*), ami tovább erősíti a hashelést. A sózás a jelszóhoz egy véletlenszerű értéket (sót) ad hozzá, amelyet a hashelés során használ. Ez megnehezíti a támadóknak a prekalkulált támadásokat, mivel minden jelszóhoz különböző só-t használ. Így lehetetlenné válik a *rainbow table* típusú támadások kivitelezése. Emellett bevezet egy nehézségi fokozatot is, egy szabadon állítható költséget (*work factor*, *cost* vagy munkatényező) a függvénynek, amely extra biztonsági réteget nyújt: a jelszavak hashelésének ideje exponenciálisan növekszik a költségek növekedésével. Ezáltal a nyers erő módszere (*brute force attacks*) és a szivárványtáblás támadások (*rainbow table attacks*) ellen is erős védelmet nyújt.

2.2. Piacelemzés

A Project Raccoon elkészítése előtt elvégeztünk egy részletes piacelemzést. Utána jártunk annak, hogy milyen hasonló szoftverek léteznek már használatban a piacon, és annak is utána néztünk, hogy az akadémiában milyen hasonló kezdeményezések léteznek.

A való világban számos olyan platform létezik, mely célja az ingatlanügyletek és a szerződéskötések egyszerűsítése. Ezek közül a platformok közül néhány hasonlít a *Project Raccoon* által szolgáltatott Íme néhány példa:

2.2.1. Zillow



2.15. ábra. A Zillow főoldala

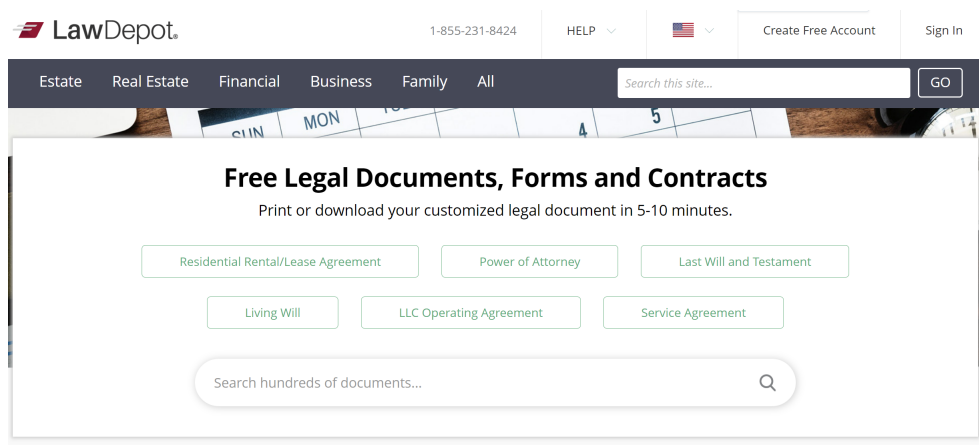
A Zillow egy online ingatlanpiac, amely ingatlanok vásárlásával, eladásával és bérletével kapcsolatos szolgáltatásokat nyújt. Lényegében a Zillow számos ingatlanhirdetés adatbázisaként szolgál, összesítve az ingatlanügynököktől és maguktól a lakástulajdonosoktól származó információkat. A 2.15 ábrán látható, hogy a felhasználók házakat, lakásokat és más típusú ingatlanokat kereshetnek keresőszűrők segítségével, mint például a hely, az árkategória, a hálószobák és a fürdőszobák száma.

A Zillow részletes információkat nyújt az ingatlanokról, beleértve az ingatlanok leírásait, kiváló minőségű fényképeket, az ingatlanok előzményeit és a becsült értékeit. Ez a rengeteg információ segít a felhasználóknak tájékozott döntéseket hozni ingatlanok vásárlásakor vagy bérletekor.

Ha azonban a témát ingatlanszerződésekre tereljük, a Zillow egyáltalán nem segíti elő a szerződések létrehozását vagy aláírását. Ehelyett olyan platformként működik, amely összeköti a vevőket, az eladókat és az ingatlanszakértőket. Amint a vevő és az eladó megállapodásra jut az ingatlanvásárlás vagy bérbeadás feltételeiről, általában egy ingatlanügynök vagy egy ügyvéd szolgáltatásait veszik igénybe a szerződés kidolgozásához. Az olyan platformok, mint a Zillow, gyakran nem teszik lehetővé a tényleges szerződések létrehozását. Az ingatlanokat könnyű létrehozni és számon tartani, de nem olyan egyszerű létrehozni a jogi keretet, sem megvalósítani a követelményeket egy ilyen rendszerhez. Itt ragyoghat a Project Raccoon. Célunk a szerződések létrehozása, generálása és sablonozása, nem pedig az ingatlanok nyilvános kilistázása.

Nem csupán a Zillow az egyetlen ilyen ingatlanvásárlást megkönnyítő platform. Léteznek más platformok is, mint például a Redfin vagy a Trulia, viszont ezek a platformok annyira hasonlítanak a Zillow-ra, hogy nem éri meg külön tárgyalni őket.

2.2.2. LawDepot



2.16. ábra. A LawDepot főoldala

A LawDepot egy online platform, amely jogi dokumentum sablonok széles skáláját nyújtja magánszemélyek és vállalkozások számára. Kényelmes megoldást kínál személyre szabott jogi dokumentumok, többek között ingatlanszerződések létrehozására is.

A LawDepot elsődleges célja a jogi szerződések létrehozásának egyszerűsítése azáltal, hogy testreszabható sablonokat biztosít, amelyek a felhasználók egyedi igényeihez igazíthatóak. A felhasználók a LawDepot weboldalán keresztül hozzáférhetnek ezekhez a sablonokhoz, és könnyen kitölthetik a szükséges információkat a jogilag kötelező erejű dokumentumok létrehozásához.

Ami az ingatlanszerződéseket illeti, a LawDepot különböző sablonokat kínál, amelyek az ingatlanügyletek különböző aspektusait fedik le. A felhasználók sajátos körülményeik és követelményeik alapján választhatják ki a megfelelő sablont. A LawDepot legnagyobb hibája, hogy bár megkönnyíti a legális dokumentumsablonok kitöltését, nem teremt kapcsolatot az adó és a vevő felek között. A kigenerált dokumentumok csupán egy ember által láthatóak, és csupán egy emberre érvényesek. Ehhez képest a *Project Raccoon* kapcsolatot teremt az adó és a vevő felek között, és az adásvételi folyamatokat elejétől végéig követhetővé teszi. Ezeknek a folyamatoknak a végeredménye digitális aláírásra is kerül, ami növeli jogi értéküket, amit a LawDepot képtelen elérni.

2.2.3. Kormányzati kezdeményezések

Egyes kormányok online portálokat hoztak létre, amelyek megkönnyítik az ingatlantranzakciókat és a szerződéskötéseket. Ezek a kezdeményezések országonként és joghatóságoként eltérőek.

Szingapúrban például a kormány bevezette az „e-Stamping” néven ismert ingatlantranzakciós platformot. Az „e-Stamping” platform digitalizálja az ingatlannal kapcsolatos dokumentumok bélyegzési folyamatát, ami a szingapúri szerződésalkotási és hitelítési folyamat utolsó lépése. [8]

Az e-Stamping lehetővé teszi a felhasználók számára, hogy elektronikusan fizessenek bélyegilletéket, és igazolásokat állítsanak elő az ingatlannal kapcsolatos tranzakciókhoz, beleértve az adásvételi szerződéseket, bérleteket és jelzálogkölcsönöket. A platform biztonságos és kényelmes módot biztosít magánszemélyek és vállalkozások számára a bélyegzési folyamat befejezésére, fizikai dokumentumok benyújtása vagy kézi bélyegzés nélkül.

Ezek a megközelítések sajnos nagyon korlátozottak, és csak egyetlen konkrét problémát oldanak meg. Ezzel szemben arra törekszünk a *Project Raccoon* keretén belül, hogy olyan megoldást alkossunk, amely számos helyzetben újrahasznosítható, nem csak ingatlanügyleteknél. Tekintettel arra, hogy az adminisztrátorok tetszés szerint hozhatnak létre újabb és újabb szerződéseket, semmi sem akadályozza meg az adminisztrátorainkat abban, hogy a *Project Raccoon*-t más típusú szerződések kezelésére is kiterjesszék.

3. fejezet

A rendszer specifikációja

3.1. Felhasználói követelmények

A felhasználói követelmények leírják azokat a funkciókat, amelyeket a felhasználók elvárnak egy adott szolgáltatástól. Ezek a követelmények felvázolják a felhasználók igényeit és céljait, és alapul szolgálnak a későbbi szoftver megtervezéséhez és fejlesztéséhez. A rendszerünk felhasználói követelményeit négy részre bontom a továbbiakban:

1. Felhasználói követelmények vendég felhasználók számára,
2. Felhasználói követelmények hitelesített felhasználók számára,
3. Felhasználói követelmények ügyvéd felhasználók számára.
4. Felhasználói követelmények adminisztrátor felhasználók számára.

3.1.1. Vendég felhasználók

A vendég felhasználók olyan személyek, akik még nem hoztak létre fiókot vagy nem jelentkeztek be a rendszerbe.

3.1. táblázat. Felhasználói követelmények vendég felhasználók számára

| | |
|--------------------------|---|
| Hitelesítés | <ol style="list-style-type: none">1. A vendég felhasználóknak lehetőségük van egy fiók létrehozására, további funkciók eléréséhez.2. Ebben az esetben a regisztrációs folyamatnak intuitív-nak és egyszerűnek kell lennie, végigvezetve a felhasználókat a szükséges lépéseken.3. A felhasználók minél kevesebb adat megadásával már szeretnének regisztrálni.4. Ideális esetben captcha kitöltése nélkül szeretnének regisztrálni.5. Abban az esetben, hogyha már van fiókjuk, regisztrálás helyett meg kell legyen adva a lehetőség, hogy a már meglévő fiókjába lépjen be a felhasználó. |
| Kapcsolatfelvétel | <ol style="list-style-type: none">1. A felhasználók bármelyik időpillanatban használatba vehetnek egy kapcsolatfelvételi űrlapot, amellyel felveshetik a kapcsolatot a weboldal készítőivel.2. A felhasználók a weboldal készítőinek szeretnének egy e-mail címet is hagyni, hogy a weboldal készítői tudják felvenni a kapcsolatot a felhasználóval.3. A kapcsolatfelvételi űrlap elegendő helyet kell szolgáltatson ahhoz, hogy a felhasználók tudják elejétől végéig leírni a problémáikat, észrevételeiket vagy kéréseiket. |

| | |
|------------------|--|
| Kézikönyv | <ol style="list-style-type: none"> 1. A felhasználók bármelyik időpillanatban segítséget szeretnének kérni a weboldal felhasználásával kapcsolatban. 2. Mivel nem minden felhasználó szeretne emberi kommunikációval előre haladni a weboldal használatában, ezért meg kell adni a lehetőséget, hogy egy felhasználói kézikönyvet lapozgassanak. 3. A felhasználói kézikönyv leírásokat kell tartalmazzon a regisztrációról, a tulajdonok menedzsmentjéről és a szerződéskötés folyamatáról is. |
|------------------|--|

3.1.2. Hitelesített felhasználók

A hitelesített felhasználók olyan személyek, akik munkamenetéhez egy fiók van csatolva. Ez azt jelenti, hogy vagy beléptek egy meglévő fiókba, vagy regisztráltak egy új fiókot. A hitelesített felhasználók lehetnek rendes felhasználók, vagy akár adminisztrátor felhasználók is. A következő felhasználói követelmények az összes hitelesített felhasználóra érvényesek.

3.2. táblázat. Felhasználói követelmények hitelesített felhasználók számára

| | |
|----------------------------|--|
| Szerződéskötés | <ol style="list-style-type: none">1. A hitelesített felhasználók bármelyik pillanatban létrehozhatnak egy új szerződést.2. Az új szerződés kötése során egy másik felhasználóval egy közös szerződést lehet létrehozni, egy meglévő szerződés sablon alapján.3. Új szerződés kötés esetén a felhasználók kiválaszthatnak egy olyan tulajdoncikket, amit el szeretnének adni a vevő félnek.4. A felhasználók a második fél e-mail címének ismeretében tudnak szerződést létrehozni.5. A felhasználók nem csupán szerződéseket tudnak létrehozni, hanem a meglévő szerződéseiket meg is tekinthetik.6. A szerződéskötés befejezése előtt a felhasználók feltölthetnek melléleteket, amelyek a szerződésre érvényesek.7. A szerződések megtekintése során a felhasználók kereshetnek a szerződéseik között. A keresés intuitív kell legyen. |
| Digitális aláírások | <ol style="list-style-type: none">1. A szerződés digitális aláírása előtt a felhasználók meghívhatnak a szerződéshez tanúkat (bármilyen hitelesített felhasználót) vagy ügyvédeket (különleges felhasználókat), az e-mail címük ismeretében.2. A szerződés megkötése előtt a tanúk és az ügyvédek egyaránt át kell nézzék a szerződést, és digitálisan alá kell írják azt.3. Az adó és a vevő is digitálisan alá kell írja a szerződést.4. Abban az esetben, hogyha a felhasználók nem szeretnék kézírással aláírni digitálisan a szerződést, kiválaszthatják, hogy csupán a nevükkel írják alá azt. |

| | |
|---------------------------------|--|
| <p>Szerződésletöltés</p> | <ol style="list-style-type: none"> 1. A szerződések kitöltése és digitális aláírása után a felhasználók letölthetik a végleges szerződést. 2. A letöltött szerződések tartalmazzák az összes mellékletet, amely a szerződésre, illetve az esetlegesen eladott tulajdoncikkre érvényesek. 3. A letöltött szerződések tartalmaznak egy AVDH (<i>Azonosításra Visszavezetett Dokumentumhitelesítés</i>) igazolást, amely igazolja a felek mivoltát. [9] 4. A végleges szerződés minden fél által alá kell legyen írva. 5. A szerződések végtelen ideig letölthetőek kell legyenek, a letöltési idő lejáratára nem életképes megközelítés. 6. A végleges szerződés továbbküldhető e-mailben is egy gombnyomásra. |
| <p>Tulajdoncikkek</p> | <ol style="list-style-type: none"> 1. A felhasználók létrehozhatnak saját tulajdoncikkeket egy meglévő tulajdon sablon alapján. 2. A tulajdonsablonok megjelennek a tulajdoncikkek létrehozása előtt. 3. Egy tulajdoncikk létrehozása után a felhasználók ki kell töltsék a tulajdoncikkhez tartozó tulajdonságokat. 4. A tulajdoncikk tulajdonságai bekerülnek a szerződések tulajdonságai közé. 5. Egy szerződés keretén belül lehet hivatkozni az eladott tulajdoncikk tulajdonságaira. 6. A tulajdoncikk létrehozása közben, illetve létrehozása után is feltölthetőek olyan mellékletek, amelyek a tulajdoncikkre érvényesek. 7. A felhasználók bármikor megnézhetik, hogy milyen tulajdoncikkeik vannak egy adott tulajdonsablonon belül. |

3.1.3. Ügyvéd felhasználók

Az ügyvéd felhasználók azok a felhasználók, akiknek az oldal adminisztrátorai igazolták személyazonosságukat igazolt ügyvédként. Az ügyvéd felhasználók feladata, hogy átnézzék klienseik szerződéseit, és ellenőrizzék, hogy helyesen kitöltötték az adataikat.

3.3. táblázat. Felhasználói követelmények ügyvéd felhasználók számára

| | |
|----------------------------|---|
| Szerződéskötés | <ol style="list-style-type: none">1. Az ügyvédek bármilyen felhasználó hozzáadhatja a szerződéséhez, amennyiben ismeri az e-mail címét.2. Az ügyvédek csak akkor kerülnek rá egy szerződésre, amennyiben elfogadják a szerződést, amely megjelenik a fiókukban.3. Az ügyvédek rálátást kapnak az egész szerződésre, és látják a tulajdoncikk tulajdonságait is.4. A szerződéskötés befejezése előtt a felhasználók feltölthetnek mellékleteket, amelyek a szerződésre érvényesek.5. A szerződések megtekintése során az ügyvéd felhasználók kereshetnek a szerződéseik között. A keresés minél intuitívabb kell legyen. |
| Digitális aláírások | <ol style="list-style-type: none">1. Az ügyvédek ugyanúgy alá tudják írni digitálisan a szerződéseket, és a szerződésre rá kerül az aláírásuk, illetve az AVDH igazolásuk is.2. Az ügyvédek aláírása nélkül nem kerülhet végleges szerződés kinyomtatásra. |

3.1.4. Rendszergazda felhasználók

A rendszergazda (másnéven *adminisztrátor*) felhasználók felelősek az oldalon található dokumentumsablonok, illetve tulajdoncikk sablonok létrehozásáért. Rendszergazda felhasználót csak egy más rendszergazda felhasználó tud létrehozni. Másodlagos feladatuk, hogy az ügyvéd felhasználókat ellenőrizzék, hogy mivoltuk a valóságnak megfelel.

3.4. táblázat. Felhasználói követelmények rendszergazda felhasználók számára

| | |
|-----------------------|--|
| Adminisztráció | <ol style="list-style-type: none">1. A rendszergazda felhasználók bármikor el kell tudják érni a rendszergazdai munkához szükséges munkálato- kat a felhasználói felületről.2. A rendszergazda felhasználók munkálatai a rendes munkálatok mellett kell megjelenjenek.3. Hasonlóan a rendes munkálatokhoz, az adminisztrációs tevékenységek is többnyelvűsítésre szorulnak. |
| Kézikönyv | <ol style="list-style-type: none">1. A rendszergazda felhasználók bármikor meg kell tud- ják tekinteni az adminisztrátorok számára készített ké- zikönyvet.2. Hasonlóan a rendes felhasználóknak készített kézi- könyvhöz, az adminisztrátorok kézikönyve is használati utasításokat kell tartalmazzon.3. A használati utasítások ki kell terjedjenek a szerződés- sablonok, illetve a tulajdoncikk-sablonok készítésére.4. A kézikönyv elérhető kell legyen ugyanúgy a honlapról, mint az adminisztrációs panelből is. |

| | |
|--------------------------|--|
| Szerződéssablonok | <ol style="list-style-type: none"> 1. A rendszergazda felhasználók képesek kell legyenek önállóan szerződéssablonokat létrehozni. 2. A szerződéssablonokat létre tudják hozni az adminisztrátorok hozzátársított tulajdoncikk típussal is, avagy anélkül is. 3. A szerződéssablonok könnyen szerkeszthetők kell legyenek, egy online szerkesztési felület segítségével. 4. A szerződéssablonok létrehozhatóak csupán egy név és egy leírás megadásával is, nincs értelme, hogy meglevő sablon feltöltése is kötelező legyen. 5. Ha mégis van meglevő sablon a szerződésről, a rendszergazda felhasználónak meg kell adni a lehetőséget, hogy feltöltsön egy ilyen sablont. 6. A sablon dokumentumok a feltöltés után is módosíthatók kell legyenek. 7. A sablonhoz tartozó <i>placeholder</i> opciók könnyen átláthatóak kell legyenek. 8. A sablonhoz tartozó <i>placeholder</i> opciók konfigurálhatónak kell lennie összetett adatok, például azonosítók és telefonszámok kezelésére. |
|--------------------------|--|

3.2. Rendszerkövetelmények

3.2.1. Funkcionális követelmények

Az alkalmazásnak rendelkeznie kell egy főoldallal, amely rövid áttekintést és magyarázatot nyújt a felhasználóknak az oldal funkcióiról. Marketing szempontjából a főoldalnak olyan nyelvezetet és vizuális elemeket is használnia kell, amelyek meggyőzik a felhasználót a weboldal használatáról.

A hitelesítési oldalnak, illetve a dokumentációnak is könnyen elérhetőnek kell lenniük a főoldal menüjén keresztül. Miután a felhasználó sikeresen hitelesíti magát, át kell irányítani az alkalmazás szerződéskötési részének kezdőlapjára. A kezdőlapon az ingatla-

nok létrehozása és kezelése, illetve szerződések létrehozása és menedzsmentje, valamint a személyes adatok kitöltésének lehetősége jelenjen meg.

A szerződéskötés kezdőlapjának fontos eleme az oldalsáv, amely a menüt tartalmazza. A menü a *Project Raccoon* összes fontosabb funkcionális részéhez rálátást nyújt, maximum három klikkelésen keresztül. Mobileszközökön a menü kezdetben el van rejtve, hogy elegendő helyet biztosítson a tartalomnak. A menü helyett egy hamburger menü gomb jelenjen meg, hogy szükség esetén a felhasználó tudja átkapcsolni a menüt.

Vendég felhasználók

3.5. táblázat. Funkcionális követelmények vendég felhasználók számára

| | |
|-------------|--|
| Hitelesítés | <ol style="list-style-type: none">1. A vendég felhasználók számára a fiókok adatait, felhasználónevet, e-mail címet és egyéb adatokat relációs adatbázisban kell tárolni.2. A felhasználó profilképét egy <i>object storage</i> rendszerben kell tárolni, semmiképp sem egy relációs adatbázisban.3. Ha egy felhasználó nagyon gyenge jelszóval próbál felregisztrálni az oldalra, a weboldalnak figyelmeztetnie kell a felhasználót és nem szabad engednie a regisztrációt.4. A rendszer nem engedhet két felhasználó regisztrációját ugyanazzal az e-mail címmel.5. A regisztrációs folyamat után, hogyha a felhasználói adatai nincsenek kitöltve, egy <i>onboarding</i> folyamat révén meg kell kérni a felhasználókat, hogy töltsék ki hiányzó adataikat, mielőtt a rendszer többi részéhez férhetnek.6. Vizuális captcha rendszer helyett láthatatlan captcha használata szükséges regisztrációkor.7. A felhasználók jelszavát a rendszer soha nem mentheti le egyszerű szöveggént. Használjunk erős jelszó hashelő algoritmust. |
|-------------|--|

| | |
|--------------------------|--|
| Kapcsolatfelvétel | <ol style="list-style-type: none"> 1. A felhasználók kapcsolatfelvételi űrlapja a beküldött üzeneteket egy előre meghatározott e-mail fiókba kell továbbítsa. 2. A felhasználók által hagyott e-mail cím és tárgy beke- rül a kapcsolatfelvételi űrlap üzenete mellé abban az e-mailben, amit a rendszer generál a kapcsolatfelvételi űrlap elküldésekor. 3. A kapcsolatfelvételi űrlap elküldésekor egy e-mail ge- nerálódik, ami tartalmazza a felhasználó nevét, e-mail címét, a kapcsolatfelvétel tárgyát és a kapcsolatfelvétel üzenetét. 4. A kapcsolatfelvételi űrlap legalább 15000 karaktert kell támogasson. |
| Kézikönyv | <ol style="list-style-type: none"> 1. A weboldal fejléce kell tartalmazzon egy hivatkozást a felhasználói kézikönyvre. 2. A felhasználói kézikönyv egy külön weboldal, amely Markdown formátumban szerkeszthető a fejlesztők ál- tal. |

Hitelesített felhasználók

3.6. táblázat. Funkcionális követelmények hitelesített felhasználók számára

| | |
|----------------------------|---|
| Szerződéskötés | <ol style="list-style-type: none">1. A szerződéskötéseket le kell menteni egy relációs adatbázisba.2. A szerződéskötések tartalmaznak hivatkozásokat az adóra, a vevőre, illetve a tulajdoncikkre, ha létezik.3. A végső, kigenerált és digitálisan aláírt szerződéseket a rendszer végtelen ideig tárolja.4. A végső szerződések digitálisan alá kell legyenek írva,5. A szerződések kilistázása során az összes szerződést meg kell jeleníteni a felhasználónak.6. A szerződéskötés folyamán feltöltött mellékleteket a rendszernek tárolnia kell.7. A feltöltött mellékletek biztonságos módon kell elérhetőek legyenek. Azok a felhasználók, akiknek nincs rálátásuk a szerződésre, a mellékleteket sem tudják letölteni.8. A szerződések keresése a kliensoldalon történik. |
| Digitális aláírások | <ol style="list-style-type: none">1. A szerződések során létrejött digitális aláírásokat a rendszer lementi egy <i>object storage</i>-ba.2. A szerződés megkötése előtt a rendszer ellenőrzi, hogyha az adó, a vevő, az ügyvédek illetve a tanúk is mind aláírták a szerződést.3. Abban az esetben, hogyha a felhasználók nem szeretnék kézírással aláírni digitálisan a szerződést, a rendszer lementi az adatbázisba, hogy nincsen kézzel írt digitális aláírás, és az aláírást a felhasználó nevével helyettesíti a rendszer. |

| | |
|--------------------------|--|
| Szerződésletöltés | <ol style="list-style-type: none"> 1. A szerződések kitöltésének előfeltétele a szerződés kigenerálása. 2. A rendszer a letöltött szerződésekbe beleékeli a szerződés összes mellékletét, illetve a tulajdon mellékleteit is, ha ez létezik. 3. A letöltött szerződése tartalmaz egy AVDH (<i>Azonosításra Visszavezetett Dokumentumhitelesítés</i>) igazolást, amely igazolja a felek mivoltát. Ez az igazolás egy PDF dokumentum, amely a végső szerződésben megjelenik, mint <i>attachment</i>. 4. A végleges szerződés, melyet e-mailben továbbküldtek, nem módosulhat a továbbküldés után. |
| Tulajdoncikkek | <ol style="list-style-type: none"> 1. A felhasználókhoz tartozó tulajdoncikkeket egy relációs adatbázisba menti a rendszer. 2. Egy tulajdoncikk felhasználása előtt a rendszer ellenőrzi, hogyha a felhasználók kitöltötték a tulajdoncikkhez tartozó tulajdonságokat. 3. A szerződések tulajdonságainak kilistázása során a rendszer lekéri a szerződéshez tartozó tulajdoncikk tulajdonságait is. 4. Ha egy szerződés keretén belül egy tulajdoncikkre hivatkozunk, a rendszer először megbizonyosodik arról, hogy a tulajdoncikk nem része egy másik, folyamatban lévő szerződésnek. 5. A tulajdoncikkhez feltöltött mellékleteket a rendszer <i>object storage</i>-ba menti. 6. A felhasználók csupán a saját tulajdoncikkeiket kérhetik le a rendszerből. |

3.7. táblázat. Funkcionális követelmények ügyvéd felhasználók számára

| | |
|----------------------------|--|
| Szerződészkötés | <ol style="list-style-type: none">1. A rendszer csak akkor számítja az ügyvédet egy szerződés részeként, hogyha az ügyvéd elfogadta a szerződést.2. Ha az ügyvéd elutasítja a szerződést, az adatbázisból törlődik az ügyvéd kapcsolata a szerződéssel.3. Ha egy ügyvéd része a szerződésnek, a rendszer engedélyezi, hogy lekérje a szerződés adatait.4. A szerződészkötés mellékleteit a rendszer egy <i>object storage</i>-ban tárolja. |
| Digitális aláírások | <ol style="list-style-type: none">1. Az ügyvédek kézzel írt digitális aláírásait a rendszer <i>object storage</i>-ban tárolja szerződések.2. A szerződések kigenerálásakor a kézzel írt digitális aláírások átkerülnek az <i>object storage</i>-ból a kigenerált PDF dokumentumba, az AVDH állományokon láthatóakká válnak.3. A végleges szerződés kinyomtatása előtt a rendszer ellenőrzi, hogy az összes ügyvéd aláírta-e a szerződést vagy sem. Ha nem írta alá az összes ügyvéd a szerződést, a szerződés nem kerül kigenerálásra. |

Rendszergazda felhasználók

3.8. táblázat. Funkcionális követelmények rendszergazda felhasználók számára

| | |
|-------------------|---|
| Szerződéssablonok | <ol style="list-style-type: none">1. A rendszergazda felhasználók által létrehozott szerződéssablonok metaadatai relációs adatbázisba kerülnek.2. A szerződéssablonok dokumentumait a Google Docs rendszeren belül tároljuk.3. A szerződéssablonok létrehozásakor, ha hozzá szeretnének társítani tulajdoncikk típust is a felhasználók, akkor a rendszernek ellenőriznie kell, hogy a kért tulajdoncikk típus létezik-e.4. A szerződéssablon dokumentumának szerkesztésére a Google Docs rendszert használjuk.5. A szerződéssablon dokumentumának feltöltésénél nem kötelező már meglévő sablon feltöltése.6. Ha a rendszergazda nem tölt fel előre elkészített szerződéssablont, akkor a rendszer automatikusan létrehoz egy üres Google Docs dokumentumot, és lementi a hozzá tartozó dokumentum azonosítót relációs adatbázisba.7. Ha a rendszergazda egy előre elkészített szerződéssablont tölt fel, akkor a rendszer arról egy másolatot készít a Google Docs rendszerben.8. A sablon dokumentum módosítása során a rendszer létrehoz egy Google Docs-os megosztott linket, ami segítségével bármilyen Google felhasználóval belépve a sablon szerkeszthetővé válik.9. A sablonhoz tartozó <i>placeholder</i> opciók kell támogatásák: szám (minimum és maximum értékekkel), azonosítószám, e-mail cím, webhely link, dátum. |
|-------------------|---|

3.2.2. Nem funkcionális követelmények

3.9. táblázat. Külső követelmények (*external requirements*)

| | |
|--------------------------|---|
| Etikai és legális | <ol style="list-style-type: none">1. Mivel az alkalmazás a szerződések kezelésének és aláírásának koncepciója körül forog, biztosítanunk kell, hogy a felhasználó által létrehozott összes szerződést ellenőrizzük, digitális tanúsítvánnyal aláírjuk és biztonságosan tároljuk a felhőben.2. Az alkalmazásnak biztosítania kell a felhasználó által megadott adatok helyességének ellenőrzését. Ez az AVDH rendszeren keresztül történik.3. Az alkalmazást a GDPR törvényei köré kell építsük, ezért a weboldal első meglátogatásakor engedélyt kell kérnie a felhasználótól a sütik (<i>cookies</i>) tárolására a számítógépén.4. Ha az adminisztrátorokat törlési kéréssel keresik meg, akkor eleget kell tenniük a kérésnek egy rövid ellenőrzési folyamat után.5. A rendszer összes komponense használható kell legyen szigorú kormányzati törvények alatt. Ha az adatok tárolását például csak az Európai Unióban engedélyezi a felhasználási környezet, akkor az összes komponensnek ebben a korlátozott módban kell működnie. |
|--------------------------|---|

3.10. táblázat. Termékkövetelmények (*product requirements*)

| | |
|-----------------------|--|
| Használhatóság | <ol style="list-style-type: none"> 1. A felhasználóknak képesnek kell lenniük bármely művelet elvégzésére a platformon segítség nélkül. 2. Ha egy felhasználónak segítségre van szüksége, képesnek kell lennie arra, hogy az alkalmazás által biztosított dokumentációra hivatkozzon. 3. A felhasználói felületnek elég intuitívnak kell lennie ahhoz, hogy a mezei felhasználók is kényelmesen tudják használni. 4. A felhasználói felület nem tartalmazhat olyan gombokat, szöveget vagy piktogramokat, amelyek túlságosan homályosak ahhoz, hogy megértsék őket a felhasználók. |
| Tárolás | <ol style="list-style-type: none"> 1. A rendszer soha nem tárolhat blobokat relációs adatbázisban. Az ilyen fajta tárolást <i>object storage</i> rendszer segítségével kell végrehajtani. 2. A rendszer összes komponense használható kell legyen szigorú kormányzati törvények alatt. Ha az adatok tárolását például csak az Európai Unióban engedélyezi a felhasználási környezet, akkor az összes komponensnek ebben a korlátozott módban kell működnie. |
| Tervezés | <ol style="list-style-type: none"> 1. A rendszernek kerülni kell a szoros kapcsolódásokat (<i>tight coupling</i>-ot). Csak olyan alkatrészeket használjon, amelyek szoros csatlakozás nélkül bármikor cserélhetők. |

| | |
|---------------------|---|
| Teljesítmény | <ol style="list-style-type: none"> 1. Az alkalmazásnak válaszolnia kell minden felhasználói interakcióra. 2. Ha egy művelet elvégzéséhez több időre van szükség, a felhasználónak vizuálisan jeleznie kell a rendszernek a haladását. |
|---------------------|---|

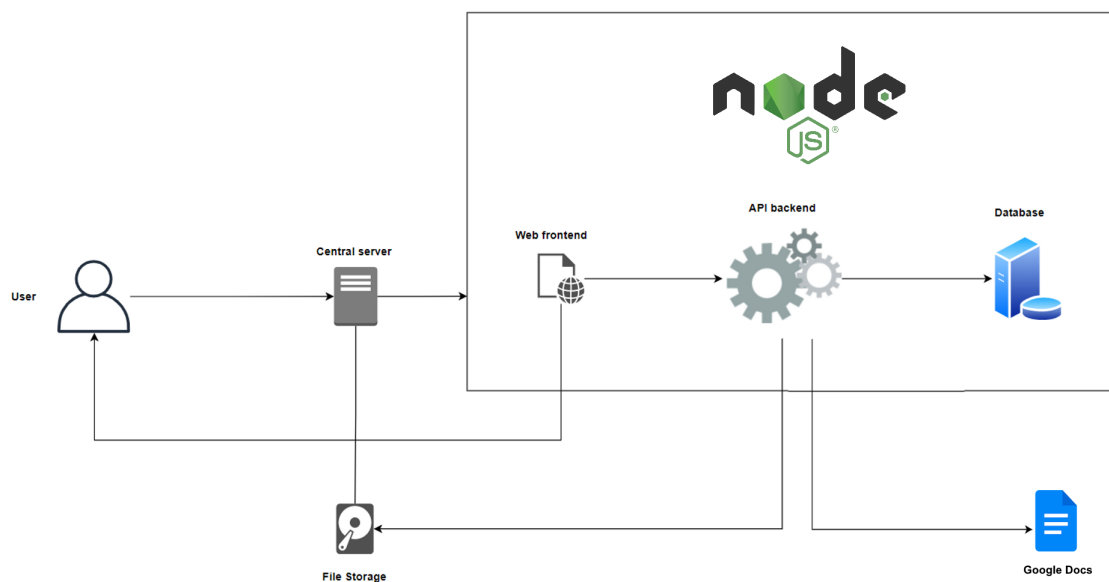
3.11. táblázat. Szervezési követelmények (*organizational requirements*)

| | |
|------------------|---|
| Környezet | <ol style="list-style-type: none"> 1. A GitHub-ot használjuk, mint verzió követő rendszer. 2. A verzió követő rendszeren tárolt kód privát kell legyen. Azokban az esetekben, amikor ez a döntés nem a mi hatáskörünkbe tartozik, a platform forráskódját ideiglenesen nyilvánossá tehetjük. 3. Fejlesztéshez a Visual Studio Code környezetet használjuk. Szükség esetén alkalmazhatóak a felhő alapú környezetek is, mint például a GitHub Codespaces. |
|------------------|---|

4. fejezet

Tervezés

4.1. A szoftver architektúrája



4.1. ábra. A Project Raccoon architektúrája, felülnézetből

Célunk az volt, hogy olyan szoftverarchitektúrát hozzunk létre, amely könnyen érthető és nehezen komplikálható túl. A legtöbb projekt kudarcot vall, hiszen túl gyorsan túl bonyolulttá válik. A kód- és mappaszerkezetet is úgy választottuk, hogy egyértelmű és tömör legyen.

A bővíthetőségre nagy hangsúlyt fektettünk. Az új funkciók beépítése a kódbázisba egyszerű folyamat az MVC alapelveit követve. A projekt keretrendszerét, könyvtárait és telepítési stratégiáit úgy választottuk, hogy támogassák a horizontális skálázhatóságot. Ez

azt biztosítja, hogy a Project Raccoon működőképes maradjon, még akkor is, ha a bejövő forgalom megnövekszik - a Project Raccoon teljesen megfelel a *serverless* paradigmának.

A Project Raccoon 5 nagyobb komponensből áll:

1. **Web Frontend** - A felhasználói felület, amelyet a felhasználók látni fognak. Az API backend-el fog kommunikálni. A *Project Raccoon* keretén belül a Next.js keretrendszert használjuk a web frontend megvalósítására, hiszen a React könyvtárra épül, responzív oldalak készítését engedi anélkül, hogy bármilyen szoftverarchitektúrai döntésekre kényszerítsen minket.
2. **API Backend** - Kezeli a frontendtől érkező kéréseket, feldolgozza azokat, majd megfelelő választ küld. Kommunikál az általunk választott tároló- és adatbázis-szolgáltatásokkal, illetve a Google Docs rendszerrel. Feladata, hogy biztonságosan lehetővé tegye a szolgáltatásaink működtetését. Futtatható egyaránt külön, erre a célra kitalált szerveren, vagy *serverless* felhőben.
3. **Adatbázis** - Itt tároljuk a felhasználók adatait, valamint a platformon keresztül létrejött és aláírt szerződéseik részleteit. Használhatunk MySQL, PostgreSQL, MongoDB vagy Firestore adatbázisokat, hogy ezt a szolgáltatást működtessük.
4. **Fájltároló** - Itt tároljuk magukat a szerződéseket, miután véglegesítették őket. Ezeket a fájlokat később nem lehet törölni vagy módosítani. A felhasználók profilképeit is a fájltároló intézi. Használhatunk S3 kompatibilis felhő szolgáltatókat, mint például magát az Amazon S3-at, a Cloudflare R2-t vagy akár a Wasabi-t is. A Google szolgáltatásai közül a Google Cloud Storage is használható.
5. **Google Docs** - Itt tároljuk a szerződések sablonait. Az adminisztrátorok közvetlenül használják a Project Raccoon keretén belül a Google Docs rendszert, hogy a sablonokat szerkesszék. Projektünk mélyen összefonódik a Google Docs rendszerével, és a dokumentumsablonok szerkesztésének frontendjeként szolgál.

4.2. A projekt mappaszerkezete

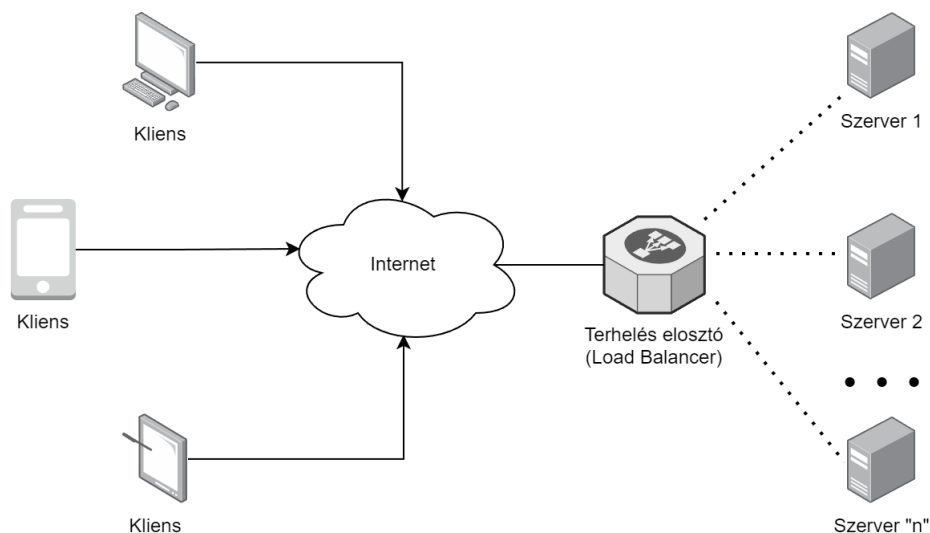
```
/
├── /.avdh
├── /.github
├── /.vscode
├── assets
│   └── ...
├── components
│   └── providers
│       └── ...
├── config
├── content
│   └── ...
├── controllers
│   └── ...
├── db
│   ├── common
│   ├── migrations
│   └── models
│       └── ...
├── hooks
├── layouts
├── lib
│   └── storageStrategies
├── middleware
├── pages
│   ├── api
│   │   └── ...
│   └── ...
├── public
│   └── ...
├── services
│   ├── apis
│   │   └── ...
├── storage
│   └── ...
├── styles
│   ├── components
│   └── fonts
├── tsconfig
├── typings
└── validators
```

4.1. táblázat. Mappaszerkezet magyarázata

| Mappa neve | Mappa tartalomjegyzéke |
|----------------------|---|
| .avdh | Tartalmazza az AVDH azonosításhoz és a digitális aláírások technikai megvalósításához szükséges digitális kulcsokat, .P12 formátumban. |
| .github | Tartalmazza azokat a GitHub Actions munkafolyamatokat, amelyek ellenőrzik a kódbázis helyességét, és linting hibákat keresnek minden egyes commit felkerülése után. |
| .vscode | Visual Studio Code-hoz tartozó segédleírásokat tartalmaz, melyek megkönnyítik a Tailwind CSS használatával írt komponensek fejlesztését. |
| assets | Tartalmazza a weboldal statikus elemeit - képeket, ikonokat és zászlókat többnyelvűsítésre. |
| components | Tartalmazza az alkalmazásban használt felhasználói felületek újrafelhasználható React komponenseit. |
| components/providers | Tartalmazza azokat a React providereket, amelyek a felhasználó szolgáltatásáért felelősek a komponensek számára. |
| config | Tartalmazza a projekt konfigurációs fájljainak a specifikációját TypeScriptes interfészek formájában. |
| content | Tartalmazza a felhasználói kézikönyv <i>Markdown</i> -ban írt tartalmait. |
| controllers | Tartalmazza azokat a vezérlőket (<i>kontrollereket</i>), amelyek a business logikát implementálják az applikáción belül. |
| db | Az adatbázissal kapcsolatos kódrészeket tartalmazza. |
| db/common | Olyan adatbázis modellek ösztályait tartalmazza, amelyek több modellben szerepelnek. |
| db/migrations | Tartalmazza az adatbázismigrációs fájlokat, amelyekkel a sémaváltoztatásokat kezeljük a TypeORM rendszer segítségével. |
| db/models | Az adatbázismodelleket tartalmazza, amelyek adataink struktúráját reprezentálják a TypeORM keretrendszerében. |
| hooks | Több komponensben is használt React hook-okat használ. A React hookok függőségbefecskendezést (<i>dependency injection</i> -t) tesznek lehetővé. |
| layouts | Tartalmazza az alkalmazás különféle weblapjainak a <i>wrapper</i> felrendezéseit. |
| lib | A projektben használt segédfüggvényeket tartalmazza. |

| | |
|-----------------------|---|
| lib/storageStrategies | Az adattárolási stratégiák implementációit tartalmazza: S3, Firebase, illetve lokális adattárolás. |
| middleware | Tartalmazza a kéréskezelésben használt middleware függvényeket. Az általunk használt middleware-ek a hitelesítés érvényesítéséért felelősek. |
| pages | Az alkalmazás egyes frontend oldalainak implementációját tartalmazza. |
| pages/api | Tartalmazza a szerveroldali logikát végző API végpontokat. Fontos megemlíteni, hogy itt nem valósul meg üzleti logika. Minden üzleti logika kontrollerekben valósul meg. |
| public | Tartalmazza azokat a szerver által kiszolgált nyilvános fájlokat, amelyek a felhasználói kézikönyv böngészése során elérhetőek. A konkrét weboldal statikus fájljai az assets mappában találhatóak. |
| services | Tartalmazza a szerveroldal által használt szolgáltatások összeköttetéseinek implementációit. |
| services/apis | Tartalmazza a kliensoldal által használt API-ok meghívására használandó szolgáltatásokat. |
| storage | Abban az esetben, hogyha a weboldal magán a szerveren tárolja a fájlokat, itt találhatóak a másképp <i>object storage</i> -ban tárolt állományok. |
| styles | Tartalmazza a globális stílusokat az alkalmazás megjelenítéséhez. |
| styles/components | Tartalmazza a komponensekhez tartozó stílusokat. |
| styles/fonts | Az alkalmazásban használt betűtípusokat tartalmazza. |
| tsconfig | A projekt TypeScript konfigurációs fájljait tartalmazza. |
| typings | Tartalmazza a projektben használt custom TypeScript típusdefiníciókat, olyan függőségek számára, amelyek nem támogatják a TypeScript típusait. |
| validators | Validátorokat tartalmaz az alkalmazásban használt különféle űrlapok ellenőrzésére. |

4.3. Kliens-szerver architektúra



4.2. ábra. Tradicionális kliens-szerver architektúra

A kliens-szerver architektúrában a felelősségek két kulcsfontosságú komponens - a kliens és a szerver - között oszlanak meg. A kliens-szerver architektúra magába foglalja a **kliens** fogalmát: egy felhasználóval szemben álló alkalmazást vagy felületet, amely kapcsolatként szolgál a felhasználó és a számítógép között, illetve a **szerver** fogalmát: egy biztonságos számítógépet, mely elvégzi az adatok feldolgozását.

A kliens és a szerver egy számítógépes hálózaton keresztül kommunikál egymással. A kliensek kéréseket küldenek a szerverek felé, a szerverek pedig a kért információk szolgáltatásával vagy a szükséges feladatok elvégzésével válaszolnak. Ez az architektúra lehetővé teszi a központi vezérlését az adott applikációnak, így alkalmas például webalapú rendszerek kiépítésére.

A Project Raccoon is kliens-szerver architektúrát alkalmaz, még *serverless* környezetben is. A hagyományos telepítések során a Project Raccoon egy központi szerverre van telepítve, amely gondoskodik az API backendről, a frontendről és opcionálisan az adatbázisról és a fájl tárolásról is. A szerver nélküli *serverless* telepítések esetén sem beszélünk teljesen szerver *nélküli* architektúráról: a szerver automatikusan elindul, amikor a kliensek információt kérnek a szolgáltatástól.

A 4.2 ábrán látható egy tradicionális kliens-szerver architektúra. A kliensek kéréseket tesznek a szolgáltatás felé. Ezek a kérések átmennek az interneten, amely a kérésüket a terhelés elosztókhöz (*load balancer*-hez) irányítja, függetlenül attól, hogy a kliensek földrajzilag hol helyezkednek el.

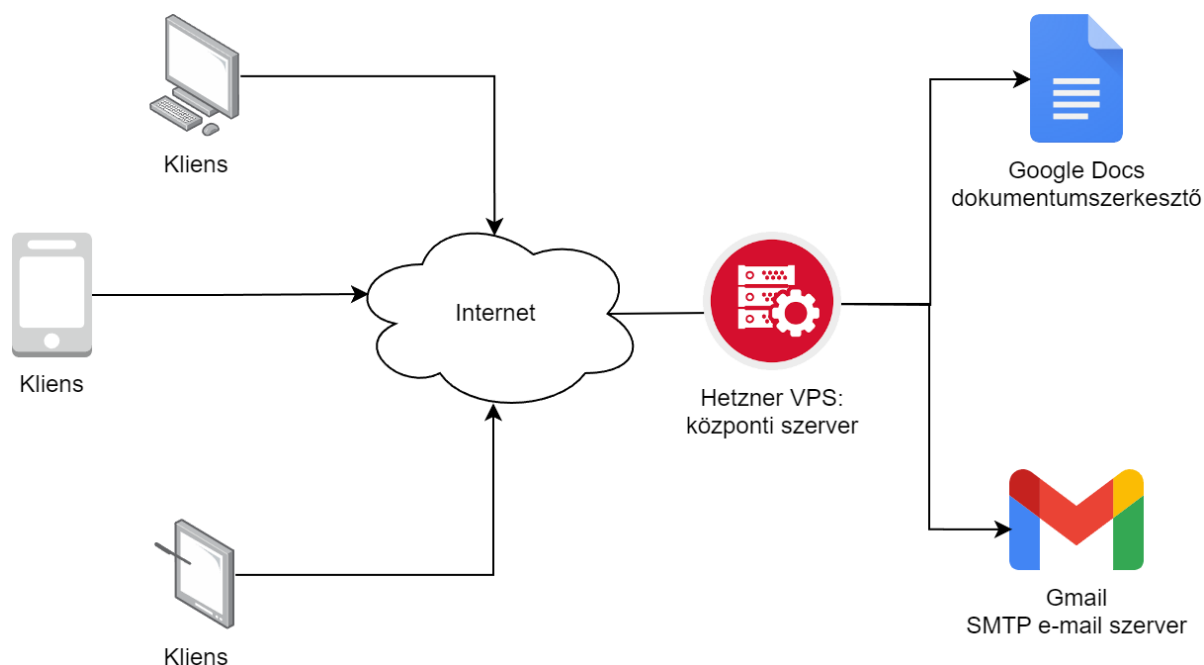
A *load balancer* feladata, hogy kiossza a beérkező kéréseket a rendelkezésre álló szerverek között. Ez lehetővé teszi a rendszer vertikális skálázását, és megelőzi, hogy bárme-

lyik szerver túlterheltté váljon. A *load balancer* a kérések útbaigazításához több stratégiát vethet be. [10] Ezek közül megemlítenék egy párat:

- **Körkörös teherelosztás (*Round-robin balancing*):** Egy egyszerű elvet követ. Körkörösén váltogatja a kiszolgáló szervert a rendelkezésre álló kiszolgálók listájából. Minden új kérést szekvenciálisan a listán következő szerverhez rendelünk, biztosítva ezzel, hogy a munkaterhelés egyenletesen oszlik el az összes rendelkezésre álló erőforrás között. A körkörös terheléelosztás megakadályozza egyetlen kiszolgáló szerver túlterhelését, miközben maximalizálja a rendszer általános teljesítményét és átbecsátóképességét. Ez a megközelítés könnyen megvalósítható, skálázható, és egyszerű módot biztosít a nagy mennyiségű kérések kiegyensúlyozott kezelésére. [10]
- **Legkisebb kapcsolaton alapuló teherelosztás (*Least-connection load balancing*):** Célja, hogy a bejövő kéréseket több az aktuális terhelési szint alapján ossza el. Ez a technika arra törekszik, hogy minimalizálja az egyes kiszolgáló szerverek által kezelt kapcsolatok számát. [10] Biztosítja, hogy a munkaterhelés egyenletesen osztódjon el az összes rendelkezésre álló erőforrás között. Amikor új kérés érkezik, a *load balancer* dinamikusan kiválasztja azt a szervert, amelyik a legkevesebb aktív kapcsolattal rendelkezik, és a kérést ehhez rendeli hozzá. Ez minimalizálja a szerver túlterhelésének kockázatát. Ez a megközelítés különösen hatékony olyan esetekben, ahol a szerverek kapcsolatterhelése jelentősen változhat.
- **IP-címen alapú teherelosztás (*IP hash load balancing*):** A bejövő kérések forrásának IP-címe alapján kiszámít egy hash-értéket, és hozzárendeli azt az egyik elérhető kiszolgálóhoz. A hozzárendelés konzisztens marad az azonos IP-címről érkező későbbi kérések esetében, így biztosítva, hogy egy adott kliens minden kérése ugyanarra a kiszolgálóra irányuljon. Azáltal, hogy a IP-címet használja a terheléelosztás kulcsaként, biztosítja, hogy a felhasználók munkamenetének késleltetése azonos marad. [11] Ez különösen létfontosságú a valós idejű alkalmazásoknál, ahol fontos, hogy a késleltetés mértéke egyenletes maradjon. A *Project Racoon* is tartalmaz valós idejű komponenseket - nevezetesen valós idejű csevegést biztosít az eladó és a vevő között a szerződéstárgyalások során.

Nem mindig szükséges terheléelosztó használata, de erősen ajánlott, ha előre látható, hogy nagyon sok felhasználó fogja használni a szolgáltatásunkat. Ha nincs terheléelosztó, akkor a kliensek automatikusan egy központi kiszolgáló szerverhez (*central server*) csatlakoznak. Ebben az esetben a szerverek kiválasztásában nem játszik szerepet terheléelosztó algoritmus, mivel mindig csak egy szerver áll rendelkezésre. Ez azonban különösen veszélyes, mert ha a központi kiszolgáló szerver összeomlik vagy más módon elérhetetlenné válik, nem marad egyetlen kiszolgáló sem, amely a beérkező kéréseket kezelné.

A terheléselosztás különösen fontos, hogy a szolgáltatás leállási ideje (*downtime*) minimális legyen. Ezért a *Project Raccoon* úgy van felépítve, hogy lehetővé tegye a vertikális skálázás használatát. Ez nem feltétlenül szükséges - a *Project Raccoon*-t továbbra is lehet futtatni az összes architektúráis többletköltség nélkül, például több adatbázis telepítése, terheléselosztók és több szerver konfigurálása nélkül.



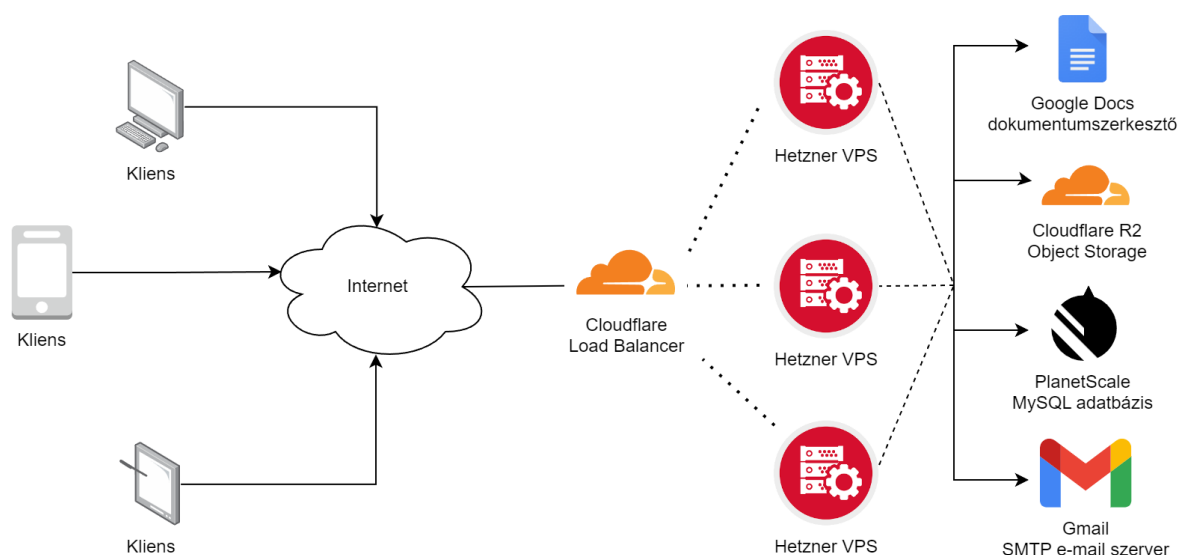
4.3. ábra. A *Project Raccoon* legegyszerűbb kliens-szerver kivitelezése

A 4.3 ábrán látható a *Project Raccoon* legegyszerűbb kitelepítése. Maga a Project Raccoon által szolgáltatott web frontend, API backend, az adatbázisok és a fájl tárolás egy Hetzner virtuális számítógép által vannak szolgáltatva. Ebben az esetben a terhelés nincs kiegyenlítve. Ha a Hetzner VPS meghibásodik vagy leáll, a szolgáltatás teljesen elérhetetlenné válik. Megengedhetjük magunknak, hogy az adatbázist és a fájl tárolót ezen az egy szerveren hosztoljuk, mert csak ennek az egy szervernek kell hozzáférnie ezekhez.

A 4.2 táblázatban levezetjük ennek a megoldásnak az előnyeit és hátrányait. Mivel ennek a megközelítésnek nagyon erős hátrányai vannak, ajánlott egy bonyolultabb, de nagyobb teljesítményű telepítési architektúrát választani.

| Előnyök | Hátrányok |
|--|---|
| Nincs korlátja annak, hogy mit tehetünk a szerveroldalon. | Szükséges egy szerver kibérlése, bekonfigurálása és folyamatos üzemeltetése. |
| A szerver mindig gyorsan válaszol, hiszen nem szükséges külső szolgáltatásokra várakozni. | Nem skálázható vertikálisan - nem tud nagy mennyiségű felhasználót kiszolgálni. |
| Relatív kevés anyagi erőforrást igényel. | Vízszintes skálázás esetén a szerver erőforrásainak kibővítése rendkívül költséges lehet. |
| Egyszerű felépítése révén könnyen telepíthető és futtatható is. | A telepítés gyakran unalmasabb, és Linux szerver ismereteket igényel. |
| A Docker használatával bármilyen architektúrájú processzorra kitelepíthető, például ARM szerverekre is. [12] | |

4.2. táblázat. A legegyszerűbb kivitelezés előnyei és hátrányai



4.4. ábra. A *Project Raccoon* részletesebben kidolgozott telepítésének kivitelezése

A 4.4 ábrán látható a *Project Raccoon* részletesebben kidolgozott telepítésének felvázolása. Ebben az esetben is marad a hagyományos szerver-kliens architektúra, hiszen a centralizált szerver felváltja három, egymástól független Hetzner virtuális gép.

Mivel már három különböző szerverről beszélünk, ezért át kell gondoljuk a szerverek hatáskörét. Az előző megvalósításban az adatbázis, illetve a fájlok tárolása egyetlen egy szerveren történt, nem volt szükség külső szolgáltatás használatára. Ebben az esetben viszont szigorúan el kell különítsük a szerverektől az adatokat tárolását. Hogyha az adatokat ugyanazon a szerveren tároljuk, akkor az az adott szerver könnyen túlterheltté válik.

| Előnyök | Hátrányok |
|---|--|
| Vertikálisan skálázható, új szerverek hozzáadásával a rendszerben. | Még mindig szükséges a szerverek kibérlése. Mivel több szervert bérlünk, ezért több pénzbe kerül a rendszer futtatása. |
| A szervereket nem terhelik túl a felhasználók kérései. A Docker használatával egyszerűsíthető a szerverek kitelepítése. | Nem skálázható vertikálisan - nem tud nagy mennyiségű felhasználót kiszolgálni. |
| Ha megsemmisülnek a szerverek egy hiba során, az adatok megmaradnak a Cloudflare object storage-ban és a PlanetScale adatbázisában. | Ha nincsen túl sok felhasználónk, nincs értelme egy ennyire komplex kitelepítési architektúrát érvénybe léptetni. |
| A Cloudflare és a PlanetScale automatikus biztonsági mentéseket végeznek az adatokról. | A telepítés még mindig Linux szerver ismereteket igényel. |
| | A PlanetScale adatbázisa kicsit különbözik a megszokott MySQL adatbázisoktól, így fennáll a <i>vendor lock-in</i> veszélye. (A vendor lock-in olyan helyzetet jelent, amikor erősen függünk egy adott szolgáltatástól, ami megnehezíti vagy költségesé teszi az alternatív szolgáltatásokra való váltást.) |

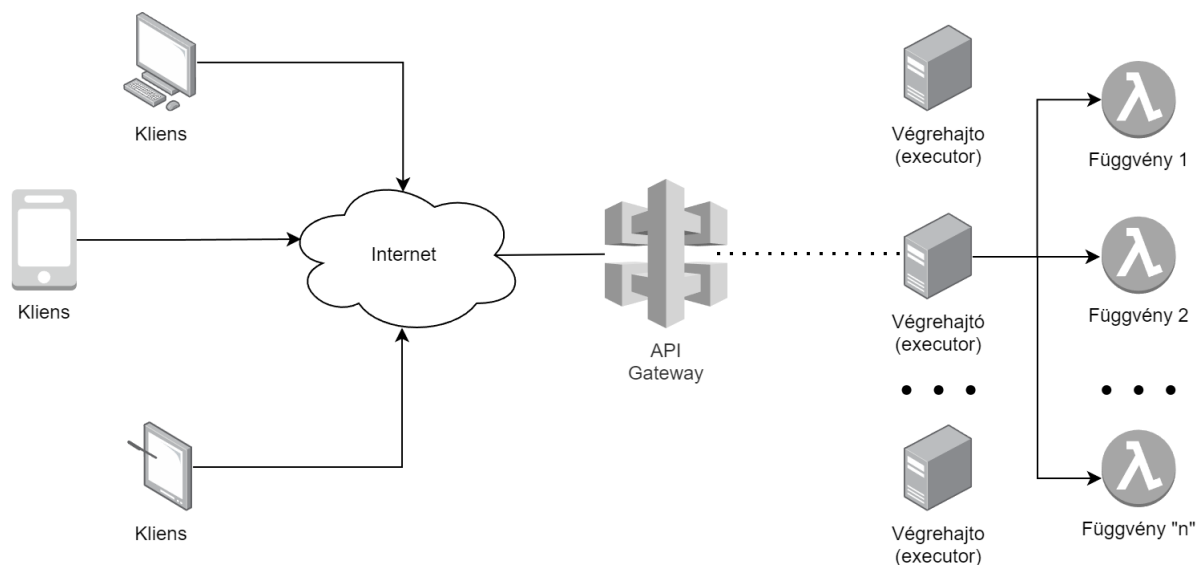
4.3. táblázat. A kidolgozott kivitelezés előnyei és hátrányai

Hogyha az adatokat külön-külön minden szerveren tároljuk, akkor redundáns adattárolás történik, sőt, szinkronizálnunk kell az adatokat az összes szerver között, ami egyáltalán nem egyszerű dolog.

Két új szolgáltatást vezetünk be: egy automatikusan önskálázó adatbázis-szolgáltatót, a *PlanetScale*-t illetve a Cloudflare R2 *Object Storage*-t. A *PlanetScale* feladata, hogy a három szerverünk felé szolgáltatson egy olyan adatbázist, amely automatikusan skálazza magát, és bármely időpillanatban elérhető a szerverek számára. A relációs adatbázisunk adatait ebben a rendszerben tárolhatjuk. A három szerver ezentúl nem tárolja saját magában az adatbázis adatait. Hasonlóképpen, a Cloudflare R2 *Object Storage* átveszi a fájl tárolás feladatait a Hetzner VPS-től. A Cloudflare R2 centralizált tárolóhelyet biztosít a *Project Raccoon*-nak, lehetőséget adva a rendszernek, hogy itt tárolja a végleges szerződéseket, esetlegesen feltöltött mellékleteket, illetve a felhasználók profilképeit.

A 4.3 táblázatban levezetjük e megoldás előnyeit és hátrányait. Mivel ez a megoldás kevesebb hátránnyal rendelkezik, már alkalmazható huzamosabb ideig, több felhasználó kiszolgálására is.

4.4. Serverless architektúra



4.5. ábra. Kliens-szerver architektúra megvalósítása serverless technológiával

A *serverless* architektúra egy felhőalapú modellre utal, ahol az infrastruktúra és a szerver menedzsment feladatai el vannak vonatkoztatva a fejlesztőktől, így azok kizárólag a kód írására és telepítésére koncentrálhatnak. Ez a megközelítés kiküszöböli a szerverek rendelkezésre bocsátásának, konfigurációjának és kezelésének szükségességét, mivel a felhőszolgáltató dinamikusan kezeli a kód végrehajtásához szükséges erőforrásokat. Mivel a *Project Raccoon* alkalmazást a Next.js keretrendszerének kontextusában építettük fel, az alkalmazás teljes mértékben kitelepíthető *serverless* technológiákkal is. A *serverless* technológiák segítségével a *Project Raccoon* kitelepítése egyszerűbbé és skálázhatóbbá válik.

A szerver nélküli kitelepítés stratégiájával a weboldalunk minden egyes elérhetőségét egy külön függvényként kitelepítjük a serverless felhőbe. Ezek a függvények mindig élnek a felhőben, viszont csak akkor futnak, mikor a felhasználók kérik. Mikor egy kérés érkezik egy felhasználó számáról, a kérés legelőször egy API Gateway-hez kerül. Az API Gateway megkeresi a felhőben lementett függvény lokációját, ahol tárolva van a függvény. Ezután megnézi, hogyha már van olyan szerver a felhőben, amely jelenleg futtatja az adott függvényt. Ha van, a kérést ahhoz a szerverhez továbbítja, ahol a kért függvény fut. Ha nincs, akkor keres egy olyan szervert, amely nincs túlságosan lefoglalva kérésekkel. A kapott szerver letölti a felhőből a függvény forráskódját, és elkezdi futtatni. Ezt nevezik hidegindításnak (angolul: *cold boot*). Sajnos ez enyhe késleltetést eredményez az olyan alkalmazások esetében, amelyeket egy ideje nem használt egyetlen felhasználó sem.

| Előnyök | Hátrányok |
|--|---|
| Könnyű telepítés és skálázhatóság | A serverless szolgáltató által előírt összes korlátozás érvényesek. |
| Kevesebb időtöltés szerverek beállításával | Egy esetleges DDoS (<i>Distributed Denial of Service</i>) támadás során a szolgáltatás működtetésének költségei az egekbe szökhetnek, nagy anyagi károkat okozva. |
| Nem szükségesek a Linux szerver adminisztráció lépéseinek ismerete. | A hidegindítások nagyon lassúvá teszik a kapcsolat létesítését a szolgáltatással, ha hosszabb ideig nem használják. |
| Nincs szükség külön szerverre - minden a szerver nélküli funkciókon fut a felhőben. | |
| Kevés felhasználó esetén sokkal olcsóbb a szolgáltatás serverless futtatása, mint a szerver bérlése. | |

4.4. táblázat. A serverless kivitelezés előnyei és hátrányai

A hidegindítás után, ha az alkalmazást nem használja huzamosabb ideig egyetlen felhasználó sem, a függvény lekapcsolódik és új kérés esetén újra szükségessé válik egy új szerver keresése és a függvény újratelepítése (még egy hidegindítás szükséges). [13]

A Project Raccoon esetében a *serverless* technológiákkal való kitelepítést a Vercel platform használatával ajánljuk. Mivel a Next.js keretrendszert a Vercel készítette, ezért nyilvánvaló, hogy a Next.js teljes mértékben fel van szerelve a *serverless* technológiák támogatására. A *serverless* telepítés során ugyancsak szükséges az alkalmazás összekötése a Google Docs, Cloudflare R2, PlanetScale és Gmail rendszerekkel. A Cloudflare Load Balancer szerepét lecseréli a Vercel API gatewayje, amely a Vercel hálózatában automatikusan kiosztja a szerver nélküli függvények a kiszolgáló szerverek között. Nincs szükség saját szerver futtatására, karbantartására.

A 4.4 táblázatban a *serverless* megközelítés előnyeit és hátrányait tárgyaljuk. Ez a megközelítés olyan esetben megfelelő, amikor vagy alig felhasználó, és nincs igazi oka egy teljes szerver felállításának, vagy akkor, amikor olyan sok felhasználó van, hogy nagyon nehéz lenne egy egész szervercsoportot (*server cluster*-t) kezelni.

4.5. MVC - Modell-Nézet-Vezérlő

A *Project Raccoon* keretén belül az MVC szoftver architektúrális mintát használjuk. Az **MVC** - Modell-Nézet-Vezérlő minta megengedi nekünk, hogy a kódbázisunkat moduláris modulokból építsük fel.

A modell képviseli az alkalmazás adatait és üzleti logikáját, és kezeli az adatbázisokkal és külső szolgáltatásokkal való adatinterakciókat. A nézet (*view*) kezeli az adatok megjelenítését a felhasználók számára, felhasználói felületét jelképezve a szoftvernek. A modell és a nézet közötti hídként működő Vezérlő kezeli a felhasználók kéréseit, dolgozza fel az adatokat, és ennek megfelelően frissíti a modellt és olykor a nézetet. A felelőségek egyértelmű szétválasztása így módon megkönnyíti a kód karbantartását.

4.5.1. Modellek

A *Project Raccoon* az adatok modellezésére a TypeORM keretrendszert használja. A TypeORM keretén belül a TypeScript-el megírt osztályok automatikusan leképezhetőek relációs és nem-relációs adatbázisokra is egyaránt. A relációs adatbázis sémák metaadatait, mint több más hasonló keretrendszerben is, a modellekben megadhatjuk dekorátorok segítségével. Minden modellünk két részből áll:

1. egy **interfészből**, amely meghatározza az entitás adatait egy általános, nem TypeORM-specifikus módon - lásd [4.1](#) kódrészlet;
2. illetve egy **TypeORM entitásból**, amely implementálja az interfészt, TypeORM-specifikus és metaadatokat is tartalmaz dekorátorok formájában - lásd [4.2](#) kódrészlet.

```
export interface IItem {
  id?: number;
  createdAt?: Date;
  updatedAt?: Date;
  friendlyName?: string;
  slug?: string;
  description?: string;
  options?: IItemOption[];
}
```

4.1. kódrészlet. Egy modell interfészének megvalósítása

```

@Entity()
export class Item implements IItem {
  @PrimaryGeneratedColumn()
  id: number;

  @CreateDateColumn()
  createdAt: Date;

  @UpdateDateColumn()
  updatedAt: Date;

  @DeleteDateColumn()
  deletedAt?: Date;

  @Column()
  friendlyName: string;

  @Column({ unique: true })
  slug: string;

  @Column()
  description: string;

  @OneToMany(() => ItemOption, (itemOption) => itemOption.item)
  options: Partial<ItemOption[]>;
}

```

4.2. kódrészlet. Egy modell entitásának megvalósítása

4.5.2. Nézetek

A *Project Raccoon*-ban a nézetek React komponenseknek felelnek meg. Hosszú csata folyik a React programozók között. Néhányan a régi, osztályalapú megközelítést részesítik előnyben a React felhasználói felület komponensek létrehozásakor, míg mások az újabb, funkcionális megközelítést szeretik jobban. Mi a funkcionális komponensek mellett döntöttünk. Ez azt jelenti, hogy ahelyett, hogy minden React nézetet egy osztályként hozunk létre, helyette minden nézet egy-egy függvénynek írunk meg. Ezért nem is tudunk szolgáltatni hagyományos osztálydiagrammal.

Az osztály-alapú nézetek és a funkcionális nézet között a legnagyobb különbség, hogy míg az osztály-alapú nézetekben explicit módon meg kell határozni az állapotváltozókat és minden más adatforrást, a funkcionális komponensek esetén használható a függőség befecskendezés fogalma (angolul: *dependency injection*), lásd: [4.3.](#) kódrészlet.

```
const LoginForm = () => {
  const [user, setUser] = useCurrentUser();

  if (user) {
    return <p>You are already logged in as {user.username}!</p>;
  }

  return (
    <form>
      ...
    </form>
  );
}
```

4.3. kódrészlet. A felhasználó objektum befecskendezése egy React komponensbe

4.5.3. Vezérlők

A *Project Raccoon* rendszeren belül csupán a vezérlők képesek módosítani közvetlenül a modelleken. A vezérlők a *controllers* mappában élnek, és entitás szerint csoportosítva vannak. Minden egyes vezérlő minden egyes függvényére hivatkozik egy API route a *pages/api* mappából.

A *next-bar* csomag használatával a *pages/api* mappában meghatározhatunk API útvonalakat ([4.4.](#) kódrészlet). Ezek az API útvonalak a HTTP metódus típusa alapján kerülnek kiválasztásra, mint például POST, GET, DELETE stb.

```
import bar from 'next-bar';
import { ensureAdministrator } from '@middleware/auth';
import * as contractOptionsController from
  '@controllers/contract-options/contractOptionsController';

export default bar({
  post: ensureAdministrator(contractOptionsController.newContractOption),
  get: ensureAdministrator(contractOptionsController.listContractOptions),
});
```

4.4. kódrészlet. Példa implementáció egy API útvonalra, mely egy vezérlőre csatlakozik

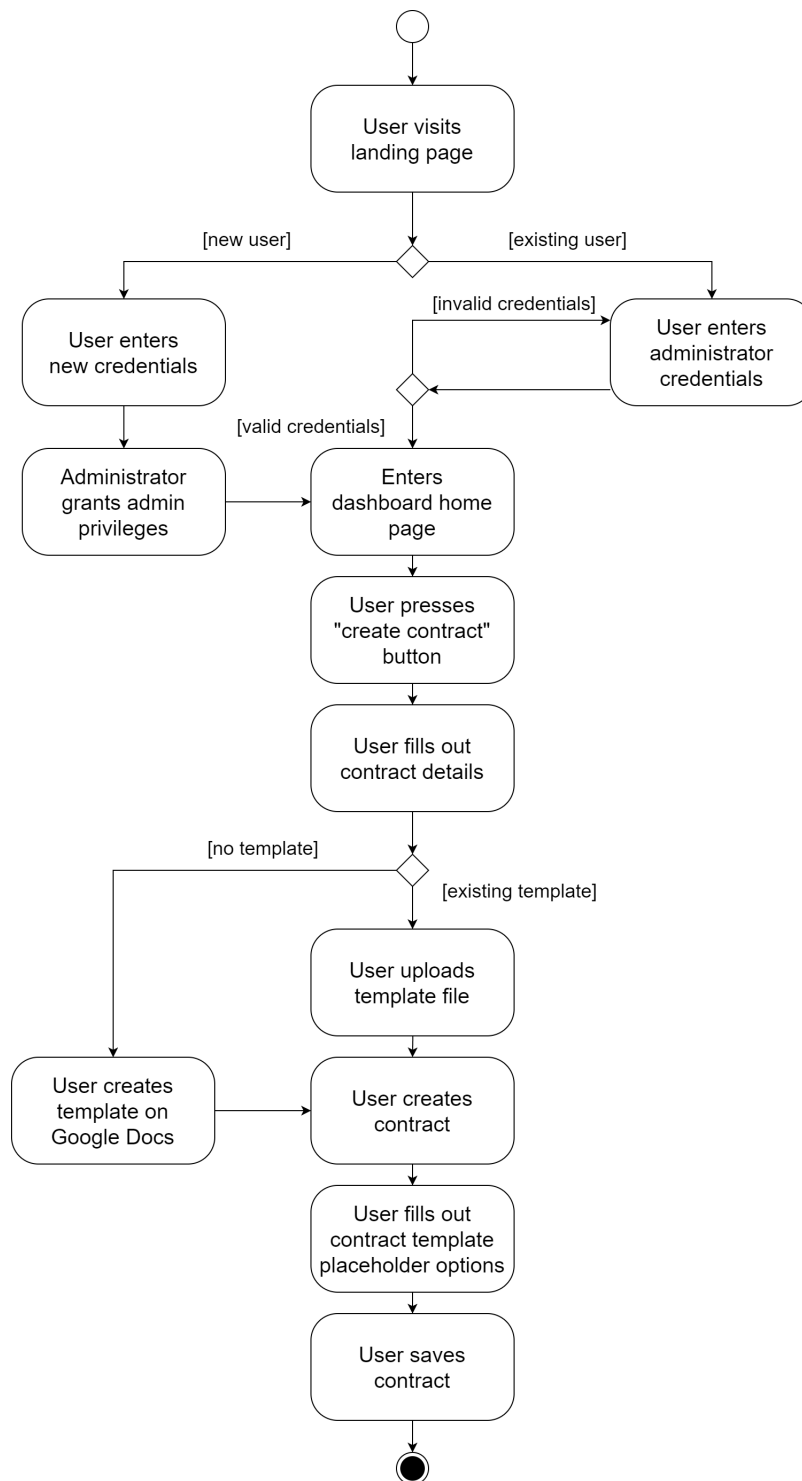
Ez esetben a vezérlő a *controllers/contract-options/contractOptionsController.ts* állományban él. Két függvényt tartalmaz. Az egyik függvény egy POST hívásra létrehoz egy új szerződés sablon opciót, a másik függvény pedig egy GET hívásra kilistázza az összes szerződés sablon opciót. A vezérlő feladata, hogy kiolvassa a HTTP kérésekből a szükséges információkat, és hogy hajtsa végre a kért műveletet. Abban az esetben, hogyha olyan megszorítások vannak egy API útvonalra, amelyek nem teljesülnek, a kérés nem hajtódik végre. A 4.4. kódrészletben például mindkét funkcionalitás csak akkor érhető el, hogyha a felhasználó adminisztrátor felhasználó.

Fontos megemlíteni, hogy a fő vezérlő fájlokban csupán a HTTP kérések feldolgozása hajtódik végre. Minden üzleti logikát, aszerint, hogy mit hajt végre, külön fájlban implementáltunk ugyanabban a modulban. Ez esetben például a „*newContractOption*” függvényhez tartozik egy *controllers/contract-options/newContract.ts* állomány is, amely végrehajta a szó szerinti üzleti logikát.

4.6. Szolgáltatási réteg

A felhasználói felület a HTTP-kéréseket nem közvetlenül a felhasználói felületből indítja, hanem a szolgáltatási rétegnek delegálja. A szolgáltatási réteg a *Project Raccoon* azon modulja, amely a fejlesztőtől elabsztraktizálja a HTTP kérések logikáját, és az entitásokhoz tartozó szolgáltatásokban gondolkodik. Amikor a felhasználói felületnek egy műveletet kell végrehajtania, vagy adatokat kell kérnie az API-tól, az adott entitásért felelős szolgáltatással lép kapcsolatba, ahelyett, hogy magára HTTP-kérést intézne. Ez nagyon megkönnyíti egy stabil API létrehozását, amelyet nagyon nehéz helytelenül használni, mivel az összes API-specifikus kommunikációt különálló modulokba szét lehet választani és külön-külön tesztelni lehet.

4.7. Szerződésablonok létrehozása



4.6. ábra. A szerződések létrehozásának activity diagramja

A 4.6. ábrán látható a szerződések létrehozásának activity diagramja. A tevékenység kezdetén a felhasználó meglátogatja a weboldal főoldalát.

A weboldal főoldalából két lehetőség nyílik a felhasználó fele: Ha a felhasználó már rendelkezik egy adminisztrátor fiók hitelesítési adataival, akkor beírhatja ezeket a hitelesítési adatokat. Hogyha a rendszer helyesnek véli ezeket az adatokat, akkor továbbengedi a műszerfal kezdőlapjára. Ha viszont a rendszer helytelennek gondolja a hitelesítési adatokat, a felhasználó újból be kell írja azokat.

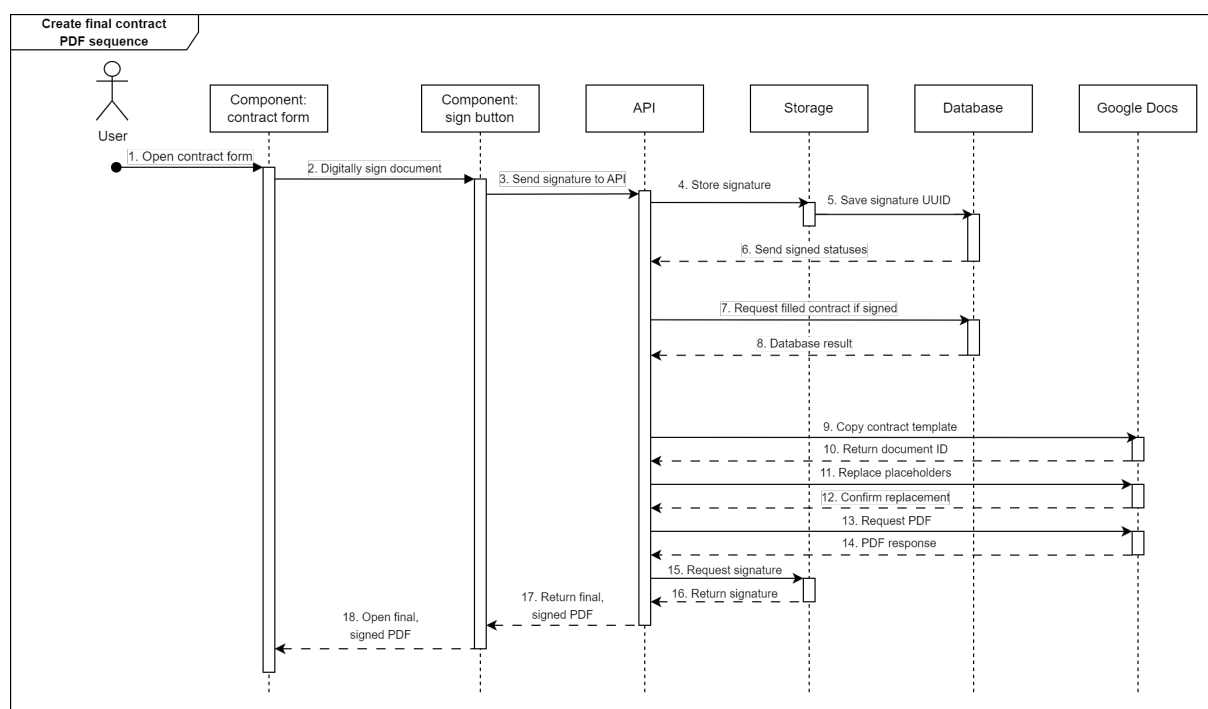
Ha nincsen adminisztrátor fiókja a felhasználónak, akkor létrehoz egy új fiókot, új hitelesítési adatok megadásával. A hitelesítési adatok megadása után megvárja, hogy egy másik adminisztrátor adminisztrátor jogokkal ruházza fel őt. Ezután kerül át a felhasználó a műszerfal kezdőlapjára.

Miután a felhasználó a műszerfal kezdőlapján van, rákattint a "Szerződés létrehozása" gombra. Ezután kitölti a szerződés sablon összes részletét. Hogyha a felhasználó már készült egy előre elkészített sablonnal, akkor feltöltheti azt. Ha nem készült már előre elkészített sablonnal, akkor a felhasználó létrehoz egy új, üres sablont a Google Docs-ban.

Miután létrejött a dokumentum sablonját, a felhasználó létrehozza a szerződéstípust. A szerződéstípus létrehozása után a felhasználó kitölti az összes olyan űrlapopciót, amelyet a felhasználó megváltoztathat, és amely a Google Docs rendszerben létrehozott dokumentumsablonon megtalálható.

Az űrlapopciók létrehozása után a felhasználó lementi a szerződés típusát, a folyamat végére érve.

4.8. Végleges szerződések kigenerálása



4.7. ábra. A végleges szerződések kigenerálásának szekvencia diagramja

A 4.7 ábrán látható a végleges szerződések kigenerálásának szekvencia diagramja. Ebben a szekvenciában hét alkomponens játszik szerepet: maga a felhasználó, a szerződés űrlapja, az aláíró gomb, az API backend, a háttértároló, az adatbázis illetve a külső Google Docs rendszer.

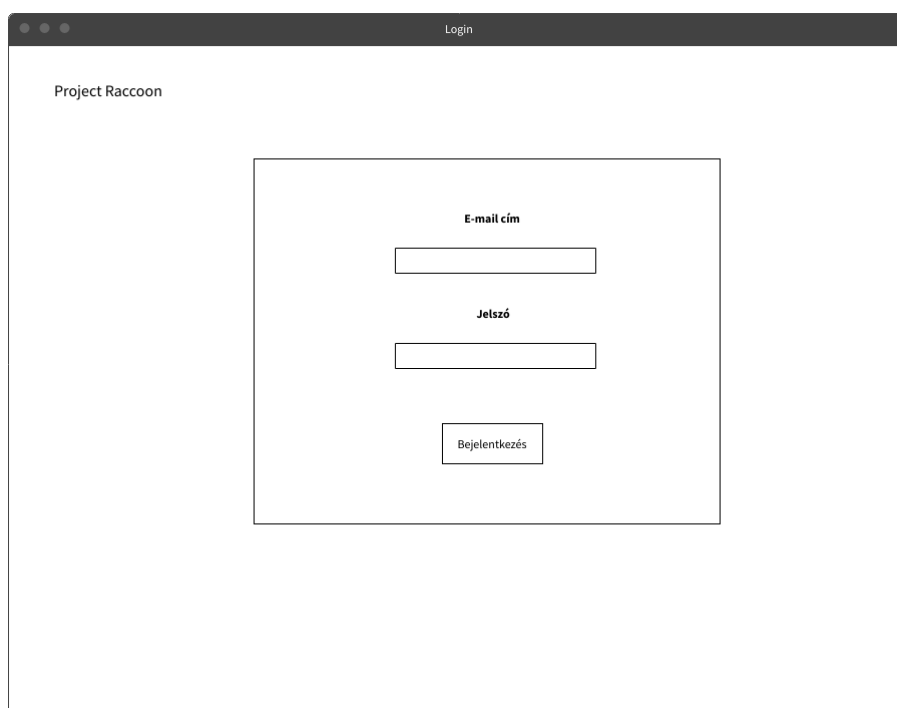
A folyamat elején a felhasználó kinyitja a szerződés űrlapját, és a szerződés űrlapján rákattint az aláíró gombra. A dokumentum így digitális aláírásra kerül. Az aláíró gomb elküldi a digitális aláírást az API számára, amely továbbküldi azt a háttértárolónak. A háttértároló kigenerál egy egyetemesen egyedi azonosítót, ami jellemzi az adott digitális aláírást, és elküldi az adatbázisnak. Az adatbázis visszatéríti az API-nak, hogy mely felhasználók írták eddig alá a szerződést és melyek nem.

Ha az összes felhasználó aláírta a szerződést, akkor folytatódik a végleges szerződés kigenerálása. Az API kér egy másolatot a szerződés sablonáról a Google Docs rendszertől. A Google Docs visszaszolgáltatja az új másolat azonosítóját. Az API ezután összegyűjti az összes űrlap elemet, amelyet helyettesíteni szeretne a dokumentumon. Ezeket a helyettesítéket kötelegelve elküldi a Google Docs rendszernek. A Google Docs visszaküldi a kicserélt űrlapelemek számát az API-nak. Ezután az API megkéri a Google Docs rend-

szertől, hogy generáljon egy PDF dokumentumot. A Google Docs rendszer visszatéríti az API-nak a kitöltött PDF dokumentumot.

A kitöltött PDF dokumentum még nincs aláírva. Az API lekéri a háttértárolóból a digitális aláírásokat. A háttértároló visszaszolgáltatja az API-nak ezeket az aláírásokat. Az API beszúrja a PDF dokumentumba a *pdf-lib* és *node-signpdf* könyvtárak segítségével a digitális aláírásokat, kigenerálja az AVDH tanúsítványokat, és visszatéríti a végleges, digitálisan aláírt PDF állományt az aláíró gombnak, ahonnan visszakerül a szerződési űrlapra, és kinyílik a felhasználó számára, hogy megtekintse.

4.9. Felhasználói felület tervezése



4.8. ábra. A Project Raccoon belépési felületének drótváza

A következőekben a tervezés során kialakított drótvázról (*wireframe*) lesz szó. Ezen *wireframe*ek segítségével átfogó képet kaphatunk a felületek kinézetéről, az elemensek elhelyezkedéséről, egymáshoz való viszonyulásáról és méretéről. Mindezt anélkül, hogy az apró részletek, színek elhomályosítanak ezen nagyobb körvonalakat. A tervezés során elsődleges szempont volt a felület könnyű használhatósága, az átláthatóság és a rezponzivitás. A folyamat során igyekeztünk betartani Nielsen által felsorolt heurisztikákat [14] a felület minél nagyobb fokú használhatóságának érdekében.

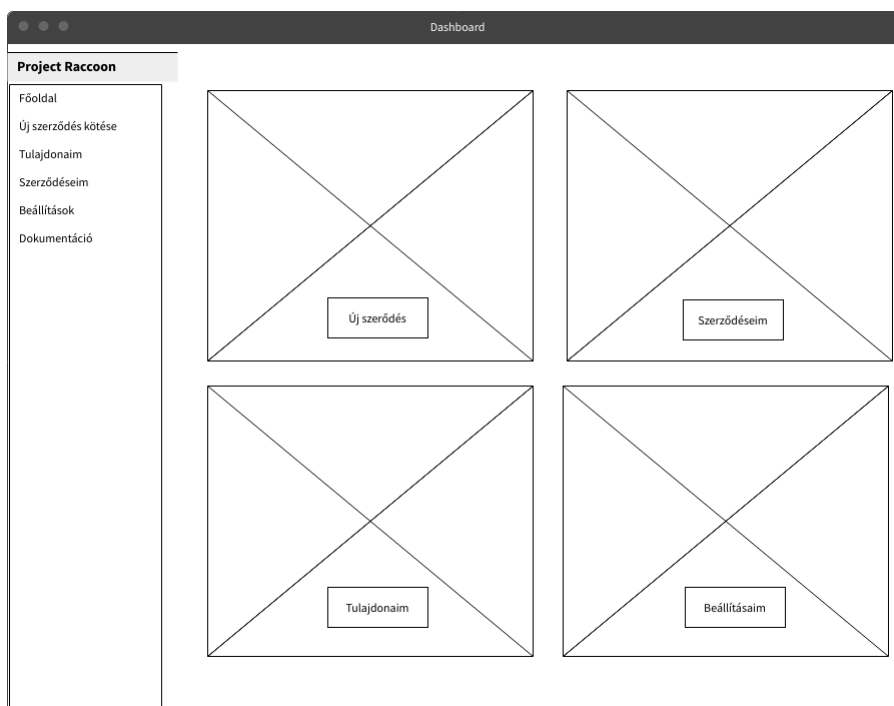
Tekintsünk meg néhány fontosabb oldal *wireframe*jét. A bemutatott dizájn az átlagfelhasználók szemszögéből fogja bemutatni a weboldalt (tehát eltekintünk most az

4.9. ábra. A Project Raccoon regisztrációs felületének drótváza

adminisztrátori jogkörrel ellátott felhasználóktól). Legelsőnek a 4.8. ábrán láthatjuk a bejelentkezési oldal drótvázát, amely egyszerűen csak a felhasználót hitelesítő adatok beírása végetti szövegdobozokat és a belépéshez szükséges gombot tartalmazza középre igazítva. A belépéshez egy email cím és egy jelszó szükséges. Miután a felhasználó ezeket beírta, a „Bejelentkezés” gombra kattintva be tud lépni a platformra és megtekintheti a saját *dashboard*-ját.

Amennyiben a felhasználó nem rendelkezik fiókkal, regisztrálhat a regisztrációs felületen, amely a következőképp néz ki (4.9. ábra). A felület tartalmaz négy szövegdobozt, amelybe a felhasználó beírhatja adatait ezek rendre a következők: név, email cím, jelszó és a jelszó újbóli beírása - az elgépelések kivédéséért fontos. Ezután pedig egy „Regisztráció” gomb következik a beírt szövegek elküldése és a regisztráció finalizálása végett.

Sikeres regisztráció után a felhasználó be tud lépni a bejelentkezési felületen. Miután ez megtörtént, megtekintheti a *dashboard*-ját, amely a tervezési fázisban a következőképp nézett ki. Ezt a 4.10. ábra is szemlélteti. Bal oldalt található egy menü, amely segítségével a felhasználó navigálhat a weboldalon: létrehozhat új szerződést, megtekintheti a tulajdonait, valamint a szerződéseit. Különböző beállításokat végezhet, mint például: adatai módosíthatja, vihet be új adatokat, amelyek szükségesek lehetnek a szerződések kitöltése érdekében, de nem voltak elkérve a regisztráció során (anyja születési neve, személyi igazolványának száma, lakhelye stb.). Illetve megtekintheti a platform dokumentációját,



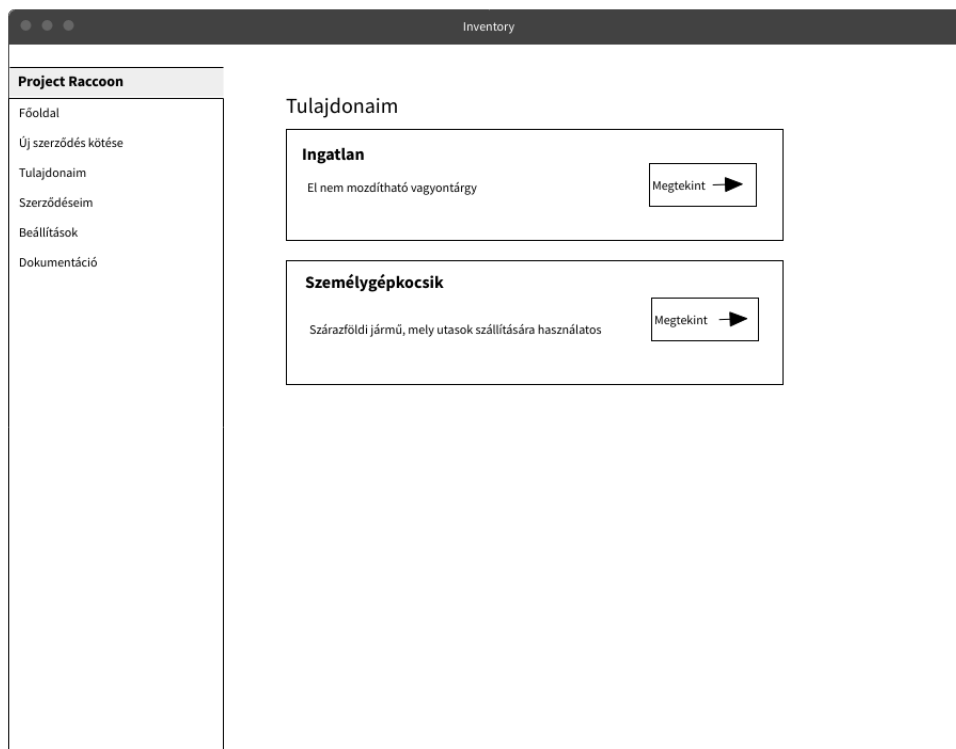
4.10. ábra. A Project Raccoon *dashboard*jának drótváza

vagyis a súgót (használati kézikönyv) - ha bármilyen problémája akad a felhasználónak az oldal használata közben, többek között könnyedén megkeresheti a megoldást/informálódhat ebből is. Ez növeli a felhasználók kényelmét, és az önálló használatot. Tanulmányok már 2013-ban kimutatták, hogy az emberek szívesebben választanak önálló segítségkérési lehetőségeket, mintsem emailt írnak, hívást kezdeményezzenek egy ügyfélszolgálatnál, illetve, hogy „a felhasználók 70% elvárja, hogy egy portálon igénybe tudjanak venni olyan segítséget, amelyhez nem kell más ember beavatkozása” [15]. Végezetül pedig egy „Főoldal” menüpont is szerepel, amely visszavisz a *dashboard*ra. A navigációs menü felett található a platform neve is, amely megjelenik ugyanúgy belépéskor és regisztrációkor is. Jelenleg ezt egy egyszerű „Project Raccoon” felirat szemlélteti.

Középre helyezve pedig megtalálhatóak ugyanezek a főbb alponatok, csupán nagyban kiemelve, hogy még szembetűnőbbek legyenek a felhasználók számára.

A következő 4.11. ábrán a „Tulajdonaim” oldalt láthatjuk, amely a portál adatbázisába felvitt ingó és ingatlanokat listázza. Oldalt a már elmagyarázott navigációs menü található, középső pedig a két kategóriára osztott tulajdonok. Ezek megnevezései alatt egy rövid leírás található az adott kategóriáról, illetve egy-egy gomb, ami még részletesebb megtekintését/listázását adja az adott tulajdonoknak.

A fontosabb wireframek közül utolsónak a szerződéseket listázó oldalt tekintsük meg. A 4.12. ábra ezt szemlélteti. Bal oldalt a fentebb elmagyarázott navigációs sáv található, középső pedig a szerződések típusai szerint kategóriákra osztva láthatjuk és menedzserel-



4.11. ábra. A Project Raccoon tulajdonokat (kategóriánként) megjelenítő felületének drótváza

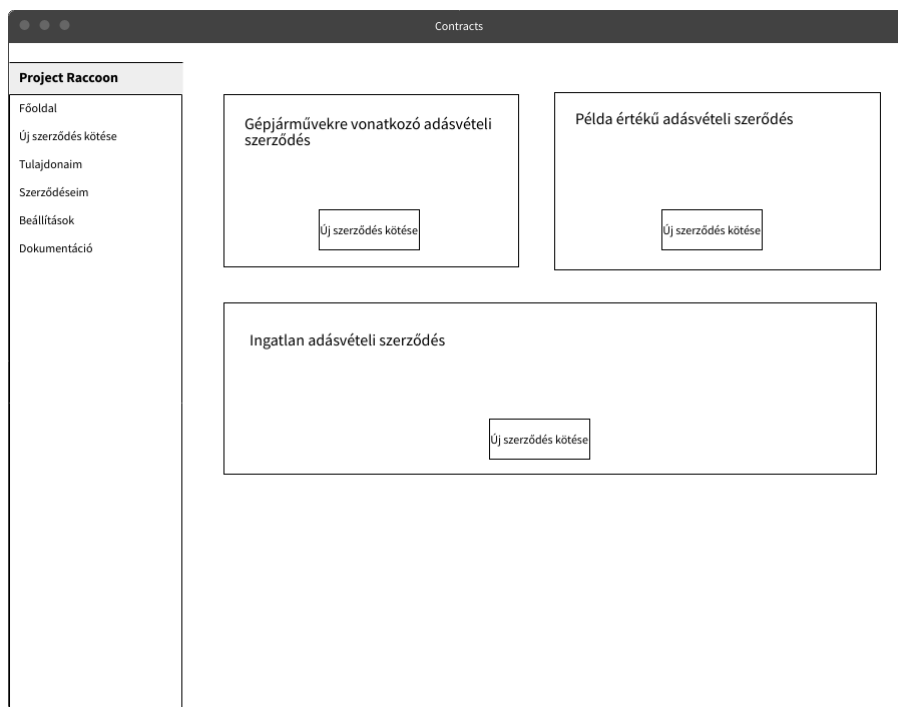
hetjük a szerződéseinket. Ezalatt értjük új szerződések kitöltését/létrehozását, amelyet a kategórián belüli „Új szerződés kitöltése” gombra kattintva meg is tehet a felhasználó.

Az eddigi példákból látható, hogy a tervezést a minél egyszerűbb használat, a letisztult design valamint a hatékony használhatóság vezérelte. A következő, 5. fejezetben részletesebben kitértünk a megvalósítás részleteire, a felületek megvalósítására.

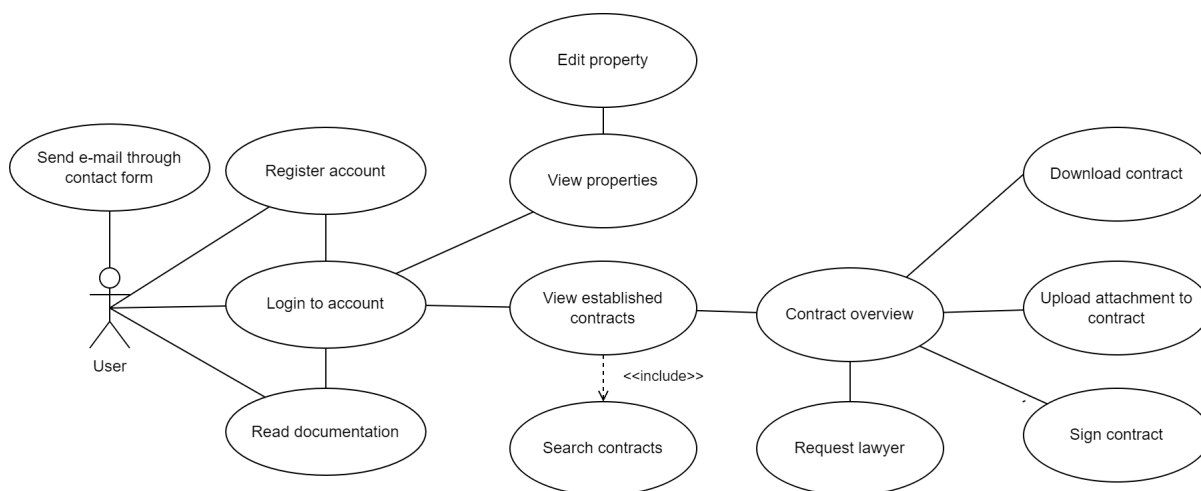
4.10. Felhasználói use case-ek

A 4.13. ábrán látható az összes funkcionalitás, amelyre jogosultak a felhasználók. Mivel nagyon sok funkcionalitás van, az ábrán egy leegyszerűsített diagram látható. Kezdeti fázisban, mikor a felhasználó a főoldalon van, küldhet egy e-mail-t a weboldal adminisztrátorainak egy kapcsolatfelvételi űrlapon keresztül, készíthet egy saját fiókot, beléphet egy már létező fiókba, vagy megtekintheti a felhasználók számára készített kézikönyvet.

A belépés után a felhasználók megnézhetik a meglévő szerződéseiket vagy megtekinthetik a saját tulajdonaikat. A saját tulajdonaik megtekintése közben az egyes tulajdonokat szerkeszteni is tudják. A szerződések megtekintése közben kereshetnek a szerződések között, vagy akár megtekinthetnek egy meglévő szerződést.



4.12. ábra. A Project Raccoon szerződéseket listázó felületének drótváza



4.13. ábra. A hitelesített felhasználók use case diagramja

A meglévő szerződések megtekintése során a felhasználók kérhetnek ügyvédet a szerződés ellenőrzésére, letölthetik a szerződést, amennyiben mindenki aláírta, feltölthetnek mellékleteket egy nem véglegesített szerződés keretén belül, vagy aláírhatják a szerződést.

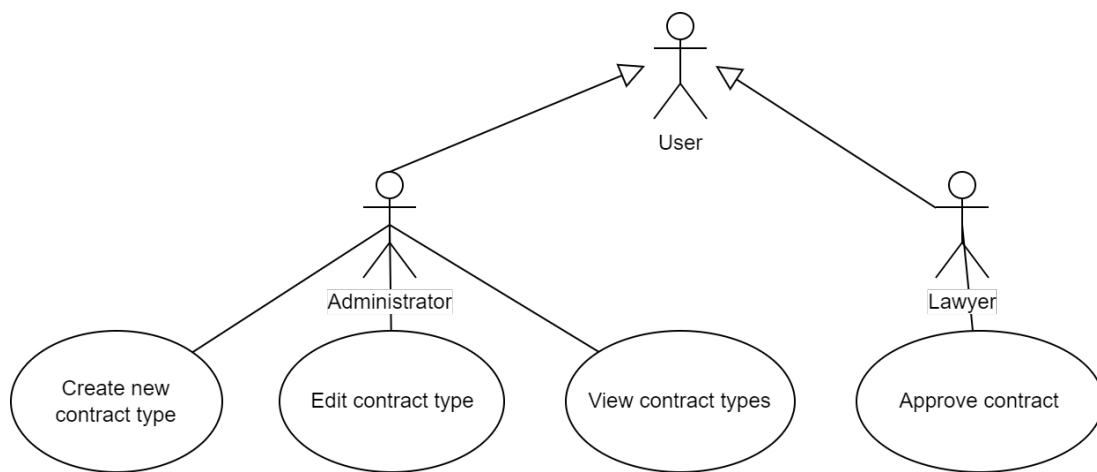
A 4.5. táblázatban az egyik legfontosabb use case: a szerződések letöltésének kiterjesztett magyarázatát láthatjuk.

4.5. táblázat. Felhasználási eset: Szerződés letöltése

| Felhasználási eset | |
|------------------------------------|--|
| Felhasználási eset száma: | PR-1 |
| Felhasználási eset leírása: | Egy felhasználó letölti az egyik szerződése véglegesített, kitöltött PDF verzióját. |
| Előfeltételek: | <ol style="list-style-type: none"> 1. A felhasználónak be kell jelentkeznie. 2. A szerződés összes kell lennie. 3. A szerződést minden félnek alá kell írnia, beleértve az ügyvédeknek is. 4. A felhasználónak hozzáférése kell legyen a szerződéshez. |
| Alapfolyamat: | <ol style="list-style-type: none"> 1. A felhasználó bejelentkezik a kezelőfelületre. 2. A felhasználó listázza az összes szerződését, majd a kívánt szerződésre kattint, legyen az manuális kereséssel vagy a "Szerződések keresése" felhasználási esettel. 3. A felhasználó rákattint a "Szerződés letöltése" gombra. 4. A szerződés letöltődik a felhasználó számítógépére PDF fájlként. |

A 4.14. ábrán látható a speciális felhasználók számára elérhető funkcionálisok. Mivel az adminisztrátorok és az ügyvédek egyaránt felhasználók, mindenre képesek, amire a rendes felhasználók is képesek.

Az ügyvéd felhasználók alá tudják írni azokat a szerződéseket, amelyre meghívták őket, illetve az adminisztrátorok képesek létrehozni új szerződés típusokat, ki tudják listázni és meg tudják módosítani egyenként őket.

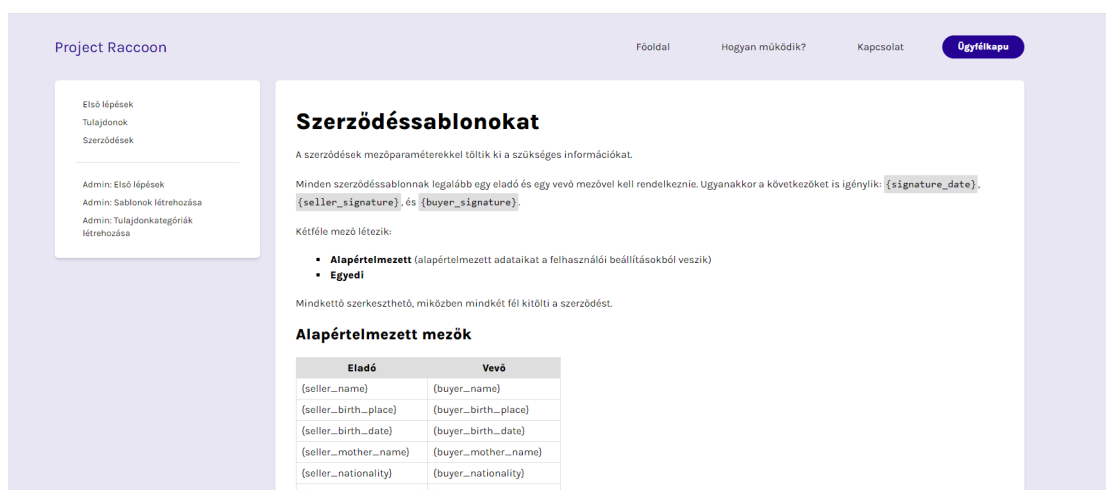


4.14. ábra. A speciális felhasználók use case diagramja

5. fejezet

Kivitelezés

Ahogy az előző fejezet végén tárgyaltuk a felületek létrehozásánál, az elsődleges szempont a könnyű kezelhetőség és átláthatóság volt. Ezt kiegészítette a felhasználók számára létrehozott Sútó opció is, amelyet a portál kézikönyvének is nevezhetünk - a felhasználók útbaigazítását, esetleges problémamegoldást biztosít egyéb külső segítség (ügyfélszolgálat) nélkül. Az 4. fejezetben kifejtettük ennek egyre jelentősebb fontosságát.



5.1. ábra. A szerződéssablonok létrehozásának súgója

A súgó, vagyis a menüben „Dokumentáció”-ként megjelenő menüpont két részre osztozik. Létezik egy az átlagos felhasználók számára, és készült egy az adminisztrátorok számára is. Erre a döntésre azért jutottunk, mert nem szerettük volna az átlagos felhasználók számára is megjeleníteni olyan információkat, amelyeket nem tudnak elérni, tehát használhatatlan lett volna számukra. Így minden, amit megtalálnak a súgóban az releváns és naprakész marad számukra, nem árasza el őket a végtelen/szüretlen információk tárháza. A 5.1. ábrán láthatjuk az adminisztrátorok számára készült verzió egyik oldalát, pontosabban a szerződéssablonok létrehozását és magyarázatát. Ezzel kanyarodjuk

is át egy fontos témakörhöz, mégpedig ahhoz, hogyan kezeli a Raccoon, és pontosabban hogyan adhatunk hozzá mi szerződéssablonokat, mint adminisztrátorok a portálon.

A szerződéssablonok feltölthetők az „Új szerződés létrehozása” menüpont alatt (amely megtalálható a navigációs menü admin részén, közvetlenül az előző fejezetben tárgyalt menüpontok alatt). A szerződések mezőparaméterekkel töltik ki a szükséges információkat. Minden szerződéssablonnak legalább egy eladó és egy vevő mezővel kell rendelkeznie. Ugyanakkor a következőket is igénylik: `{signature_date}`, `{seller_signature}`, és `{buyer_signature}`. Kétféle mező létezik:

- Alapértelmezett mező (alapértelmezett adataikat a felhasználói beállításokból veszik)
- Egyedi mező

Alapértelmezett mező esetén a következőkre kell gondolni: a vevő, illetve eladó neve, email címe, telefonszáma, személyi igazolvány típusa és száma, születési helyük és idejük stb. A `{signature_date}` egy speciális mező, amely tartalmazza azt a dátumot, amikor a szerződést mindkét fél aláírta. A `{seller_signature}` és a `{buyer_signature}` helyére mind az eladó, mind a vevő neve nagybetűvel kerül.

Egyedi mezők megadására is lehetőség van, ekkor ezeknek meg kell adni típusát, beírható karakterek hosszát és egyéb esetleges megkötéseket stb a második fázisban.

A 5.2. ábrán láthatunk egy ilyen példa szerződést, hogy hogyan is nézne ki a mezőkkel.

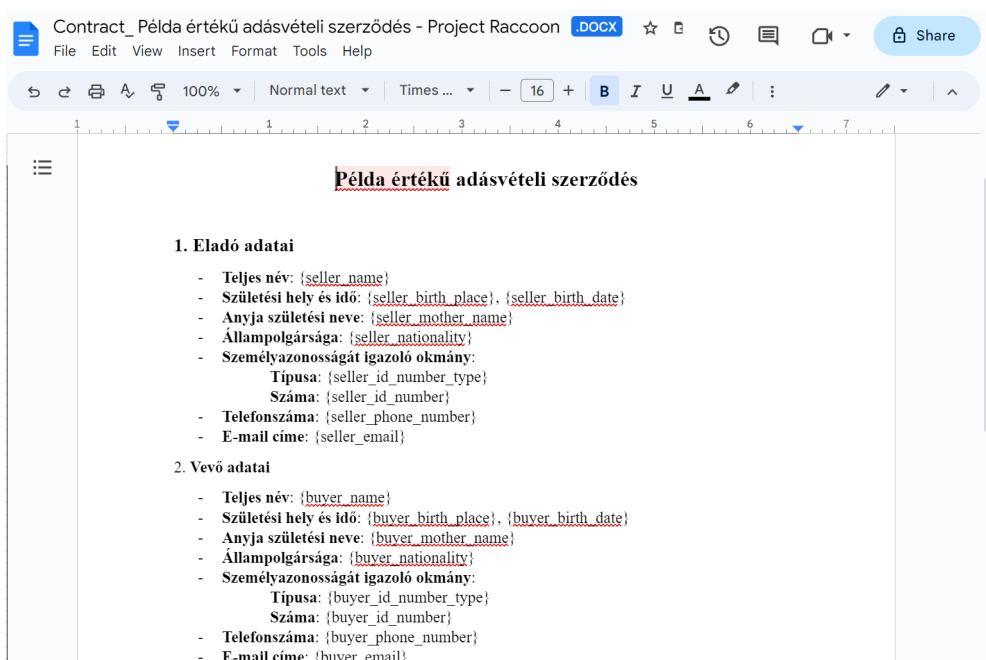
Ezen sablondokumentumok szerkeszthetők Google Docs segítségével, illetve új sablonokat is fel lehet tölteni.

Egy tényleges új sablon bevezetéséhez szükséges a továbbiakban a platformon megadni a helyettesítési mezőket és azok „értékeit”. Vagyis miután feltöltöttük a sablont, amely tartalmazza a cserélendő szövegeket a következő formában: `{replacement_option}`, az adminisztrátornak meg kell mondania, hogy ezeket a `{}` közé írt szövegeket a Raccoon-mal helyettesítse. Illetve itt történik a különböző megkötések megadása is, amelyek ezekre a mezőkre érvényesek kell legyenek.

Összefoglalva tehát a folyamat két lépésből áll:

1. sablon készítése Google Docs-on
2. helyettesítési mezők megadása a Raccoonon

Hasonlóképp létrehozhatóak új tulajdonkategóriák is (5.3. ábra). Ebben az esetben egy űrlapot kell kitölteni a Raccoon platformján. Új tulajdonkategória létrehozásához meg kell adnia egy nevet és egy rövid leírást. A kategória azonosítója automatikusan



5.2. ábra. Példa értékű adásvételi szerződés szerkesztése Google Docsszal

5.3. ábra. Egy új tulajdonkategória (autó) hozzáadása a Raccoon platformján

kitöltődik a kategória neve alapján, de szerkeszthető. Az összes korábban létrehozott kategóriát megtekinthetjük a „Tulajdonkategóriák” menüpont alatt. Új mezőt is hozzáadhatunk egy meglévő tulajdonkategóriához, ehhez az előbb említett menüben ki kell választani egy, majd megnyomva a „Kategória szerkesztése” gombot elénk tárul egy menü, ahol új mezőket adhatunk hozzá a kiválasztott tulajdonkategóriához. A következő paramétereket tudjuk megadni egy ilyen mezőnek: a típusa, prioritása (magasabb priorítás hamarabb jelenik meg), neve, egy hosszabb leírás/magyarázat (nem kötelező), sugó szöveg (nem kötelező), helyettesítő karakterlánc a dokumentumban (ennek a mezőnek a .docx sablonban is szerepelnie kell), minimális hossz (megkötés, nem kötelező), illetve maximális hossz (megkötés, szintén nem kötelező). Ha a hossz helyére –1-et írunk, az

azt jelenti, hogy a hossz nincs meghatározva. A 5.4. ábrán egy ilyen űrlapot töltöttünk ki egy új mező létrehozására.

Mező hozzáadása

Mező típusa

Karakterlánc (szavak, betűk) ▼

Mező prioritása (magasabb prioritás hamarabb jelenik meg)

10

Mező neve

Márka

Hosszabb leírás/magyarázat (nem kötelező)

Az autó márkája

Sugó szöveg (nem kötelező)

car_brand

Helyettesítő karakterlánc (a dokumentumban)

Minimális hossz (megkötés, nem kötelező)

-1

Maximális hossz (megkötés, nem kötelező)

-1

Mező hozzáadása

5.4. ábra. Egy új mező (márka) hozzáadása a Raccoon platformján

A szerződésekhez lehet ügyvédek is hozzáadni, az „Ügyvéd hozzáadása” menüpont alatt, amely email cím alapján történik. Ezt szintén csak adminisztrátori jogkörrel lehetséges.

Az átlagfelhasználó számára rendkívül fontos, hogy könnyen áttekinthető módon kezelje szerződéseit és tulajdonait. Az alkalmazás lehetővé teszi számára, hogy listázza az összes szerződését és tulajdonát, amelyekben könnyedén kereshet név vagy típus alapján.

Az alkalmazás továbbfejlesztésével lehetőséget adhatunk az átlagfelhasználónak arra is, hogy új szerződéseket vigyen be a rendszerbe. Ezáltal egyszerűen rögzítheti és kezelheti új üzleti vagy személyes szerződéseit a platformon keresztül. Az új szerződések felvitelekor a felhasználó részletes információkat adhat meg, például a szerződés leírását, a felek nevét és az érvényességi időtartamot. Ez segíti az átlagfelhasználót abban, hogy a szerződések könnyen azonosíthatók és kezelhetők legyenek.

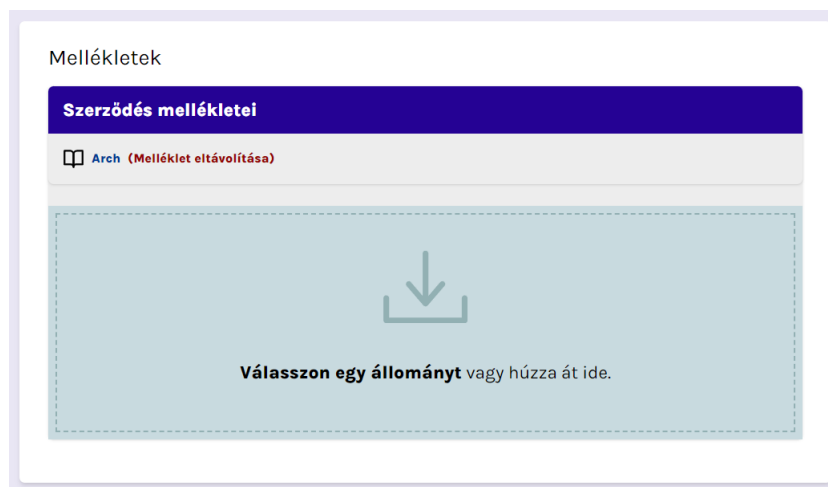
Az aláírás és letöltés funkció nagyban megkönnyíti az átlagfelhasználó életét. Az alkalmazás lehetővé teszi, hogy a felhasználók elektronikusan aláírják a szerződéseket, így elkerülve a hagyományos papíralapú eljárásokat és a helyhez kötöttséget. Az aláírás után a felhasználók letölthetik a szerződéseket PDF formátumban, így könnyedén megőrizhetik és megoszthatják őket más féllel.

A szerződések aláírása digitális formában történik, erre lehetőséget ad a platform is. Az aláírásra való felkérést a 5.5. ábra szemlélteti.



5.5. ábra. A szerződés digitálisan való aláírásra felhívó ablak

A felhasználók képesek a szerződések mellé csatolmányokat is feltölteni. Erre kényelmes megoldást biztosít az alkalmazás drag-and-drop formájában. Illetve, a felületre rákattintva az operációs rendszer fájlkiválasztó ablakát is megjeleníti. Ez látható a 5.6. ábrán. Abban az esetben, hogyha a felhasználó nem szeretné digitálisan aláírni a szerződést, kiválaszthatja a másik opciót is: az egyszerű aláírás lehetőségét. Az egyszerű aláírás segítségével nem kerül rá konkrétan a szerződésre a felhasználó kézírása, helyette a felhasználó neve jelenik meg.



5.6. ábra. A szerződésekhez különböző csatolmányokat lehet hozzáadni

Ezek a fejlesztések hozzájárulnak az alkalmazás hatékonyságához és felhasználóbarát jellegéhez, lehetővé téve az átlagfelhasználó számára, hogy egyszerűen kezelje, keresse és aláírja a szerződéseit, valamint megőrizze fontos tulajdonainak nyilvántartását egy helyen.

Végezetül pedig a weboldalt elláttuk többnyelvűsítéssel is, így a felhasználók könnyedén válthatnak a különböző nyelvek között egy egyszerű kattintással. Ez lehetővé teszi,

hogy az alkalmazás használata még inkább személyre szabható legyen, és a felhasználók anyanyelvükön vagy preferált nyelvükön használhassák azt.

Emellett a platform elérhetővé tette a sötét téma opciót is. Ez lehetővé teszi a felhasználóknak, hogy a hagyományos világos háttér helyett a sötét témát válasszák, ami kevésbé megterhelő a szemüknek és energiatakarékosabb lehet, különösen alacsony fényviszonyok esetén.

Ezek az újítások további funkcionalitást és kényelmet biztosítanak az felhasználók számára, lehetővé téve számukra, hogy könnyedén váltogassanak nyelvek között, és kiválasszák az optimális megjelenést a számukra. Az alkalmazásunk így még rugalmasabbá válik, és az egyéni preferenciákhoz igazodva személyre szabott felhasználói élményt nyújt.

6. fejezet

Mérések

A webes alkalmazások kitelepítése hagyományosan szerverek felállításával, bekonfigurálásával és kezelésével járt, viszont a felhőalapú technológiák rohamos fejlődése egy alternatív megközelítést vezetett be: a *serverless* megközelítéseket. Ennek a fejezetnek az a célja, hogy összehasonlítsa a *serverless* telepítések teljesítmény- és költségmetrikáit a hagyományos szerver telepítésekével.

A vizsgálat megvalósítása érdekében a *Project Raccoon*-nak két változatát hoztuk létre: az egyiket Docker segítségével telepítettük ki, ami a hagyományos szerveres megközelítést képviseli, a másikat pedig *serverless* technológiák segítségével, konkrétan a Vercel használatával.

A teljesítménymutatók, például a válaszidők, az erőforrás-kihasználás, valamint a költségmutatók körbejárásával a fejezet célja, hogy rávilágítson ezeknek a telepítési stratégiáknak az előnyeire és korlátaira.

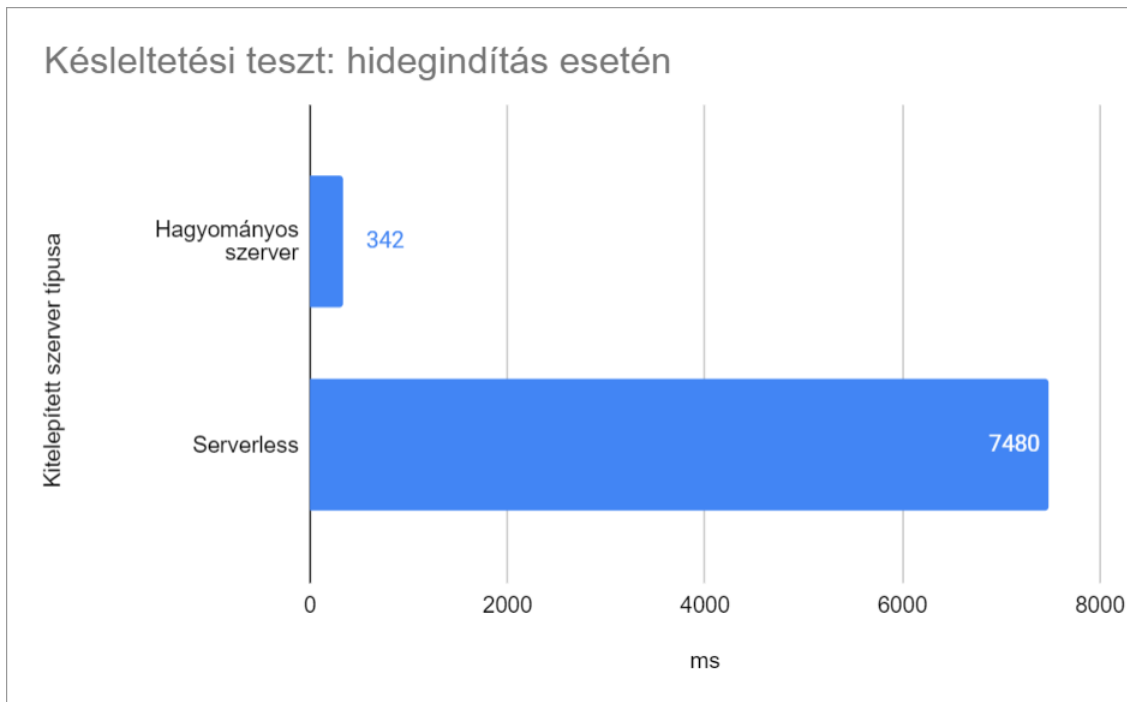
Olyan információkat szeretnénk mutatni, melyek informálhatják a döntéshozatali folyamatot a fejlesztők számára a kitelepítést illetően.

Az összes mérést Marosvásárhelyen készítettük, egy AMD Ryzen 5 5600H-es számítógépen, 16 GB memóriával és 1000 megabites hálózati sávszélességgel. A hagyományos szerver Németországban van kitelepítve a Hetzner szolgáltatónál: egy CPX11 szerverről van szó.

A Docker-es megvalósítás elérhető a következő linken, illetve letölthető a diplomadolgozat mellékleteiből is: <https://raccoon.tohka.us>

A Vercel-es (*serverless*) megvalósítás elérhető a következő linken: <https://raccoon.sallai.me>

6.1. Késleltetési teszt: hidegindítás esetén

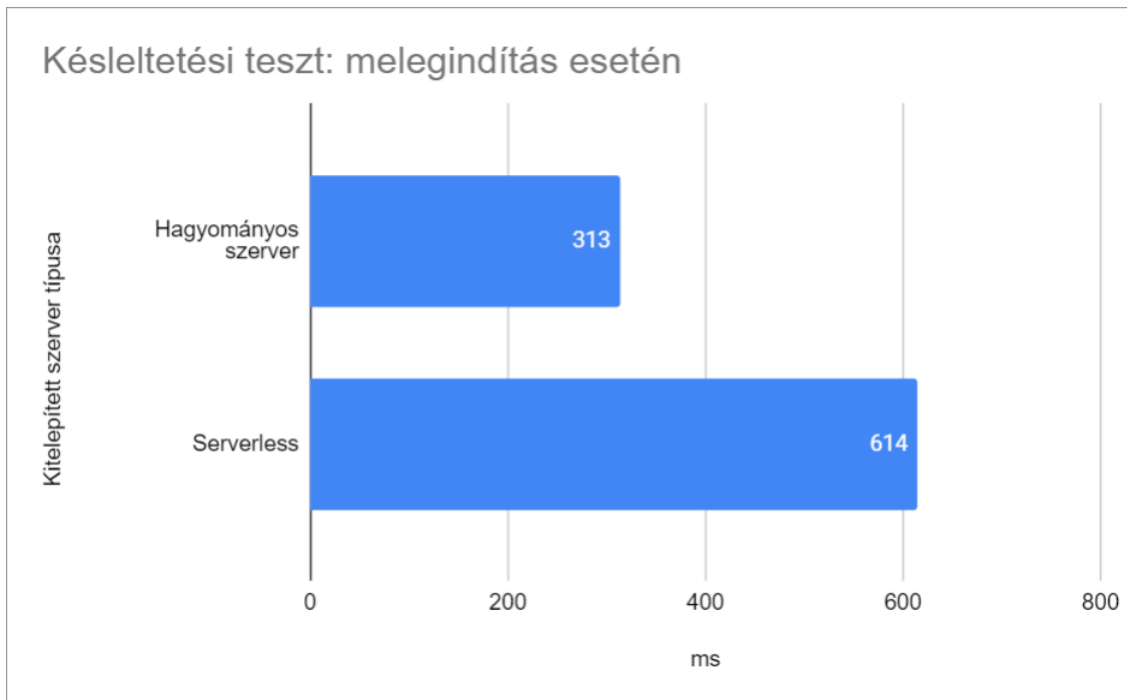


6.1. ábra. Késleltetési teszt: hidegindítás esetén

A 6.1. ábrán látható a késleltetési teszt eredményei hidegindítás esetén. A tesztet 10-szer hajtottuk végre. A teszt során egy Python program segítségével lemértük, mennyi időbe kerül, hogy válaszoljon a weboldal. Az eredményeket átlagoltuk. A futtatások között vártunk 10 percet, annak érdekében, hogy a serverless környezetben futtatott alkalmazás kikerüljön a meleg futtatási környezetéből.

Látható, hogy a serverless weboldalnak jelentősen több időbe kerül válaszolnia a hidegindítás során a kérésekre. Ez azzal magyarázható, hogy a felhőben a szerver le kell töltsen a futtatás előtt a futtatandó függvényeket. Ha fontos, hogy mindig egyből elérhető legyen az alkalmazásunk, válasszuk a hagyományos szerveret.

6.2. Késleltetési teszt: melegindítás esetén



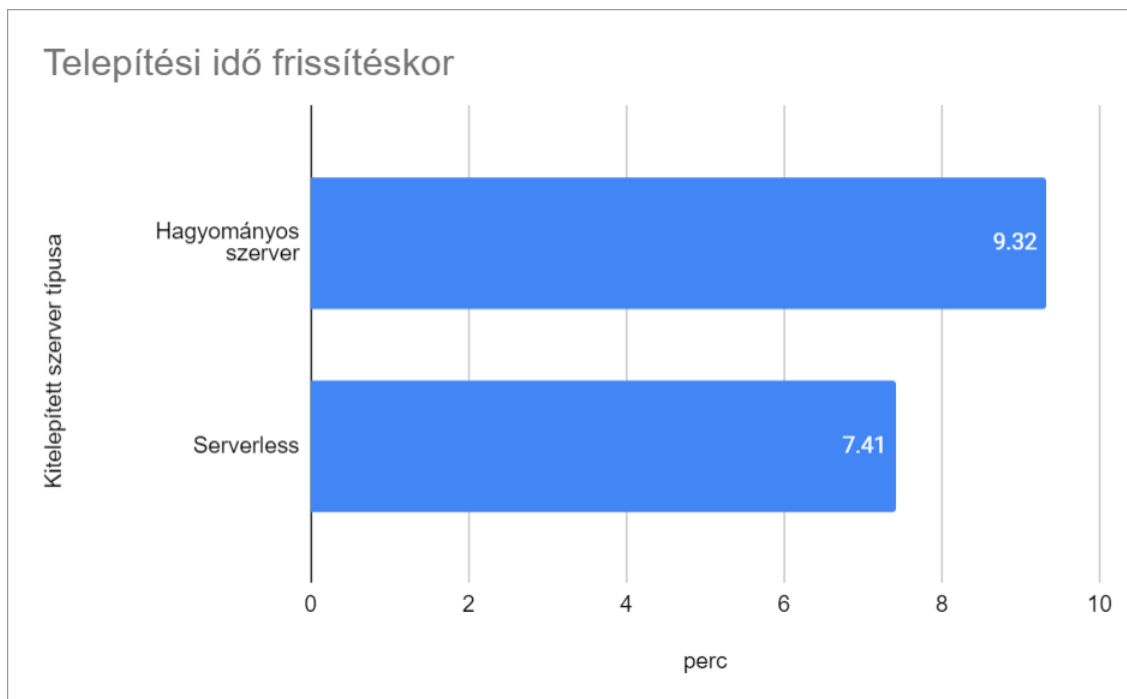
6.2. ábra. Késleltetési teszt: melegindítás esetén

A 6.2. ábrán látható a késleltetési teszt eredményei melegindítás esetén. Hasonlóképpen az előző tesztet, egy Python program segítségével lemértük, hogy mennyi időbe kerül az alkalmazás válasza a kéréseinkre. A weboldal főlapját kértük el, hitelesítés nélkül. A tesztet 100-szor futtattuk le, egymás után, várakozás nélkül. Az első 5 eredményt elvetettük, a többi átlagoltuk.

Ha összehasonlítjuk az eredményeket a 6.1. ábrával, észrevehetjük, hogy nem létezik különösebb különbség a késleltetés szempontjából a meleg és hideg indítások során a hagyományos szervereknél.

A 6.2 ábrát nézve megfigyelhetjük, hogy bár a hagyományos szerver Németországban van kitelepítve, a hagyományos szerver kevesebb késleltetésben szenved mint a serverless megoldás.

6.3. Telepítési idő frissítéskor



6.3. ábra. Telepítési idő frissítéskor

A 6.3. ábrán látható a telepítési teszt eredményei frissítéskor. A Vercel platformról a legutolsó 5 telepítés idejét vettük a serverless adatoknak, míg a hagyományos szerverek esetében lefuttattuk ötször a Docker buildelési és telepítési folyamatát. A telepítési időket átlagoltuk.

A serverless platform majdnem két egész perccel gyorsabb a hagyományos szervernél. A hagyományos szerver telepítésének lassúsága magyarázható azzal, hogy a Hetzneres szerver, amelyre telepítettük a rendszert, csupán egy CPX11 instance: két CPU core-al rendelkezik, ami nem segíti elő egyáltalán a párhuzamos telepítési folyamatot.

Ha gyorsabb telepítést szeretnénk, akkor mindenképp válasszunk serverless megoldást.

6.4. Költségelemzés havonta: alacsony használati előrejelzés

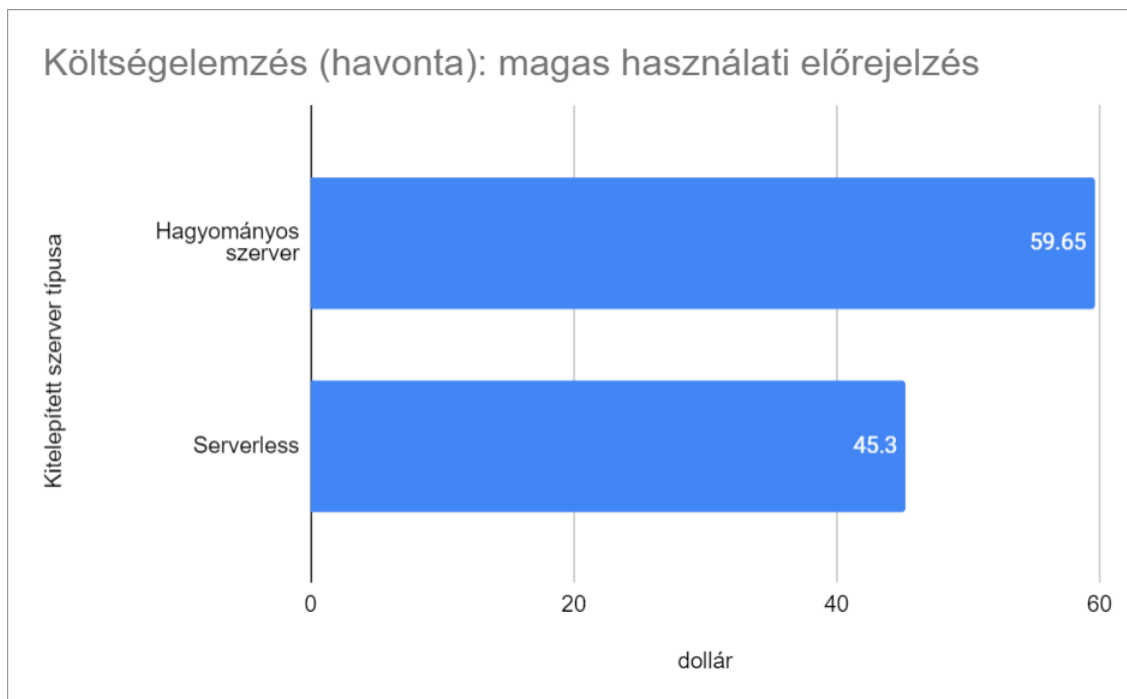


6.4. ábra. Költségelemzés havonta: alacsony használati előrejelzés

A 6.4. ábrán látható a költségelemzés alacsony használati előrejelzésre. A költségelemzésünk egyszerű volt ebben az esetben: a hagyományos szervernek egy Hetzner CPX11 szervert választottunk, míg a serverless megoldásnak a Vercel ingyenes *free tierjét* választottuk.

Mivel a Vercel teljesen ingyenesen használható 500,000 kérésig, ezért alacsony használat esetében teljesen ingyenes. A hagyományos szerver futtatása viszont pénzbe kerül: pontosan 4.73 dollárba. Ha nagyon kevés felhasználónk van, nincs értelme egy egész szervert kibérelni.

6.5. Költségelemzés havonta: magas használati előrejelzés



6.5. ábra. Költségelemzés havonta: magas használati előrejelzés

A 6.5. ábrán látható a költségelemzés magas használati előrejelzésre. A költségelemzésünk itt már komplikáltabb. Hagyományos szervernek a Hetzner CPX51 szerverét választottuk. Ennek költségét nagyon könnyű kiszámítani – egyetlen összetevőről van szó.

Serverless kontextusban sokkal bonyolultabbak a számítások. A serverless világban kell fizetni a programozók számáért is, a kérések számáért, illetve a sávszélességért is. 3 millió kérésben gondolkodtunk, 2 programozóban és 1 terabyte sávszélességben. Mivel egy felhasználó 20 dollárba kerül havonta, illetve 1 millió után minden millió kérés után 2.65 dollárt kér a Vercel, ezért a végeredmény 59.65 dollár.

Ha nem szeretnénk túlkomplikálni az életünket hasonló számítgatásokkal, jobban megéri a hagyományos szerver, amelynek futtatási költségei mindig konstansok maradnak.

Összefoglaló

A felgyorsult digitalizáció rengeteg újítást hoz a modern ember életében, beszivárog a mindennapjainka, legyen szó munkahelyről, oktatásról vagy épp egyéb területről.

A Project Raccoon a digitalizáció által nyújtott lehetőségeket kihasználva továbbfejleszti az ügyintézési folyamatokat, segítve ezzel a modern ember mindennapjait. A platform rugalmas és felhasználóbarát felülete révén könnyen kezelhető, és a felhasználók számára egyszerűbbé teszi az adminisztratív feladatokat.

Ez a rendszer számos előnyt kínál a felhasználók számára, akik így időt és erőforrásokat takaríthatnak meg azáltal, hogy a hagyományos papíralapú folyamatokat digitális formába helyezik át. Segítségével az emberek könnyedén intézhetik ügyeiket otthonról, anélkül, hogy személyesen kellene felkeresniük különböző hivatalokat vagy intézményeket. Az online felületen elérhetővé válnak az igazolások, űrlapok, dokumentumok, amelyeket könnyedén kitölthetnek. Ezenkívül a platform lehetővé teszi a dokumentumok elektronikus aláírását, amely hitelesített módon helyettesíti a hagyományos, papíralapú aláírást. Ezáltal a felhasználók biztonságosan és megbízhatóan intézhetik ügyeiket az online térben.

A mérésekből bebizonyosodott, hogy mind a hagyományos szerver alapú kitelepítéseknek, mind a *serverless* technológiákkal kitelepített környezeteknek még mindig van létjogosultságuk. Attól függően, hogy mekkora projektet készítünk, és milyen gyakran fogják használni a felhasználók, kell döntenünk a hasonló telepítési kérdésekről.

GitHub projekt linkje

A projekt forráskódja elérhetően a következő linken: <https://github.com/darktohka/project-raccoon>.

Ábrák jegyzéke

| | |
|--|----|
| 2.1. A Next.js logója | 13 |
| 2.2. A szerveroldali renderelés magyarázata | 14 |
| 2.3. A Node.js logója | 15 |
| 2.4. A React logója | 16 |
| 2.5. A Tailwind CSS logója | 16 |
| 2.6. A TypeScript logója | 19 |
| 2.7. A Docker Desktop kinézete | 20 |
| 2.8. A Cloudflare logója | 21 |
| 2.9. A Google Cloud logója | 22 |
| 2.10. A TypeORM logója | 22 |
| 2.11. A MariaDB logója | 23 |
| 2.12. A PostgreSQL logója | 24 |
| 2.13. A Google Docs logója | 25 |
| 2.14. Az MDX logója | 26 |
| 2.15. A Zillow főoldala | 29 |
| 2.16. A LawDepot főoldala | 30 |
| | |
| 4.1. A Project Raccoon architektúrája, felülnézetből | 49 |
| 4.2. Tradicionális kliens-szerver architektúra | 54 |
| 4.3. A <i>Project Raccoon</i> legegyszerűbb kliens-szerver kivitelezése | 56 |
| 4.4. A <i>Project Raccoon</i> részletesebben kidolgozott telepítésének kivitelezése | 57 |
| 4.5. Kliens-szerver architektúra megvalósítása serverless technológiával | 59 |
| 4.6. A szerződések létrehozásának activity diagramja | 65 |
| 4.7. A végleges szerződések kigenerálásának szekvencia diagramja | 67 |
| 4.8. A Project Raccoon belépési felületének drótváza | 68 |
| 4.9. A Project Raccoon regisztrációs felületének drótváza | 69 |
| 4.10. A Project Raccoon <i>dashboard</i> jának drótváza | 70 |
| 4.11. A Project Raccoon tulajdonokat (kategóriánként) megjelenítő felületének drótváza | 71 |

| | |
|---|----|
| 4.12. A Project Raccoon szerződéseket listázó felületének drótváza | 72 |
| 4.13. A hitelesített felhasználók use case diagramja | 72 |
| 4.14. A speciális felhasználók use case diagramja | 74 |
| 5.1. A szerződéssablonok létrehozásának súgója | 75 |
| 5.2. Példa értékű adásviteli szerződés szerkesztése Google Docsszal | 77 |
| 5.3. Egy új tulajdonkategória (autó) hozzáadása a Raccoon platformján | 77 |
| 5.4. Egy új mező (márka) hozzáadása a Raccoon platformján | 78 |
| 5.5. A szerződés digitálisan való aláírásra felhívó ablak | 79 |
| 5.6. A szerződésekhez különböző csatolmányokat lehet hozzáadni | 79 |
| 6.1. Késleltetési teszt: hidegindítás esetén | 82 |
| 6.2. Késleltetési teszt: melegindítás esetén | 83 |
| 6.3. Telepítési idő frissítéskor | 84 |
| 6.4. Költségelemzés havonta: alacsony használati előrejelzés | 85 |
| 6.5. Költségelemzés havonta: magas használati előrejelzés | 86 |

Táblázatok jegyzéke

| | |
|---|----|
| 3.1. Felhasználói követelmények vendég felhasználók számára | 33 |
| 3.2. Felhasználói követelmények hitelesített felhasználók számára | 35 |
| 3.3. Felhasználói követelmények ügyvéd felhasználók számára | 37 |
| 3.4. Felhasználói követelmények rendszergazda felhasználók számára | 38 |
| 3.5. Funkcionális követelmények vendég felhasználók számára | 40 |
| 3.6. Funkcionális követelmények hitelesített felhasználók számára | 42 |
| 3.7. Funkcionális követelmények ügyvéd felhasználók számára | 44 |
| 3.8. Funkcionális követelmények rendszergazda felhasználók számára | 45 |
| 3.9. Külső követelmények (<i>external requirements</i>) | 46 |
| 3.10. Termékkövetelmények (<i>product requirements</i>) | 47 |
| 3.11. Szervezési követelmények (<i>organizational requirements</i>) | 48 |
| 4.1. Mappaszerkezet magyarázata | 52 |
| 4.2. A legegyszerűbb kivitelezés előnyei és hátrányai | 57 |
| 4.3. A kidolgozott kivitelezés előnyei és hátrányai | 58 |
| 4.4. A serverless kivitelezés előnyei és hátrányai | 60 |
| 4.5. Felhasználási eset: Szerződés letöltése | 73 |

Irodalomjegyzék

- [1] „We measured the SSR performance of 6 JS frameworks – here’s what we found.,” *Enterspeed*. Letöltve: 2023-07-02. <https://www.enterspeed.com/blog/we-measured-the-ssr-performance-of-6-js-frameworks-heres-what-we-found>.
- [2] A. Wathan, „CSS Utility Classes and "Separation of Concerns",” Letöltve: 2023-07-02. <https://adamwathan.me/css-utility-classes-and-separation-of-concerns>.
- [3] Z. Gao, C. Bird, and E. T. Barr, „To type or not to type: quantifying detectable bugs in JavaScript,” in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pp. 758–769, IEEE, 2017.
- [4] „Measuring the Horizontal Attack Profile of Nabla Containers,” Letöltve: 2023-07-02. <https://blog.hansenpartnership.com/measuring-the-horizontal-attack-profile-of-nabla-containers>.
- [5] „Data localization suite,” Letöltve: 2023-07-02. <https://developers.cloudflare.com/data-localization>.
- [6] „Interoperability with other storage providers,” Letöltve: 2023-07-02. <https://cloud.google.com/storage/docs/interoperability>.
- [7] N. Provos and D. Mazieres, „Bcrypt algorithm,” in *USENIX*, 1999.
- [8] „e-stamping and where to e-stamp documents,” Letöltve: 2023-07-02. <https://www.iras.gov.sg/taxes/stamp-duty/for-shares/basics-of-stamp-duty-for-shares/e-stamping-and-where-to-e-stamp-documents>.
- [9] Nemzeti Infokommunikációs Szolgáltató, „Azonosításra Visszavezetett Dokumentumhitelesítés (AVDH),” Letöltve: 2023-07-02. <https://www.nisz.hu/hu/avdh-âĀĖ-azonosÃĖsra-visszavezetett-dokumentumhitelesÃĖs>.

- [10] M. E. Mustafa, „Load balancing algorithms round-robin (rr), leastconnection, and least loaded efficiency.,” *Computer Science & Telecommunications*, vol. 51, no. 1, 2017.
- [11] A. Kumar and D. Anand, „Study and analysis of various load balancing techniques for software-defined network (a systematic survey),” in *Proceedings of International Conference on Big Data, Machine Learning and their Applications: ICBMA 2019*, pp. 325–349, Springer, 2021.
- [12] T. Tsai, „Getting started with Docker for Arm on Linux,” Letöltve: 2023-07-02. <https://www.docker.com/blog/getting-started-with-docker-for-arm-on-linux>.
- [13] G. McGrath and P. R. Brenner, „Serverless computing: Design, implementation, and performance,” in *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*, pp. 405–410, IEEE, 2017.
- [14] J. Nielsen, „Ten usability heuristics,” 2005.
- [15] S. Van Belleghem, „The self service economy,” 2013.