SZAKDOLGOZAT FELADAT

**Pelyhe Ádám (U0X77G)**

Műszaki informatikus hallgató részére

Alkalmazásfejlesztés blokklánc alapú CBDC ökoszisztémában

A digitális jegybankpénz (Central Bank Digital Currency, CBDC) a következő években várhatóan meghatározó változásokat hoz majd egyfelől a pénzügyi szolgáltatások, másfelől az ezekre épülő alkalmazások területén. Bár önmagában a jegybankpénz könyvelése sokféle módon megvalósulhat, a világban folyó kutatások és prototípus fejlesztések egyértelműen azt mutatják, hogy várhatóan (konzorciális) blokklánc fölötti megvalósítások fognak elterjedni.

Ezen alkalmazások fejlesztése a hagyományos banki alkalmazásoktól eltérő szemléletet igényel és részben különböző technológiákra támaszkodik (tárcák – walletek, digitális azonosítók, okosszerződések, tokenek kezelése, stb). A hallgató feladata egy olyan alkalmazás követelményspecifikációja, tervezése és megvalósítása, mely példát ad a digitális jegybankpénz alkalmazási lehetőségére üzleti/ipari környezetben, egyben bemutatja fejlesztés során felmerülő döntési lehetőségeket, ideértve a CBDC integráció logikáját.

A hallgató feladatának a következőkre kell kiterjednie:

* Mutasson példát CBDC alapú alkalmazásokra (ill. tágabb értelemben alkalmazási területekre) a nemzetközi szakirodalom alapján.
* Tervezzen meg alkalmazást (logisztika, biztosítás stb. iparágak valamelyikében), melyben a digitális jegybankpénz alapú elszámolás előnyei érvényesülhetnek. Mutassa be a tervezés szempontjait ill. a megvalósítás során eldöntendő kérdéseket.
* Valósítsa meg az alkalmazást, és értékelje ki az elkészült szoftvert a különböző érintettek szempontjai alapján.
* A tervezés során kiemelt szempont, hogy az elkészült alkalmazás a BME Mit tanszék CBDC prototípusához illeszkedjen.

**Tanszéki konzulens:** Dr. Gönczy László, docens

Budapest, 2022. október 9.

…………………………..

Dr. Dabóczi Tamás

tanszékvezető

egyetemi tanár, DSc

C:\Users\szarnyasg\Downloads\bme_logo_nagy.eps

**Budapesti Műszaki és Gazdaságtudományi Egyetem**

Villamosmérnöki és Informatikai Kar

Méréstechnika és Információs Rendszerek Tanszék

Alkalmazásfejlesztés blokklánc alapú CBDC ökoszisztémában

Készítette

Pelyhe ÁdámKonzulens

Dr. Gönczy László

2022

Tartalomjegyzék

[SZAKDOLGOZAT FELADAT 1](#_Toc121346036)

[Összefoglaló 6](#_Toc121346037)

[Abstract 7](#_Toc121346038)

[1. Bevezetés 8](#_Toc121346039)

[1.1. Témaválasztás indoklása 8](#_Toc121346040)

[1.2. Alkalmazás célkitűzései 9](#_Toc121346041)

[2. Irodalomkutatás, technológiák, hasonló alkalmazások bemutatása 10](#_Toc121346042)

[2.1. Blokklánc bemutatása 10](#_Toc121346043)

[2.2. CBDC bemutatása 12](#_Toc121346044)

[2.3. Példák blokklánc alapú biztosításokra 15](#_Toc121346045)

[2.4. További technológiák bemutatása 18](#_Toc121346046)

[2.4.1. Flutter 18](#_Toc121346047)

[2.4.2. Ethereum 18](#_Toc121346048)

[2.4.3. Interplanetary File System 19](#_Toc121346049)

[2.4.4. Node.js 20](#_Toc121346050)

[2.4.5. WalletConnect 21](#_Toc121346051)

[2.4.6. Web3.js, Ether.js 21](#_Toc121346052)

[2.4.7. Hyperledger Fabric 22](#_Toc121346053)

[3. Rendszertervezés 24](#_Toc121346054)

[3.1. Architektúra 24](#_Toc121346055)

[3.1.1. Felhasználói felület 24](#_Toc121346056)

[3.1.2. Biztosítási folyamat blokkláncon 25](#_Toc121346057)

[3.1.3. Elosztott dokumentumtár 25](#_Toc121346058)

[3.1.4. Eseménykezelő 25](#_Toc121346059)

[3.1.5. Payment gateway – fizetési átjáró 25](#_Toc121346060)

[3.1.6. CBDC réteg 26](#_Toc121346061)

[3.2. Használati esetek 26](#_Toc121346062)

[3.2.1. Ügyfélhez tartozó használati esetek 26](#_Toc121346063)

[3.2.2. Biztosító céghez tartozó használati esetek 27](#_Toc121346064)

[3.3. Követelmények 29](#_Toc121346065)

[3.3.1. Funkcionális követelmények 29](#_Toc121346066)

[3.3.2. Nemfunkcionális követelmények 30](#_Toc121346067)

[3.4. Főbb funkciók bemutatása 30](#_Toc121346068)

[3.4.1. Havidíj befizetés 31](#_Toc121346069)

[3.4.2. Kárbejelentés 32](#_Toc121346070)

[4. Önálló munka bemutatása 33](#_Toc121346071)

[4.1. Felhasználói felület 33](#_Toc121346072)

[4.1.1. Tárca csatlakoztatása 33](#_Toc121346073)

[4.1.2. Biztosítás kötése 34](#_Toc121346074)

[4.1.3. Főoldal 36](#_Toc121346075)

[4.1.4. Biztosítási kategória megváltoztatása 37](#_Toc121346076)

[4.1.5. Kárbejelentés 37](#_Toc121346077)

[4.1.6. Károk elbírálása 38](#_Toc121346078)

[4.2. Ethereum okosszerződések 38](#_Toc121346079)

[4.2.1. Biztosítást kezelő okosszerződés 39](#_Toc121346080)

[4.2.2. Kárbejelentést kezelő okosszerződés 40](#_Toc121346081)

[4.2.3. Biztosítási tokent kezelő okosszerződés 42](#_Toc121346082)

[4.3. Elosztott dokumentumtár 43](#_Toc121346083)

[4.3.1. Állomány tárolására szolgáló végpont 44](#_Toc121346084)

[4.3.2. Állomány lekérdezésére szolgáló végpont 45](#_Toc121346085)

[4.4. Eseménykezelő 46](#_Toc121346086)

[4.5. Fizetési átjáró 48](#_Toc121346087)

[4.6. CBDC réteg 50](#_Toc121346088)

[5. Összefoglalás 51](#_Toc121346089)

[5.1. Alkalmazás értékelése 51](#_Toc121346090)

[5.2. Továbbfejlesztési lehetőségek 52](#_Toc121346091)

[Irodalomjegyzék 53](#_Toc121346092)

[Függelék 57](#_Toc121346093)

Hallgatói nyilatkozat

Alulírott Pelyhe Ádám, szigorló hallgató kijelentem, hogy ezt a szakdolgozatot meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző(k), cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy hitelesített felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Kelt: Budapest, 2022. 12. 07.

Pelyhe Ádám

# Összefoglaló

Szakdolgozatom során egy olyan alkalmazás fejlesztése volt a célom, amely megoldást nyújthat egy, a világban aktuálisan felmerülő problémára. Továbbá fontos volt még számomra, hogy az elkészült prototípus minél több korszerű technológiát használjon.

Úgy gondolom, hogy manapság egy nagy problémát jelent a földgáz világpiaci árának növekvő tendenciája. Egy 2022-es statisztika alapján a háztartásban történtő energiafogyasztás több, mint felét a szobák felfűtése teszi ki [1], mely nagy százalékban földgázzal történik. Ezek alapján egyre megterhelőbb lehet a családok számára a fűtési berendezések nem várt meghibásodása során keletkező többletköltségek finanszírozása.

Az alkalmazásom erre a problémára kínálhat megoldást, a gázfűtési berendezések biztosításával. Mivel a biztosítók egyre nagyobb előszeretettel támaszkodnak a blokklánc technológiára, ezért a fejlesztés során a tanszék és a Magyar Nemzeti Bank által közösen készített digitális jegybankpénz mellett blokklánc technológiát is használtam. Így jött létre az elkészült termék, mely használata során a felhasználók több különböző, rugalmasan változtatható biztosítási konstrukció közül választhatnak és ahol a havidíj befizetése CBDC formájában történik.

Bár a termék nincs kész éles környezetben történő használatra – tekintve például, hogy a biztosításkötés jogi oldala a feladat keretein túl mutat – úgy gondolom, nagyszerűen bemutatja a korszerű, elosztott adatbázisokban rejlő lehetőségeket.

# Abstract

The main goal of my thesis was to develop an application, which can provide a solution to a currently occurring problem in the world. Furthermore, using modern technologies in the prototype was important for me as well.

As far as I am concerned, the increasing tendency of the price of natural gas is a serious problem nowadays. According to a survey made in 2022, space heating is responsible for more than 60 percent of households’ energy consumption [1], which often happens with the use of natural gas. Thus, it might be harder financially for the families to pay out the plus fees if a gas heating device goes out of service.

My application can provide a solution to this problem by insuring the gas heating devices. Because insurance companies nowadays use blockchain more likely, during the developing process I took benefit of this technology. Also, I used a central bank digital currency (CBDC) created by the MIT department and the Hungarian National Bank. With these technologies, I built an application whereby the users can choose from multiple insurance designs, which can be flexibly changed afterwards, and where they pay their monthly fees with CBDCs.

In my opinion, this app highlights the potential of modern, decentralized databases, even though it is not production ready yet.

# Bevezetés

## Témaválasztás indoklása

A 18. század közepén az első biztosítócégek egyike volt az 1752-ben megalapult *Philadelphia Contributionship,* amely által kínált biztosítás a tűzesetben károsult házakra terjedt ki. A biztosítási folyamat a következő módon zajlott: szakemberek kimentek a kár helyszínére és felülvizsgálták, hogy jogosult-e az ügyfél kártérítésre, mely során figyelembe vették az épület kockázatértékelését is [2]. Ez a módszer a 21. századból is ismerős lehet, sok biztosítócég ma is ily módon jár el. A technológia fejlődésével megnőtt az igény a változásra a biztosítások világában is.

Ezzel párhuzamosan az elmúlt évtized során egyre nagyobb figyelmet kap az informatika világában az elosztott tárolásra való törekvés. Az elosztott rendszerek segítségével nemcsak az adataink biztonságát tudjuk növelni, hanem adott esetben a hatékonyságra és a rendszer hibatűrésére is pozitív hatással lehet. A decentralizált adatbázisok egyik úttörője a blokklánc technológia is, mely mellett szól az előbb említett előnyökön kívül az automatizálhatósága is. A programok függvényei meghívhatók önvezérelt módon abban az esetben, ha egy bizonyos feltétel teljesül. A biztosítások világában ez egy nagyon pozitív tulajdonság, mellyel növelhető a transzparencia és a bizalom a biztosítócég és az ügyfél között. Ezek mellett sokkal flexibilisebb üzleti logikát definiálhatunk, mely szintén egy óriási előny lehet.

A technológia fejlődés elég sok ágazatra terjedt ki az elmúlt időben, ezek között van a pénzügyi szektor is, ahol megnőtt az igény a fizetőeszközeink digitalizálására. Ennek hatására egyre nagyobb figyelem jut az elosztott, blokkláncokon futó kriptovalutákra, melyek piaci kapitalizációja évről évre erősen növekszik [3]. Értékük azonban rendkívül kiszámíthatatlan, köszönhetően többek között a kereslet-kínálat folyamatos változásainak, kormányzati szabályzásoknak. A digitális jegybankpénz, vagyis a CBDC a magas volatilitásra kínál megoldást és kitűnően integrálható blokklánc technológiát használó alkalmazásokba.

Ezek alapján adott volt a technológia, valamint az, hogy a készülő applikáció valamilyen biztosítást fog megvalósítani. Mivel a fűtőberendezések és a fűtésre használt gáz ára is egyre inkább emelkedő tendenciát mutat nemzetközi viszonylatban, úgy gondoltam megfelelő ötlet lehet egy gázfűtési berendezések biztosítására szolgáló applikáció fejlesztése. A program valamennyire enyhítheti a fűtési berendezések nem várt hibái során felhalmozódó költségeket.

## Alkalmazás célkitűzései

Az alkalmazás egyik legfontosabb célkitűzése a lehető legnagyobb flexibilitás biztosítása az ügyfelek számára. Abban az esetben, ha például biztosítási kategóriát kívánnak váltani, rögtön és egyértelműen meg tudják tenni azt a kezelőfelület segítségével. Fontos ezen kívül, hogy minél kevesebb dolguk legyen a felhasználóknak a biztosítási szerződésük létrehozása és karbantartása során. Minden hónapban a havidíj befizetése legyen az egyetlen kötelességük.

Amellett, hogy szeretném a lehető legjobban redukálni az alkalmazáson belüli, havonta kötelezően elvégzendő műveleteket, természetesen egy fejlesztő számára fontos, hogy az általa készített applikációt önszántukból is minél többet használják az ügyfelek. Ennek érdekében egy olyan megoldást kívánok implementálni, amivel ösztönzöm a felhasználókat a napi használatra, valamilyen jutalomért cserében.

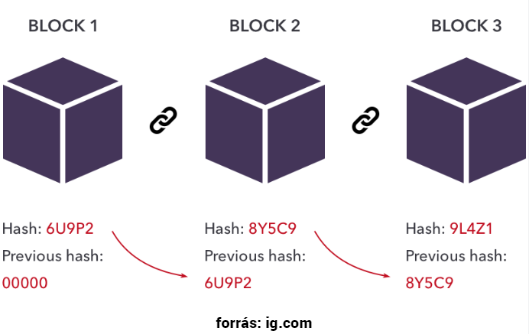
Az előbb említett célokon kívül szeretnék minél több korszerű technológiát is használni a fejlesztés során. Úgy gondolom, hogy ez egy kitűnő lehetőség az elosztott rendszerek mélyebb megismerésére.

# Irodalomkutatás, technológiák, hasonló alkalmazások bemutatása

## Blokklánc bemutatása

A technológiák bemutatását a blokklánc alapjaival kezdem. A blokklánc technológia egy elosztott adatbázis. Ennek értelmében nincs egy központi szerver, amely meghibásodása az alkalmazásunkat működésképtelenné teszi. Leggyakrabban a kriptovaluták világában hallhatunk róluk, azonban ezek a szakdolgozatomban nem szerepelnek. A decentralizált felépítésnek köszönhetően számos előny szól a technológia mellett, mint például a biztonság és automatizálhatóság.

Az egyik legfőbb különbség a centralizált adatbázisokkal szemben az adatok tárolásának struktúrája.



1. ábra: Adatok tárolásának struktúrája blokkláncon

Az 1. ábra látható a blokklánc hálózat adatstruktúrája. Minden blokk rendelkezik egy egyedi azonosítóval, másnéven úgynevezett hash-el, valamint az előző blokk azonosítójával is. Ez segíti elő a láncszerű elrendezést. Ezen kívül eltárolásra kerül a blokk keletkezésének időbélyege is. Ezek a tárolási egységek a hálózaton érvényesnek nyilvánított tranzakciókat tartalmazzák.

A hálózatban szereplő csomópontok különböző eszközök, melyek tárolják a hálózat állapotának – másnéven a főkönyv – másolatát, illetve elvégzik a tranzakciók felülvizsgálatát (*lásd 2. ábra*).



2. ábra: A blokklánc hálózatban részt vevő csomópontok

A csomópontoknak egymással kommunikálva szükséges elérniük az egyetértést a rendszer jelenlegi állapota felett, melyben segítségükre szolgálnak a különböző konszenzus mechanizmusok [4]. Ezek segítenek az együttműködésben, illetve a rendszer biztonságának megőrzésében. A biztonság egyik lefontosabb eleme az, hogy a hálózat 51%-át kell egy felhasználónak birtokolni azért, hogy megdöntse a konszenzust. Különféle konszenzus mechanizmusok léteznek, a két leghíresebb a Proof of Work, illetve a Proof of Stake protokoll.

A Proof of Work esetében a bányász csomópontok versenyeznek egy blokk létrehozásáért. Ebben az esetben az hozza létre a blokkot, akinek a számítógépe a leggyorsabban képes megoldani egy matematikai feladványt. Nagy hátránya az, hogy a nagy számítási kapacitással bíró bányászgépek exponenciálisan nagyobb eséllyel hozhatnak létre blokkokat, mint a kisebb teljesítményű eszközök, ezáltal ösztönözve a felhasználókat a magas energiafelhasználásra.

A Proof of Stake protokoll során azon felhasználók szállnak versenybe, akik előzetesen egy bizonyos mennyiségű összeget – mely például Ethereum hálózat esetében etherben mért – letétbe helyeztek. Minél nagyobb összeg van a letétben, annál nagyobb valószínűséggel hozhatnak létre blokkokat. Mivel annak kiválasztása, hogy egy adott időpontban ki változtathatja a blokklánc állapotát nagyjából véletlenszerű, ezáltal nagyobb eséllyel kapnak lehetőséget a kisebb számítási kapacitással rendelkező csomópontok a blokk megalkotására [5].

Természetesen a tranzakciók elfogadásában részt vevő csomópontok ezt nem ingyen teszik, hanem az előállított blokkok után megadott mennyiségű összeget kapnak, mely ezáltal motiválja őket a hálózat működésének és biztonságának fenntartására.

A blokkláncon eltárolt adatok megváltoztathatatlanok. Sok rendszer esetében lehetőség van programokat, úgynevezett okosszerződéseket is telepíteni a hálózatra. Ezek nagyon könnyen automatizálhatóak. Függvényei legtöbb esetben külső alkalmazásokból meghívhatók a web3 könyvtárak (*lásd 2.4.6 bekezdés*) segítségével. Az okosszerződések segítségével úgynevezett *tokeneket* hozhatunk létre. Ezeknek a két legfőbb típusa a helyettesíthető, úgynevezett *fungible token*ek, illetve a nem helyettesíthető, *non-fungible token*ek (vagy másnéven NFT-k). Ezeknek különböző szabványoknak kell megfelelniük, előbbi esetében ez az Ethereum hálózatán az ERC20 szabvány [6], míg az NFT-k esetében az ERC721-es szabvány [7].

## CBDC bemutatása

A CBDC az angol *Central Bank Digital Currency* rövidítése. Ahogyan arról neve is árulkodik, ez a központi bank által kibocsátott digitális jegybankpénz, mely mögött az adott ország valutája áll, így elősegítve annak stabilitását. Mivel a központi bank végzi a CBDC-k kibocsátását, ezért azonos feltételek vonatkoznak rá, mint a hagyományos pénznemekre. Alapvetően két típust különböztethetünk meg: a nagykereskedelmi, illetve kiskereskedelmi célokra kibocsátott CBDC-ket. Mind a kettő esetében a jegybank felel a kibocsátásért, azonban a nagykereskedelmi CBDC során ez valamilyen pénzintézet részére történik, mely felhasználhatja azt tartalék letét formájában. A gyakoribb eset a kiskereskedelmi célokra kibocsátott digitális jegybankpénz, mely közvetlen a végfelhasználókhoz, vagy kiskereskedőkhöz kerülhet. Segítségével csökkenthető a nyomtatási, illetve kezelési költség is.

Fontos, hogy a CBDC valutákat valamilyen elosztott főkönyvi hálózaton menedzselik, azonban ez nem kötelezően kell, hogy blokklánc legyen [8]. Ennek ellenére a blokklánc technológia kitűnő választásnak tűnhet erre a feladatra, hiszen pont azokra a problémákra kínál megoldást, amelyekre az emberek pénzének kezelése során szükség van: automatizálás, robusztusság és hatékonyság növelése, valamint a legfontosabb, a kellő biztonság nyújtása.

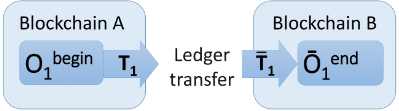
Előfordulhat olyan forgatókönyv, hogy az alkalmazásunkba történő CBDC integráció után két, vagy több különböző blokkláncnak szükséges egymással kommunikálni az inkonzisztens állapotok elkerülése végett. Erre a problémára több megoldás is létezik, az alkalmazásfejlesztés korai stádiumában két opció közül kellett választanom.

Az egyik lehetőség egy REST API[[1]](#footnote-2) fejlesztése, amely során a végpontokra küldött kérések segítségével kommunikálhatunk a CBDC-t megvalósító okosszerződéssel. Feltételezzük, hogy adott két blokklánc, melyek neve *A* és *B*. Az *A* láncon található egy okosszerződés, amelyben szükséges végrehajtani egy CBDC fizetést. A CBDC-t megvalósító okosszerződés a *B* blokkláncon található.

Ekkor az *A*-beli okosszerződésben keletkezik egy esemény, amelyet egy külön komponens web3 könyvtárak (*lásd 2.4.6 bekezdés*) segítségével hallgat. Amennyiben létrejön az esemény, tudjuk, hogy esedékes a CBDC-vel történő fizetés. Ekkor az eseménykezelő meghívja az előbb említett REST API-t, amely segítségével képes kommunikálni a *B* blokkláncon található CBDC szerződéssel. Ha az API hívás segítségével elindított tranzakció sikeres, úgy ez az *A* blokkláncon megerősítésre kerül, oly módon, hogy az eseménykezelő meghívja az erre szolgáló okosszerződés függvényt.

Az általam megvalósított megoldás ezt az elvet követi. Azért döntöttem emellett, mivel ennek segítségével az alkalmazásom független maradhat a CBDC fizetéstől, tekintve, hogy az API szabadon lecserélhető más, azonnali fizetést lehetővé tevő szolgáltatásra. A megoldásom részletesebb leírása a *4.4.* és a *4.5. bekezdésben* látható.

Van azonban egy másik megoldás is a blokkláncok összekötésére, ez az úgynevezett *bridging*. Ilyen *bridging* alapú megoldást kínál a Hyperledger Cactus alkalmazása is [9], melyen belül több tervezési minta is található, azonban a leggyakoribb esetekben a blokkláncok közötti mozgatás (*lásd 3. ábra 4. ábra*) a jellemző, így ezt szeretném bemutatni.



3. ábra: Egyirányú mozgatás *bridging* segítségével

A megoldás lényege, hogy az *A* blokkláncon található *token*eknek (O1) létezik egy B blokkláncon értelmezett megvalósítása is (). Abban az esetben, ha az O1 *token*t át kívánjuk mozgatni a *B* blokkláncra, úgy először az O1-etzárolni, vagy törölni kell. Amennyiben ez megtörtént, a -t létrehozzuk, ezáltal megvalósítva a mozgatást. Ugyanannak a *token*nek sosem létezik két reprezentációja a blokkláncokon.

A képen szöveg, aláírás, képernyőkép látható

Automatikusan generált leírás

4. ábra: Kétirányú mozgatás *bridging* segítségével

Amennyiben az üzleti logika szerint nem létezik dedikált forrás és cél blokklánc, hanem a *token*ek mozgathatóak mind a két irányba, úgy ez egy másik tranzakció során hasonló módon megtörténhet (lásd *4. ábra* - T2 tranzakció).

Jelenleg világszerte komoly erőfeszítéseket tesznek az országok a CBDC gazdaságba történő integrációjára [10]. Jelenleg 11 országban működik aktívan CBDC alapú infrastruktúra, melyek közül 10 Közép- és Észak-Amerikában található, illetve egy Nigériában.

A világon elsőként a Bahamai-szigeteken bocsátottak ki CBDC-t 2020 októberében. A Sand Dollar nevű jegybankpénz legfőbb motivációja a pénzmosás elleni védekezés volt. A fizetőeszköz bevezetése előtt egy évvel elkezdték felmérni az igényeket a lakosság részéről, amelyek pozitívnak bizonyultak, ezért döntöttek 2020 őszi bevezetés mellett [11]. Mivel a projektnek egyértelműen pozitív visszacsatolása volt, később egyre több közép-amerikai ország döntött a CBDC valuta bevezetése mellett, köztük például Jamaica is.

Az amerikai kontinensen kívül, a világon egyedülálló módon Nigéria is bevezette saját CBDC valutáját 2021 októberében, mely az eNaira nevet kapta. A kezdeti szakaszban körülbelül 1.21 millió amerikai dollárnak megfelelő értékű eNaira került kibocsátásra. 2021 decemberére több, mint 600 ezer digitális tárcát hoztak létre, melyek között mintegy 35 ezer tranzakció történt. A következő tíz évben a terv szerint 29 milliárd dollárt adhat hozzá az eNaira bevezetése az afrikai ország GDP-jéhez.

Fontosnak érzem emellett kiemelni a Méréstechnika és Információs Rendszerek Tanszék és a Magyar Nemzeti Bank közös projektjét, mely során létrejött egy CBDC prototípus, az eHuf. A kooperáció célja az autó lízingelés folyamatának megkönnyítése, felgyorsítása volt annak digitalizálásával. Mivel az elektromos autók vételére és lízingelésére a való életből vett példa alapján támogatás jár, ezért az elkészült eHuf szerződés mellett létrejött egy másik CBDC prototípus is, a greenHuf. Amennyiben a felhasználó elektromos autó lízingelése mellett döntött, úgy a kellő összeg befizetését megtehette a greenHuf tárcájából is, amelynek köszönhetően előre definiált mértékű támogatásban részesült.

Az általam készített alkalmazás során az eHuf valutát használom, mint fizetőeszköz. Emellett a CBDC okosszerződéshez készült fizetési átjáró is ezen projekt keretein belül jött létre, annak elkészítése sem az én munkám eredménye.

## Példák blokklánc alapú biztosításokra

A blokklánc technológia jelentős előnyei nagy hatással lehetnek a globális biztosítópiacra. Jól mutatja mennyire szívesen alkalmazzák a biztosítócégek a blokklánc technológiát napjainkban, hogy míg 2020-ban 177.85 millió dollárt tett ki a blokklánc technológia mérete a globális biztosítópiacon, ez a szám becslések alapján 2030-ra 39.56 milliárd dollárra nőhet [12].

Egy pár példával szeretném bemutatni, hogyan alkalmazzák a nagy biztosítócégek a blokklánc technológiát napjainkban. Megemlítek olyan eseteket is, amelyek ma már nem működnek aktív szolgáltatásként, de korábban komoly potenciált hordoztak magukban.

Az első említésre méltó eset az AXA [13] nevéhez köthető, amely egy Párizsi székhelyű multinacionális biztosítócég, mely 2021-ben nettó 7.29 milliárd euró bevételre tett szert. 2017 szeptemberében bemutattak egy új biztosítás konstrukciót. Ez volt a Fizzy [14], amely a repülőjáratok késése esetén automatikus kártérítést ígért a felhasználóknak, amennyiben a késés időtartama elérte a 2 órát. Az alkalmazás logikája egyszerű volt: amennyiben a felhasználó vásárolt egy repjegyet, a Fizzy alkalmazásában lehetősége volt erre késedelmi biztosítást kötni. A biztosítás ekkor feljegyzésre került egy Ethereum okosszerződésben. A platform folyamatos kapcsolatban volt a nemzetközi légi forgalmi adatbázissal és abban a pillanatban, amikor a késedelem elérte, vagy meghaladta a két órát a kártérítés kifizetése automatikusan megtörtént a felhasználó digitális tárcájának a címére. A biztosítás teljes egészében transzparens volt, a felhasználók előre tudták mennyi kártérítést fognak kapni késés esetében. Emellett biztonságos is, köszönhetően a blokklánc technológiának.

A Fizzyt kezdetben csak Párizs és az Amerikai Egyesült Államok között közlekedő tesztjáratokra lehetett használni, cél volt a járatok kiterjesztése. Ez azonban nem teljesen valósult meg, köszönhetően annak, hogy 2020 második felében a projektet eltörölték [15]. A biztosítócég elmondásai alapján a szolgáltatást érdeklődés hiányában szüntették meg.

A következő szolgáltatás, amit be kívánok mutatni, a *Blockchain Insurance Industry Initiative*, vagy röviden *B3i* nevéhez köthető. A 2016 végén alapított cég más megközelítésben használta a blokkláncot, mint az előbb említett AXA. Úgy gondolták, hogy a biztosítócégek közötti kommunikáció megfelelő standardizálása lehetőséget hordoz magában a hatékonyság és biztonság növelése szempontjából [16]. A bankrendszerben már létezik egy ehhez hasonló nemzetközi kommunikációs rendszer, a SWIFT, mely a mai napig meghatározó szerepet játszik a külföldi utalások esetében a bankok közötti kommunikáció lehetővé tételével.

A B3i tagjai úgy gondolták, hogy a kommunikációs rendszer alapja lehetne valamilyen blokklánc hálózat. Erre kitűnően alkalmasnak bizonyult egy privát, engedélyköteles blokklánc, így kezdetben a Hyperledger Fabric mellett döntöttek, 2018-ban azonban a Corda blokkláncára váltottak [17]. A rendszer célja a biztosítók és viszontbiztosítók hatékonyságának növelése volt, különböző adatok megosztásával. Természetesen az egyik legfőbb kritérium a személyes adatok védelme volt, így nem oszthattak meg egymás között a felhasználó beleegyezése nélkül szenzitív adatokat.

A blokklánc technológiának köszönhetően könnyen automatizálható, a hálózat résztvevői számára transzparens tranzakciókat lehetett végrehajtani. A biztosítást igénybe venni kívánt személyeknek is könnyebbséget jelenthetett a technológia, tekintettel arra, hogy nem kellett minden biztosításkötés alkalmával hosszú ideig elhúzódó, sok személyes találkozást igénylő folyamatot megvárni, mivel a biztosítócégek egymással is kommunikáltak. A rendszer mögé több híres cég is beállt. Az 5 alapítótag között helyet kaptak olyan cégek, mint az Allianz, Aegon, illetve a világ lehíresebb viszontbiztosítói közé tartozó Munich Re. és Swiss Re. Bár a szolgáltatás az indulás első éveiben több tíz millió dollárt hozott, 2022 júliusában sajnos bejelentették, hogy a részvényesekkel folytatott tárgyalást követően közösen arra az elhatározásra jutottak, hogy kellő anyagi támogatás hiányában meg kell szüntetniük a céget [18].

Az utolsó projekt, amelyet meg kívánok említeni, egy aktuális problémára nyújthat megoldást. Manapság sajnos egyre gyakrabban hallhatunk olyan esetekről, amelyek során az időjárás szélsőséges viselkedése miatt komoly károk keletkeznek a mezőgazdaságban. A biztosítócégek ritkán hoznak létre Afrika-szerte olyan konstrukciókat, amelyek a mezőgazdasági földeket biztosítják, tekintettel arra, hogy ez hatalmas pénzügyi kockázatot jelent számukra. A szegény országokban a farmerek kevesebb, mint 3%-a jut mezőgazdasági biztosításhoz egy 2021 novemberében készült felmérés alapján [19].

A *Lemonade Foundation* azonban létrehozta a *Lemonade Crypto Climate Coalition* névre hallgató koalíciót, mely célja egy olyan biztosítási szolgáltatás létrehozása, amely kezdetben csak Afrika-szerte vehető igénybe [20] körülbelül 2023 elejétől. A vállalat három fő problémát célzott meg: az időjárási kockázat számszerűsítése, a kártérítés automatizálása, valamint a kellő tőke és viszontbiztosítás garantálása. Így döntöttek végül a blokklánc technológia használata mellett, amely fő előnye az automatizálás.

A biztosítás az Avalanche nevű publikus blokkláncon fog futni és a felhasználók mobilalkalmazás segítségével tudnak biztosítást kötni [21]. A díjak fizetése, valamint a kártérítés jóváírása úgynevezett *stable coin* valutákban fog megtörténni [22]. Ezen valuták minden esetben rendelkeznek a megfelelő FIAT valuta (például amerikai dollár, euró, font stb.) alapú letéttel, így biztosítva a *token* értékének stabilitását. A kifizetések teljesen automatikusak lesznek, amennyiben az időjárás szélsőségessége elér egy bizonyos, előre definiált értéket, a kártérítés kifizetése azonnal megtörténik. A koalíció tagjai közt olyan cégek is megtalálhatók, mint a Chainlink, Etherisc és a Tomorrow.io.

## További technológiák bemutatása

Egy kész mobilalkalmazás elkészítése általában több technológia felhasználását igényli. Különösen igaz ez a blokklánc alapú alkalmazásokra, ahol valamelyest bonyolultabb a komponensek egymással történő kommunikációja, mint egy központi szerverrel rendelkező alkalmazás esetében. Tekintve, hogy számos technológiát felhasználtam a fejlesztés során, szeretném ezeket röviden bemutatni a dokumentum könnyebb megértése érdekében.

### Flutter

A Flutter a felhasználói felület programozására szolgáló, platformfüggetlen fejlesztői keretrendszer, ami azt jelenti, hogy azonos kódbázisból webre, Windows és Mac operációs rendszerre, valamint Android és IOS eszközökre is értelmezhető kódot generál a fordító. Ennek köszönhetően nem szükséges platformonként külön-külön fejlesztői eszközökkel megírni az alkalmazást.

A platformot a Google adta ki 2017 májusában és azóta évről évre növekvő népszerűségnek örvend [23]. Az ingyenesen használható, nyílt forráskódú keretrendszerben történő programozás Dart nyelven történik, mely fejlesztése szintén a Google nevéhez köthető. A frontend fejlesztést[[2]](#footnote-3) nagy mértékben leegyszerűsítik a Flutter által használt elemek, az úgynevezett *widget*ek. Ezek az elemek hierarchikusan egymásba ágyazhatók, így egy nagyszerűen testreszabható felületet készíthetünk, könnyen olvasható, kompakt kód segítségével. A technológiához készült könyvtárak közül válogathatunk a Dart nyelvhez készült, hivatalos tárházból [24], mely ugyancsak felgyorsítja a fejlesztés mentetét.

### Ethereum

Az alkalmazás mögött álló üzleti logikai réteg megvalósítása blokkláncon történt. Manapság egyre többféle blokklánc létezik. Egyik legnépszerűbb nyilvános blokklánc az Ethereum [25], mely megengedi az okosszerződések tárolását. Ezek különböző függvényeket tartalmaznak, melyek az üzleti logikát fogalmazzák meg, valamint eseményeket hozhatunk létre velük, melyek valamilyen külső alkalmazásból, bizonyos könyvtárak segítségével (lásd *2.4.6 bekezdés*) figyelhetők.

Az Ethereum blokkláncnak saját natív, helyettesíthető *token*je van. Ez az ether, melynek értéke a szakdolgozat írásának időpontjában 1257 amerikai dollárnak felel meg.

Az Ethereum a Bitcoin után a legnépszerűbb blokklánc. Az ide történő fejlesztés Solidity nyelven történik. Véleményem szerint a Solidity egy rendkívül érthető és logikus programozási nyelv, mely segítségével kulcs-érték párokat tárolhatunk, a szerződések között öröklődési hierarchiát alakíthatunk ki, definiálhatunk struktúrákat, listákat és még sok hasznos műveletet végezhetünk. Az okosszerződések megírása egy online kódszerkesztőben történt [26]. Az tesztelést megkönnyítik az oldal által biztosított virtuális gépek – melyek Londonban és Berlinben is találhatók – tekintve, hogy az okosszerződés ide történő telepítése gyorsan végbemegy, illetve a tranzakciók sem igényelnek privát kulccsal történő aláírást, ami szintén gyorsítja a fejlesztés folyamatát.

Mivel az okosszerződésekben található függvények meghívása a felhasználóknak pénzbe kerül – ez az úgy nevezett *gas price* – különösen fontos egy ilyen program megírása esetében nagy hangsúlyt fektetni a programkód optimalizálására. Vannak módszerek [27], melyek betartása hasznos egy optimális kód megírásakor, azonban az online kódszerkesztőben külön optimalizálhatjuk a szerződést annak blokkláncra történő telepítése előtt, így csökkentve önmagunk és felhasználóink a költségeit.

### Interplanetary File System

Az Interplanetary File System – vagy röviden IPFS – egy decentralizált, fájlok tárolására szolgáló, peer-to-peer hálózat, melynek lényege, hogy a csomópontok egymással kommunikálnak, központi szervergép nélkül. Amennyiben a felhasználó szeretne fájlokat megosztani a hálózaton, fel kell állítani a saját csomópontját, amely kommunikálni tud a hálózat többi pontjával. Erre több kliens is létezik, van amelyik rendelkezik felhasználói felülettel, van konzolos alkalmazás is. Több API is készült az IPFS-hez, melyek segítségével a kiválasztott állományokat megoszthatjuk, lekérhetjük. A hálózaton a fájlok tartalmának eltárolása történik, nem egy mutatót tárol a rendszer. Ez azért egy fontos pont, mivel így a metaadatok (fájl kiterjesztése, mérete, neve stb.) nem kerülnek eltárolásra.

A megosztást követően az állomány tartalmát egy úgynevezett *hash* algoritmus[[3]](#footnote-4) egy egyedi azonosítóra – szakirodalomban *content identifier* vagy *CID* – képezi le a rendszer. Fontos komponense az IPFS-nek a szemétgyűjtő, amely bizonyos idő eltelte után felszabadítja az olyan fájlokat, amelyeket nem jelöltek meg, így helyet biztosítva a következő adatoknak. Abban az esetben, ha megjelöljük a fájlunkat, a szemétgyűjtő azt nem fogja felszabadítani. Léteznek olyan szolgáltatók, amelyek IPFS alapú, távoli megjelölő szolgáltatást biztosítanak fájljaink számára. Az ide feltöltött fájlokat megjelölik, ezzel megvédve azokat a szemétgyűjtőtől. Ezen kívül több extra funkciót is nyújtanak, például az állományok redundáns tárolása, valamint a saját átjáró, amely segítségével nagyságrendekkel felgyorsítható az innen lekért adatok kiszolgálása. Természetesen igencsak korlátozottak a lehetőségeink, ha ezeket a szolgáltatásokat ingyenesen kívánjuk használni, azonban minimális havidíj ellenében már egy biztonságos, decentralizált fájltárolót kapunk.

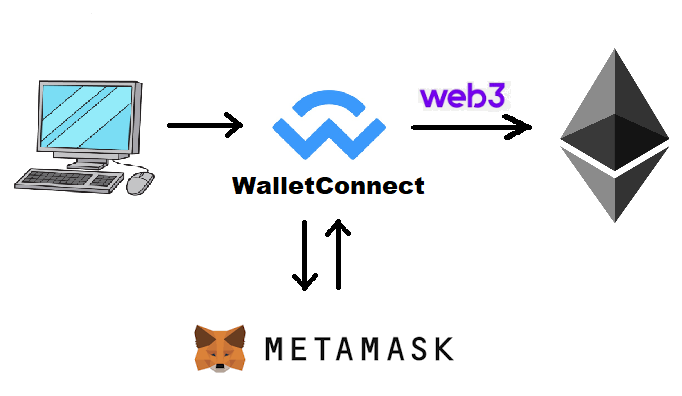
### Node.js

A Node.js egy szerveroldali JavaScript platform, amelyet JavaScript-ben írt alkalmazások webszerveren történő futtatására hoztak létre. Nagyon fontos eleme a Node.js projekteknek az npm [28], amely a JavaScript nyelv csomagkezelője. Segítségével a különböző dependenciák gyorsan integrálhatók alkalmazásunkba. Ezen kívül Node.js projekt inicializálására is használható, létrehozza a *package.json* fájlt, amelyben eltároljuk a projekt metaadatait, valamint a hivatkozásokat a külső függőségekre.

Ilyen külső függőség például az Express is, amely egy backend keretrendszer REST API létrehozásához. Az Express segítségével könnyedén definiálhatunk végpontokat, amelyekre http kéréseket küldhetünk. Az egyes kérésekre – amennyiben létezik ilyen végpont a projektben – válaszüzenettel reagálhatunk, mely tartalmazhat objektumokat is. Az Express ökoszisztémában nagyon nagy jelentőséggel bír az úgynevezett *middleware* fogalma. Az Express alkalmazások esetében a *middleware* egy olyan köztes függvény, amely három paramétert vár bemenetként, ezek a kérés, válasz és következő függvény (angolul *next*). Ezek a *middleware*-ek láncba vannak fűzve egymás után. Mindegyik függvénynek egy bizonyos feladata van, miután azt elvégezte, meghívja a következő feliratkoztatott metódust. A függvények között változókat is átadhatunk, így segítve az middleware-ek együttműködését. Amennyiben valamilyen hiba történt, a *next* függvény paraméteres meghívása után a hiba ágra futva hibaüzenetet küldhetünk a felhasználónak.

### WalletConnect

Úgy gondolom, röviden a WalletConnect technológiát is szükséges megemlítenem, hiszen a felhasználói felület és a digitális pénztárca kapcsolata elengedhetetlen az alkalmazás működéséhez. A WalletConnect segítségével Android vagy IOS alapú digitális pénztárcánkat köthetjük össze akár mobilalkalmazásokkal, akár valamilyen webes applikációval. A technológia támogatja a Metamask pénztárcáját is [29].



5. ábra: WalletConnect szerepe

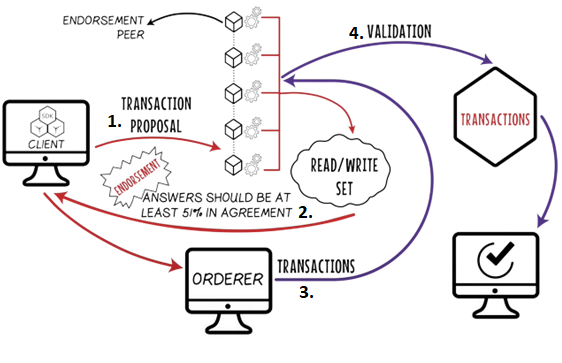
Ahogyan azt az 5. ábra is mutatja, a WalletConnect a felhasználói felületről indított, különböző blokklánc alapú kéréseket továbbítja a Metamask alkalmazásnak – vagy más, támogatott tárcának – amely a tárca privát kulcsa segítségévél aláírja a tranzakciót, így hitelesítve a felhasználót. Webes alkalmazás esetében WalletConnect használatával QR kódot jeleníthetünk meg, amely kódot beolvasva valamely támogatott tárca mobilalkalmazásából hitelesíthetjük a tranzakciót. Az okosszerződések függvényei ezt követően a WalletConnect és Web3 Dart könyvtárak együttműködésével hívhatók meg mobilalkalmazásból.

### Web3.js, Ether.js

A Web3 könyvtárcsoport különböző nyelveken implementált könyvtárak halmaza, amely segítségével az Ethereum blokklánccal kommunikálhatunk, adatokat kérdezhetünk le a blokklánc állapotával kapcsolatban. Legtöbben azonban egyértelműen az okosszerződésekkel történő kommunikációra használják ezeket a könyvtárakat. Ehhez mindösszesen annyi kell, hogy tudjuk, melyik hálózaton található a szerződés, ismerjük a címét, valamint az ABI-ját [30]. Függvényhívásokat kezdeményezhetünk, eseményeket hallgathatunk, csakúgy, mintha egy egyszerű központi backend szerverről[[4]](#footnote-5) beszélnénk. Az Ethereum blokklánc elterjedésével arányosan nő a Web3 könyvtárak által támogatott programnyelvek számossága. A legnagyobb támogatást azonban egyértelműen a JavaScript élvezi a Web3 világában. A Web3.js és az Ether.js a két legjobban dokumentált, karbantartott könyvtár, mely segítségével digitális pénztárcához csatlakozhatunk, könnyedén meghívhatjuk okosszerződésünk metódusait. A fejlesztés során mind a két könyvtárat használtam.

### Hyperledger Fabric

Korábban említettem már, hogy igen sokféle blokklánc kínálata közül választhatunk. Azonban míg az Ethereum és Bitcoin láncai az engedélymentes blokkláncok kategóriájába tartoznak, addig a Hyperledger Fabric lánca engedélyköteles[31]. Ez azt jelenti, hogy nem nyílt a hálózat mindenki számára, az ismert felek között történő tranzakciók végrehajtását támogatja. A fejlesztés a Linux Foundation nevéhez köthető, 2015-ben hozták létre a nyílt forráskódú blokkláncot. Legfontosabb felhasználása az úgynevezett *B2B,* vagyis *business to business* módszer, amely során két vagy több vállalkozás működik együtt egymással.



6. ábra: Konszenzus mechanizmus Hyperledger Fabricon

(forrás: ernesto.net)

A 6. ábra látható a Fabric konszenzus mechanizmusa. A beküldött tranzakció kérelmek kezdetben egy eredménnyel térnek vissza. Amennyiben legalább a csomópontok 51%-a egyetért, úgy a tranzakciók a validációt követően bekerülhetnek a főkönyvbe.

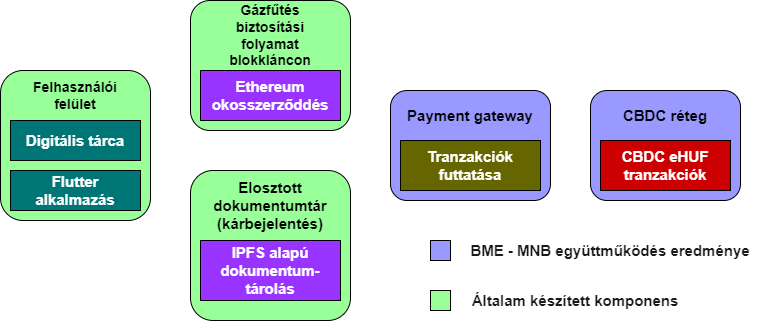
Fabricon is létrehozhatunk okosszerződéseket, amiket ebben az ökoszisztémában a szakirodalomban *chaincode*nak hívnak. Több nyelv közül választhatunk, amennyiben okosszerződést szeretnénk implementálni, azonban a legnagyobb támogatottságot a Go és JavaScript nyelvek élvezik. Mivel ez egy privát blokklánc, ezért natív tokenje nincs.

# Rendszertervezés

Minden komplexebb alkalmazás esetében elengedhetetlen a tervezésre szánt idő. Amennyiben ez a lépés elmarad, fejlesztés során olyan problémák merülhetnek fel, amelyek könnyedén elkerülhetőek egy alapos rendszertervvel. Az alábbi fejezetben a tervezés lépéseit, különböző fázisait mutatom be.

## Architektúra

Legelső lépésként a tervezés fázisában meg kellett határoznom, hogy milyen különböző komponenseket fog tartalmazni az alkalmazásom és ezek hogyan fognak egymással kommunikálni. Ennek eredményeképpen létrejött egy komponens diagram (lásd 7. ábra).



7. ábra: Komponens diagram

### Felhasználói felület

A komponensek bemutatását a felhasználói felülettel kezdem. Ez az a konkrét felület, amelyet a felhasználók látnak, illetve használnak. Jelen ez egy Flutter alkalmazás, különböző oldalakkal, melyek között a felhasználó navigálhat és különféle funkciók végrehajtását kezdeményezheti. Ebbe a komponensbe soroltam az applikáció használójának a digitális, kriptovaluta tárcáját is tekintve, hogy közvetlenül a felhasználói interfészhez csatlakoznak a tárcaként szolgáló böngésző bővítmények, vagy mobilalkalmazások. Amennyiben más komponensből szeretnénk meghívni az okosszerződés által implementált függvényeket, át kell adnunk a komponensek között az aláíráshoz szükséges adatokat, ami biztonsági kockázatot jelenthet.

### Biztosítási folyamat blokkláncon

A következő komponens a megvalósított biztosítási folyamat blokkláncon Ethereum okosszerződés formájában. Az üzleti logikai függvények implementálása itt történik, a felhasználói felület az okosszerződés belső állapotától függ. A felhasználói interfésszel ezért szoros kapcsolatban áll, a kommunikációt közöttük szükséges biztosítani. Ezt teszik lehetővé a Web3 Dart és WalletConnect könyvtárak.

### Elosztott dokumentumtár

Az applikáció egyik legfontosabb feladata, hogy amennyiben az ügyfelek gázfűtési berendezéseit kár éri, ezt be tudják jelenteni és a hivatalos kárbejelentés formájában kártérítést tudjanak igényelni. A kárbejelentési dokumentumok elosztott tárolását teszi lehetővé az elosztott dokumentumtár. Mivel a program az adatok és az üzleti logika elosztott tárolásán alapszik, ezért úgy döntöttem, hogy a kár valódiságát bizonyító dokumentumok tárolása is decentralizáltan történjen, amihez az IPFS technológiát használtam. Ennek a logikai részegységnek is szükséges kommunikálnia a felhasználói felülettel, ezért a BME Felhő által biztosított virtuális gépen található és http kérések formájában kommunikál az interfésszel.

### Eseménykezelő

Az eseménykezelő feladata az okosszerződéssel történő összekötés után az ott keletkező események fogadása és feldolgozása. A BME felhőben található komponens feladata a fizetéssel járó függvények hívása esetén kritikus. Az okosszerződésben létrehozott esemény hatására szükség esetén tranzakciót kezdeményez a CBDC számlák között, a fizetési átjáró segítségével.

### Payment gateway – fizetési átjáró

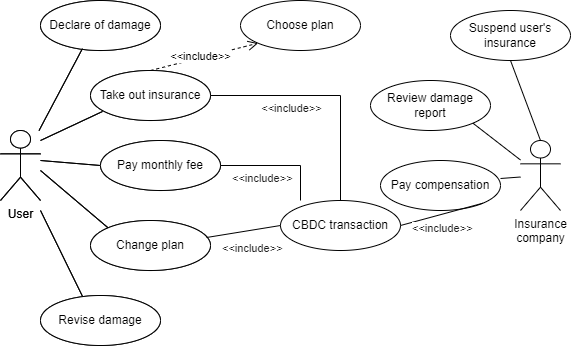
A havidíjak befizetéséhez a tanszék által készített eHuf digitális pénz használható. A Hyperledger Fabricon futó CBDC programmal valamilyen formában kommunikálni kell, ehhez elengedhetetlen egy átjáró, amely segítségével az okos szerződések függvényei meghívhatók. Ez egy Node.js REST API, mely szintén a BME Felhőben található.

### CBDC réteg

Ebben a rétegben történnek a CBDC alapú műveletek. A tanszéki digitális jegybankpénzt használva utalások hajthatók végre egy végfelhasználó és a biztosítási cég eHuf pénztárcái között, melyek egyenlegei lekérdezhetők.

## Használati esetek

Rendszertervezés során az egyik leghasznosabb lépés a különböző használati esetek definiálása. Ezeket az eseteket az alkalmazás résztvevőihez rendelve egy átfogó képet kapunk arról, hogy milyen célra használhatják a szereplők az applikációt (lásd 8. ábra).



8. ábra: Használati eseteket bemutató diagram

### Ügyfélhez tartozó használati esetek

#### Biztosítás kötése

Az alkalmazás talán legfontosabb használati esete, melyben az új felhasználó a digitális tárcájának csatlakoztatása után biztosítást köthet a biztosító céggel. Miután igazolta az ügyfél azt, hogy elmúlt 18 éves, valamint kitöltötte az adatokat, megköthető a biztosítás. Biztosítás megkötése után azonnal megtörténik az első havidíj befizetése, így ez egy CBDC tranzakciót futtató használati eset.

#### Biztosítási kategória kiválasztása

A biztosítás kötése magában foglalja a megfelelő kategória kiválasztását. A különféle kategóriák esetében más és más havidíjakkal kell kalkulálni, azonban minél több a havidíj, annál nagyobb összeget fizet a biztosítócég kár esetén.

#### Kárbejelentés

A kárbejelentés során a felhasználónak csatolnia kell egy hivatalos igazoló dokumentumot, valamint egy fényképet a kárról, majd meg kell adni a keletkezett kár mértékét eHuf pénznemben mérve. Csak akkor lesz sikeres a kárbejelentés, ha aktív a biztosítás státusza.

#### Biztosítási kategória változtatása

Az ügyfélnek lehetősége van a biztosítás megkötése után is megváltoztatni a biztosítási kategóriáját, mely a tranzakció lefutása után rögtön érvénybe lép. Magasabb kategóriába történő változtatáskor CBDC tranzakciót tartalmaz.

#### Kár felülvizsgálása

Az alkalmazás használói részt vesznek a kárbejelentések elbírálási folyamatában is azzal, hogy a kárbejelentésekben található, kárról készült képeket felülvizsgálhatják. Amennyiben két felhasználó valósnak ítél meg egy kárt, elkezdődhet a kifizetési folyamat.

#### Havidíj befizetése

Az alkalmazás során adott időintervallumokon belül szükséges befizetni a havidíjat, ellenkező esetben a biztosítócég kezdeményezheti az ügyfél felfüggesztését. Mivel digitális jegybankpénz utalása történik, ezért ez a használati eset szintén CBDC tranzakciót tartalmaz.

#### CBDC tranzakció

Ezen a használati eset során a fizetési átjáró és a CBDC okosszerződés segítségével eHuf digitális jegybankpénzek kerülnek mozgatása az ügyfelek és a biztosítócég között.

### Biztosító céghez tartozó használati esetek

#### Kárbejelentések felülvizsgálata

Ebben az esetben a biztosító cég megvizsgálja a hivatalos kárt igazoló dokumentumot és eldöntheti, hogy valósnak ítéli-e meg. Amennyiben igen, elkezdődhet a kifizetés.

#### Kártérítés kifizetése

Valósnak ítélt károk esetében a biztosító kifizeti a keletkezett kár bizonyos százalékát a károsult részére egy CBDC tranzakció során. Ez a százalék attól függ, hogy milyen biztosítási kategóriába tartozott a károsult.

#### Felhasználó biztosításának felfüggesztése

Abban az esetben, hogy ha a felhasználó nem teljesíti a biztosítási kategóriájához kapcsolódó havidíj kifizetését időben, a biztosítás felfüggeszthető.

## Követelmények

### Funkcionális követelmények

**Biztonság és adatvédelem**

**K.1** az üzleti logikát leíró függvények meghívásához érvényes szerződésre van szükség.

**K.2** a (rosszindulatú) felhasználók nem manipulálhatják a más publikus kulcshoz tartozó biztosítási szerződéseket.

**K.3** biztosítást igényelni csak azután lehet, miután a felhasználó bebizonyította, hogy nagykorú.

**K.4** a káresetek elbírálása felhasználóként anonim módon működik.

**K.5** a kártérítés összege a szerződés típusától függően kalkulálódik, az okosszerződés blokkláncra történő telepítése után nem manipulálhatók

**Megbízhatóság**

**K.6** ha a felhasználó nem megfelelő bemenetet biztosít, hibaüzenet után érvényes bemenetet kell megadnia

**K.7** a biztosítási vállalat csak a fizetési határidő utáni 10. naptól kezdődően tudja felfüggeszteni a biztosítási szerződést, addig az okosszerződés nem engedi

**K.8** felfüggesztett biztosítás esetén, a havidíj befizetése után a biztosítás automatikusan - a biztosító társaság beleegyezése nélkül – újra aktiválódik.

**Üzleti logika követelményei**

**K.9** egy publikus kulcshoz csak egy biztosítás tartozhat.

**K.10** időközben bármikor lehetséges a biztosítási kategória megváltoztatása.

**K.10.1** amennyiben a felhasználó a korábban befizetett időszak lejárta előtt vált magasabb havidíjú szerződésre, csak a korábbi havidíj és az új kategóriára vonatkozó havidíj különbségének kifizetése szükséges.

**K.10.2** amennyiben a felhasználó a korábban befizetett időszak lejárta előtt vált alacsonyabb havidíjú szerződésre, a fennmaradó összeg (korábbi havidíj és az új kategóriára vonatkozó havidíj különbsége) átváltásra kerül úgynevezett *utility token[[5]](#footnote-6)-*ekre.

**K.11** a következő havidíj legkorábban az előző havidíj befizetésének idejétől számított 30. napon fizethető be.

**K.12** a felhasználók által szerzett *utility token-*ek plusz kártérítési százalékokra válthatók be.

**K.13** minden érvényes kárelbíráló szavazat egy *utility token*-t ér

### Nemfunkcionális követelmények

**Hatékonyság**

**K.14** a szoftver internetkapcsolatot igényel.

**K.15** mivel blokkláncon tároljuk az alkalmazás állapotait, tárhelyigénye minimális.

**Hordozhatóság**

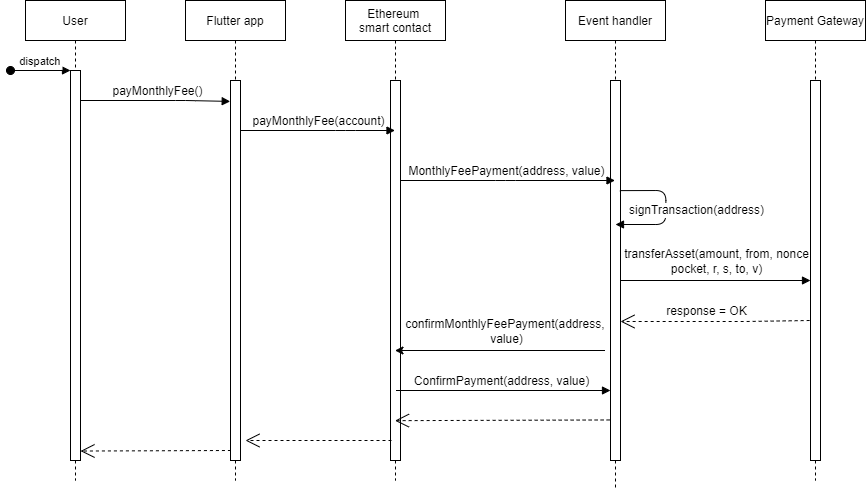
**K.16** az alkalmazás használatához minimum 13-as verziójú Android operációs rendszerre van szükség.

**K.17** a privát kulccsal történő aláíráshoz szükség van valamely, Ethereum hálózatot támogató digitális pénztárcára.

## Főbb funkciók bemutatása

Az alábbi fejezetben két fontos használati esetet mutatok be szekvencia diagramok segítségével. Ezen használati esetek a havidíj befizetése és a kárbejelentés.

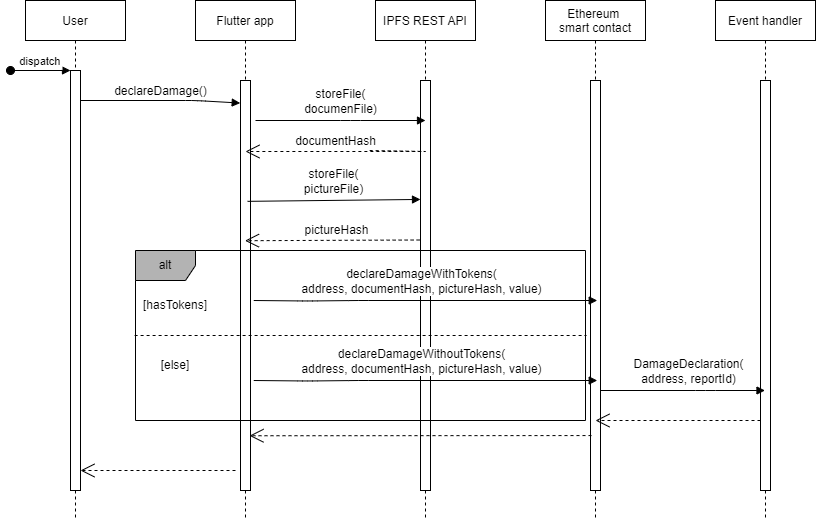
### Havidíj befizetés



9. ábra: Havidíj befizetés szekvencia diagramja

Ahogyan azt a *9. ábra* is mutatja, a havidíj befizetését a felhasználó kezdeményezi az alkalmazás felületén egy gombra kattintva. Ehhez a gombhoz van hozzárendelve a *payMonthlyFee* függvény, mely ezek után a Web3 és a WalletConnect könyvtárak segítségével meghívja az okosszerződés azonos nevű függvényét. Itt az üzleti logikai folyamat lefutása után keletkezik egy esemény két paraméterrel, a befizető felhasználó tárcájának címével, valamint a befizetett összeggel. Ezt a Node.js-ben implementált eseménykezelő figyeli, majd az esemény keletkezése után megtörténik az eseménykezelőben a tranzakció aláírása, majd az aláírás után kapott értékekkel a digitális jegybankpénz átutalása a forráscímről a célcímre, amely célcím a biztosítócég digitális pénzeket tároló tárcája. Amennyiben az tranzakció sikeresen zajlott, megtörténik a fizetés megerősítése, a megfelelő okosszerződés függvényt meghívva, ahol a szerződés státusza aktívra változik, amennyiben az inaktív lett volna korábban. Itt is keletkezik ekkor egy esemény, hogy megbizonyosodjunk a metódushívás sikerességében, majd visszatérünk a felhasználói felülethez.

### Kárbejelentés



10. ábra: Kárbejelentés szekvencia diagramja

A következő használati eset, amit bemutatok, a kárbejelentés esete. Természetesen a kiindulási pont itt is a mobilalkalmazás felhasználói felülete. Ezen a felületen 2 fájlt szükséges a felhasználónak feltölteni, melyek a kárbejelentés hivatalos dokumentuma, valamint a kárról készült kép. Miután feltöltötte a kellő állományokat a felhasználó, szükséges megadnia a keletkezett kárt eHuf – amely a tanszék által készített digitális jegybankpénz – pénznemben. Miután a kért adatok kitöltésre kerültek, a felhasználó beküldheti a kárbejelentési kérelmét. Amennyiben a kliens rendelkezik az alkalmazás által kezelt GIT tokennel, felajánlja neki az applikáció, hogy plusz kártérítési százalékokért cserébe beválthatja azokat. Ahogy a 10. ábra mutatja, attól függően, hogy rendelkezik-e tokennel a felhasználó, valamint, hogy szeretné-e azokat beváltani, két külön függvény kerül meghívásra. Itt megtörténik a kárbejelentés eltárolása a blokkláncon, valamint egyéb logikai műveletek, majd keletkezik egy *DamageDeclaration* nevű esemény két paraméterrel, melyek a bejelentő címe és a kárbejelentés azonosítója. Ezt követően a felhasználó visszatér a felhasználói felülethez és folytathatja tevékenységét.

# Önálló munka bemutatása

A 3.1. bekezdésben bemutattam az alkalmazásom komponenseit, valamint, hogy melyik részegység milyen feladat ellátásáért felelős. Az alábbi fejezetben a feladatom lényegi részét, a fejlesztés során felmerülő tervezői döntéseket és az elkészült végtermék mögötti üzleti logikát szeretném bemutatni komponensenként. Az általam választott bemutatási sorrend a felhasználói felület felől halad az üzleti logikát tartalmazó blokklánc és dokumentumtár rétegén keresztül a fizetést megvalósító átjáróig, hiszen az alkalmazás megvalósítása is ilyen irányba történt és úgy gondolom, hogy ez a leglogikusabb megközelítés.

## Felhasználói felület

Az alkalmazás fejlesztését a felhasználói felülettel kezdtem. Kezdetben ki kellett találnom, hogy milyen platformot használjak a nézetek létrehozására. Tekintve, hogy a cél egy mobilalkalmazás megvalósítása volt, lehetőségem volt natív Android, natív iOS, vagy platformfüggetlen alkalmazást létrehozni. Rövid gondolkodás után platformfüggetlen megoldás mellett döntöttem. Tekintve, hogy a két leghíresebb platformfüggetlen keretrendszer a Flutter, illetve a React Native [23] a lehetőségeket erre a két keretrendszerre szűkítettem le.

A Flutter ellen szól, hogy a felület elemeit egy vászonra rajzolva jeleníti meg, így a keresésoptimalizálás kevésbé fejlett, mint a React Native esetében, ahol az adott platform natív elemeit használja fel a felület elkészítéséhez. Ezzel szemben a programozás Flutterben rendkívül gyors, köszönhetően a hierarchikus, egymásba ágyazott komponenseknek. Mivel ez egy gyorsan fejlődő technológia, a hatalmas közösségi támogatás is mellette szól. Emellett saját megjelenítő motorja végett nagyon gyors megjelenítést is tesz lehetővé. Ezek alapján úgy döntöttem, hogy az felhasználói interfészt Flutter technológia segítségével implementálom. A Flutter projekt konfigurálását követően a következő lépés a nézetek létrehozása volt.

### Tárca csatlakoztatása

Az alkalmazás megnyitásakor a felhasználót a tárca csatlakoztatása nézet fogadja (*lásd 12. ábra*). Tekintve, hogy a logika blokkláncon készült, kötelező először a digitális tárcát csatlakoztatni, hiszen a függvényhívások privát kulccsal történő aláírást igényelnek, mely a csatlakoztatás után lehetségessé válik a WalletConnect és Web3 technológiák segítségével. A csúszka elhúzásával egy WalletConnect hívás történik, mely átirányítja a felhasználót a külső Metamask alkalmazásba. Amennyiben a csatlakozás sikeresen megtörténik a Metamask-ban, a függvényhívás visszatér a munkamenet adataival, köztük a felhasználó publikus kulcsával.

Ezt követően elmentésre kerülnek a fontosabb adatok egy központi, singleton*[[6]](#footnote-7)* osztályba. Ezek az adatok a digitális tárcához kapcsolódó publikus kulcs, valamint a felhasználó hitelesítő adatai, mely segítségével az aláírás megtörténhet. Ezen a ponton történik meg az okosszerződést reprezentáló osztály, a *GasInsurance* példányosítása is. Ez egy generált osztály, melyet egy Flutter parancs kiadása után hoz létre a fordító – az okosszerződést leíró ABI fájlból – és ezen keresztül történhetnek meg később a függvényhívások. Az adatok inicializálásához egy Github kérdésre adott válaszból merítettem ihletet [32]. A tárca sikeres csatlakoztatása után a megfelelő okosszerződés függvényt meghívva ellenőrizzük, hogy a felhasználó rendelkezik-e biztosítással. Amennyiben nem, a biztosítás megkötése oldalra navigáljuk, ellenkező esetben pedig a főképernyő tárul elé. A képernyők közötti navigálás során adatokat kérünk le a blokkláncról, így a várakozás ideje alatt egy töltőképernyő látható (lásd *13. ábra*).

### Biztosítás kötése

Abban az esetben, ha a tárcáját csatlakoztató felhasználó nem rendelkezik még biztosítással, a biztosítás kötése nézet fogadja őt. A képernyő tetején ekkor egy figyelemfelhívó üzenet jelenik meg, mely a teljesség igénye nélkül tájékoztatja az ügyfelet a biztosításkötés feltételeiről. Ezt követően szükséges azonosítania magát a felhasználónak. Ezt a személyi igazolvány két oldalának fényképével teheti meg. Ezek a fájlok csak meghatározott formátumúak lehetnek, a fájl kiválasztásakor csak az elfogadott formátumú képek jelennek meg. Amennyiben téves fájlt került kiválasztásra, törlés után újat választhat az ügyfél. Ezután a három biztosítási kategória egyikét szükséges kijelölni. A különböző kategóriák kártyákba vannak rendezve, melyen megjelennek a legfontosabb adatok, mint a havidíj és a káreset bejelentése esetén kapott kártérítés, a káreset mértékének százalékában kifejezve.

Feltöltés után először az igazolvány két oldalát ábrázoló képek kerülnek megosztásra az IPFS platformon. Ez egy általam készített REST API (*4.3.* *bekezdés*) hívás segítségével történik. Mivel az API elvár egy egyedi azonosítót, amely a képeket tartalmazó mappa neve lesz, a képek megosztása előtt generálok egy egyediséget biztosító karakterláncot. Az 20 karakter hosszú azonosító tartalmazhat kis- és nagybetűket, illetve számokat véletlenszerűen. Így elhanyagolhatóan kicsi az esély arra, hogy két ugyanolyan azonosítót hozzunk létre (ez a szám egészen pontosan 1,4 \* 10-34 százalék).

A fájlok IPFS-en történő megosztása után a biztosítás megkötéséért felelős okosszerződés függvény meghívása következik. A tranzakció elindítása előtt a felhasználó átirányításra kerül a Metamask alkalmazásba, ahol megjelenik egy felugró ablak, mely megkérdezi a tárca tulajdonosát, hogy alá szeretné-e írni a tranzakciót a privát kulcsával, vagy sem. A döntését befolyásolhatja a többletköltség mértéke is, mely a blokklánc használatáért fizetendő tranzakciók után. Ez az érték is látható lesz a felhasználó számára, így amennyiben soknak találja a költséget, visszautasíthatja a tranzakciót. Az opció kiválasztása után visszajut az ügyfél az alkalmazás kezelőfelületére, ahol pár másodperc várakozás után a főképernyő várja őt a már meglévő szerződése adataival (amennyiben elfogadta a tranzakciót).

Fontosnak érzem kiemelni a felhasználó azonosításának folyamatát, ugyanis a kezdeti ötletben nem ez a megoldás szerepelt. Felmerült az az opció, hogy mi lenne, ha a felhasználó identifikálása során az úgynevezett Self-soverign Identity megközelítésre támaszkodnék [34]. Ez egy olyan, elosztott rendszerek esetében egyre jelentősebb szerepet kapó digitális mozgalom, mely a személyazonossággal való visszaélések növekedésének hatására erősödött meg az elmúlt évek folyamán. Csak az Egyesült Államokban 2020-ban a személyazonossággal való visszaélések körülbelül 712 milliárd dollár értékű kárt okoztak [35].

A megközelítés lényege, hogy a magánemberek rendelkeznek az identitásukat igazoló, képességeiket elismerő digitális okmányok felett, mindenféle felsőbb hatóság közbejárása nélkül. Az SSI része az úgynevezett *Verifiable Credentials* [36]technológia, mely során egy engedéllyel rendelkező hatóság kiállít egy bizonyítvány kártyát, mely igazolja az ügyfél identitását, vagy valamilyen képességét. A technológia ezen résztvevője a kibocsátó*.* Ezt követően az ügyfél bemutathatja azt egy olyan hatóságnak, melynek jogosultsága van a kártya hitelesítésére. Ez a résztvevő a hitelesítő.

A személyazonosító igazolványok lefotózása helyett ezen technológia segítségével szerettem volna a felhasználókat ellenőrizni. A kártyák kiállítása sikeresen meg is történt a Trinsic és a Zapier platformok segítségével (*lásd* *14. ábra*). A kártyán mindösszesen egy név, valamint egy életkor szerepelt. Az életkor a verifikáció során nem került volna bemutatásra a hitelesítők számára, csupán egy igaz/hamis érték, hogy a felhasználó betöltötte-e a 18. életévét, vagy sem.

A probléma a kártya hitelesítése során akadt. A keretrendszer leginkább JavaScript nyelven terjedt el és még nem teljes a Dart nyelv támogatottsága. Kezdetben egy *okapi* nevű könyvtár hiánya miatt nem futott le a program, azonban köszönhetően a Trinsic közösségnek és az ott dolgozó mérnököknek, erre a problémára találtunk megoldást. Ezt követően vettem csak észre, hogy nem rendelkezik a Dart nyelvre készült API olyan metódussal, amely során egy már kibocsátott kártyát lehetne azonosítani digitális tárca segítségével. Tekintve, hogy ezzel a megoldással itt elakadtam, létrejött az ötlet, amely segítségével jelen pillanatban is ellenőrizhető a felhasználók életkora az applikációban.

### Főoldal

Amennyiben rendelkezik biztosítással a felhasználó a tárca csatlakoztatása (vagy új biztosítás megkötése) képernyő után a főoldal fogadja őt (*lásd 17. ábra*: Főoldal). Ez az oldal egy átfogó képet ad a biztosítás legfőbb adatairól. Az oldal megjelenítése előtt lekérdezésre kerül a szerződés státusza, szerzett *utility token*ek száma, fizetési határidőig hátralévő idő és jelenlegi biztosítási kategória is.

Miután megtörténik ezek változókba mentése, visszatér az alkalmazás az aszinkron inicializáló metódusból, majd megjeleníti az oldalt az elmentett adatokkal. Amennyiben érvényes a szerződés, a *valid* felirat jelenik meg címben, ellenkező esetben egy piros *invalid* feliratot láthatunk. Ezalatt helyezkedik el a szerzett *utility token*ek száma, valamint a fizetésig hátralévő idő. Ez az érték napokban történik kiírásra, amennyiben több, mint 1 nap a hátralévő idő. Ha kevesebb, akkor órákban és ha már nincs egy óra a befizetésig, akkor percekben.

A követkető sorban láthatunk két fontos gombot, melyek a szerződés lemondására és a havidíj befizetésére szolgálnak. Ezen tranzakciók mind két esetben privát kulccsal történő aláírást igényelnek, ezért Metamask alkalmazásba történő átirányítás után megtörténhet a tranzakció elfogadása. Amennyiben a havidíj befizetése sikeresen megtörtént, a fizetésig hátralévő idő megnövekszik az okosszerződésben deklarált fizetési intervallum idejével (ez a telepített szerződésben 3 perc, a tesztelhetőség érdekében). Feltéve, hogy a felhasználó felmondta a biztosítását, elnavigáljuk őt a biztosításkötés oldalára. A biztosítás ekkor törlésre kerül, azonban új biztosítás kötése esetén az *utility token*ek száma megmarad.

A képernyő második felében megtekinthetjük a jelenlegi biztosítási kategóriát. Ha ezen változtatni szeretnénk, az alatta elhelyezkedő gomb segítségével ezt megtehetjük. Ekkor a felhasználó a biztosítási kategória megváltoztatása (*4.1.4. fejezet*) oldalra lép.

### Biztosítási kategória megváltoztatása

Amennyiben a felhasználó változtatni szeretne biztosítási kategóriáján, az alábbi képernyő fogadja őt (*lásd 18. ábra*). A cím alatt egy rövid figyelmeztető üzenet látható, mely arról tájékoztatja az alkalmazás használóját, hogy a kategória megváltoztatása után automatikusan megtörténik a havidíj befizetése is.

Kétféle forgatókönyv lehetséges: a felhasználó alacsonyabb, vagy magasabb havidíjú kategóriára vált. Amennyiben alacsonyabbra, úgy nem történik változás az egyenlegén és *utility token*eket szerez, mivel már korábban kifizetett előre egy magasabb havidíjat, amit nem használhatott ki a ciklus végéig. Magasabb havidíjra váltás esetében a különbözet kifizetése szükséges. A képernyő alján található gomb megnyomása után az logikát implementáló függvények lefutnak a telepített okosszerződésben, majd az ügyfél visszairányításra kerül a főoldalra.

### Kárbejelentés

Az alul elhelyezkedő sáv lehetővé teszi az oldalak közötti navigálást. Feltéve, hogy a felhasználó a második, *Declare of damage* nézetre vált, a kárbejelentő oldal fogadja (*lásd 19. -* 20. ábra). Az oldal inicializálása után a képernyő felső részében két gombot láthatunk, melyek segítségével a fájlkezelőből kiválaszthatjuk a feltölteni kívánt állományokat. Az első esetben a várt fájl egy hivatalos dokumentum, melyet egy gázfűtési eszközök javítására specializálódott cég ad ki, a második állomány pedig egy kép kell legyen, amely jó minőségben ábrázolja a keletkezett kárt. A következő sorban található beviteli mezőben meg kell adni a keletkezett kárt eHuf pénznemben.

Ezt követően, a rövid informáló szöveg alatt látható gombra kattintva bejelenthető a keletkezett kár. A bejelentés megkezdése előtt a fájlok megosztásra kerülnek a lokális IPFS csomóponton. Az útvonalak, mely segítségével lekérdezhetők az állományok, a válaszüzenetben megtalálhatók és megosztás után elmentésre kerülnek lokális változókba. Amennyiben nem rendelkezik az ügyfél *utility token*ekkel, a tranzakció Metamaskban történő aláírása után végbemegy a kár bejelentése. Ellenkező esetben egy felugró ablak tárul az applikáció használója elé, melyben eldöntheti, hogy szeretne-e plusz kártérítési százalékokért cserébe tokent felhasználni, vagy sem (*lásd 19. -* 20. ábra). Fontos, hogy a kártérítés összege nem lehet több, mint a keletkezett kár 100%-a, illetve minden felhasznált token plusz egy százalékot ér. Attól függően mit választ az ügyfél, meghívásra kerül a megfelelő okosszerződés függvény.

### Károk elbírálása

A károk elbírálása nézet fontos eleme az alkalmazásnak, hiszen szavazásért cserébe utility tokent kaphatnak a felhasználók. Amikor a navigációs sor segítségével átlép az ügyfél erre a nézetre, töltés közben az okosszerződéshez fordul az applikáció annak érdekében, hogy lekérdezze egy, még aktív kárbejelentéshez tartozó fotó IPFS csomóponton értelmezett elérési útvonalát. Előfordulhat, hogy nincs aktív kárbejelentés, ekkor egy hibaképernyő kerül megjelenítésre, amely informálja a felhasználót erről. Ezt követően megtörténik az IPFS hálózatáról a fotó lekérdezése az előbb említett útvonal alapján. A válaszüzenet fejlécében megtalálható a fájl neve, illetve kiterjesztése is. Ez alapján létrehozunk egy fájl változót, melynek neve, illetve kiterjesztése megegyezik az IPFS hálózaton megosztott fájl metaadataival. A válaszüzenet törzsében lévő bájttömböt, mely a kép tartalmát reprezentálja átmásoljuk a változóba, majd létrehozásra kerül egy fájl típusú kép elem, amelyet megjelenít az applikáció a képernyő közepén (lásd *21. ábra*: Kár elbírálása nézet. A nézet alsó részében megtalálhatók a szavazásra szolgáló gombok, melyek megnyomása után meghívásra kerülnek az őket leíró okosszerződés metódusok.

## Ethereum okosszerződések

Az alkalmazás üzleti logikai része okosszerződések formájában került megvalósításra. Ehhez az Ethereum blokkláncát (*2.4.2. fejezet*) használtam. A karbantarthatóság és átláthatóság érdekében három különálló okosszerződést hoztam létre, melyek az öröklődési hierarchiának köszönhetően ismerik az ősszerződés(ek) nem privát láthatóságú adattagjait és metódusait. Mivel korábban foglalkoztam már blokklánc technológiával, de okosszerződések implementálására nem került sor részemről, ezért a feladat elkezdése előtt a CryptoZombies [37] ingyenes, interaktív kurzusa segítségével sajátítottam el a Solidity nyelv alapjait. Ezt követően elkezdtem az első, a biztosítást kezelő okosszerződés implementálását.

### Biztosítást kezelő okosszerződés

A biztosítás kezeléséért felelős okosszerződés egy absztrakt szerződés [38], mely ősei az OpenZeppelin standard által implementált Ownable [39] és ERC20 [40] okosszerződések, amelyek a tokenkezelés során jelentős szerep kapnak. A program fő struktúrája a biztosítást reprezentáló *Insurance,* ami automatikusan növekvő azonosítóval rendelkezik. Ezen kívül olyan adattagjai vannak, mint a biztosítás státusza, kategóriája, a következő fizetés időpontja, valamint a biztosításhoz tartozó fiók által leadott kárelbíráló szavazatok száma. Ezutóbbi azért fontos, hogy kordában tartsuk a felhasználók által birtokolt utility tokenek számát azzal, hogy naponta maximum 5 érvényes szavazatot adhatnak le. A biztosítási kategória szintén egy általam létrehozott struktúra, mely három tagváltozója a kategória száma (melynek felvehető értékei 1, 2, vagy 3), havidíja és a kár esetén a biztosítócég által fizetett százalék. A biztosítások egy szótár adatszerkezetben – melynek Solidity nyelvbeli megfelelője a *mapping* adatstruktúra – kerülnek eltárolásra. Ezek kulcsa egy digitális tárca publikus kulcsa, értéke pedig egy *Insurance* struktúra. Ezen kívül két publikus tagváltozót definiáltam, melyek a fizetés gyakoriságát, valamint a fizetési határidő lejárta utáni türelmi időszakot[[7]](#footnote-8) jelképezik.

A programkód első harmadában különböző ellenőrző függvények találhatók. Ezek hozzárendelhetők különféle metódusokhoz és amennyiben az általuk definiált feltétel nem teljesül, úgy egy – a fejlesztő által megfogalmazott – hibaüzenettel tér vissza a függvény. Az első üzleti logikához kapcsolódó metódus egy címet vár paraméterként és amennyiben rendelkezik a címhez kapcsolódó felhasználó biztosítással igaz a visszatérési érték, ellenkező esetben hamis. Ezt oly módon ellenőrzöm, hogy a címhez tartozó biztosítás azonosítója nagyobb-e, mint nulla, ugyanis a Solidity nem tárol *null* értékeket, csupán kinullázza (0x00 hexadecimális szám értékével egyenlővé teszi) a nem létező adatokat. A programban található egy kategória lekérdezésére szolgáló függvény, mely visszaadja a címhez kapcsolódó biztosítási kategória számát, valamint egy olyanmetódus is, mely igazzal tér vissza, ha a címhez köthető biztosítás státusza aktív és hamissal, amennyiben ez a státusz inaktív.

Az első olyan függvény, mely a blokklánc állapotát megváltoztatja, a biztosítás megkötésére szolgál. Két bemenő paramétert szükséges megadni, melyek a digitális tárca publikus kulcsa, valamint a megkötni kívánt biztosítás kategóriájának azonosítószáma. Ezt követően a kategória számából létrehozunk egy biztosítási kategória struktúrájú objektumot, valamint változóba mentjük a fizetési határidőt is, mely egyenlő lesz a jelenlegi időbélyeg és a publikus változó formájában elérhető, fizetés gyakoriságát definiáló változó összegével. Ezután először létrehozunk egy eseményt, mely egy új ügyfél regisztrációjára hívja fel a figyelmet, majd a biztosítások tárolására kialakított szótárban elmentésre kerül a biztosítás. A függvény a havidíj befizetésére szolgáló metódus meghívásával ér véget, hiszen új szerződés megkötésekor automatikusan befizetésre kerül az első havi díj is.

A havidíj befizetésére szolgáló függvényben csupán egy esemény keletkezik két szükséges paraméterrel, melyek a publikus kulcs és a befizetendő összeg. Ez az esemény az eseménykezelőben feldolgozásra kerül és abban az esetben, ha sikeresen lezajlott a díj befizetése, a fizetés megerősítése céljából kreált metódust hívja meg az említett komponens. Ez a függvény azt az előfeltételt ellenőrzi a lefutása előtt, hogy a tárca publikus kulcsa, ahonnan a függvényhívás megtörtént, megegyezik-e a tagváltozóként deklarált eseménykezelő publikus kulcsával. Kezdetben ezen változó értéke nulla, azonban a biztosítócég a szerződés blokkláncra történő telepítése után ezt az értéket be tudja állítani a kívánt címre. Fontos, hogy az átjáró címének beállítása csak a biztosítócég által történhet meg.

A korábban említett Ownable szerződésből való öröklődés ezen a ponton kap értelmet. A fizetés megerősítése függvény törzsében beállításra kerülnek a létrehozott biztosítás státusza, következő fizetés ideje és a türelmi időszak vége változók, majd egy újabb esemény keletkezésével megerősíthetjük a felhasználót a fizetés sikerességéről. Ezek mellett található az okosszerződésben még két fontos függvény, melyek az ügyfél felfüggesztése, illetve a biztosítás felmondása. Korábbi csak a biztosító privát kulcsával történő aláírással fogadható el és a metódus törzsében a státusz inaktívvá változtatását követően egy eseményt hoz létre. Utóbbi esetében a biztosításokat tartalmazó szótárból kitörlésre kerül az tranzakció küldője, melyet szintén egy esemény létrehozása követ.

### Kárbejelentést kezelő okosszerződés

Az absztrakt, kárbejelentésekért felelős okosszerződés a biztosítást kezelő okosszerződésből öröklődik. Programkódjában csak olyan függvényekkel találkozhatunk, melyek a kártérítéshez kapcsolódnak szorosan. Hasonlóan az őséhez, itt is van egy fő struktúra, mely a *Report* nevet kapta. A típus változói többek között az automatikusan inkrementálódó azonosító, a hivatalos kárbejelentő dokumentum, a kárt bizonyító fotó IPFS-en értelmezett elérési útja, a kár összege, melyből kalkulálható a kártérítés összege, illetve egy igaz/hamis értékkel rendelkező státusz, mely megmutatja, hogy a bejelentés elfogadott állapotban van-e, vagy sem. Ezeket a bejelentéseket – hasonlóan a 4.2.1. fejezetben leírt okosszerződéshez – egy szótárban tároljuk. A későbbi továbbfejleszthetőség érdekében a már elfogadott kárbejelentések egy külön szótárban is tárolásra kerülnek, hogy a keresés gyorsabb és olcsóbb legyen. Publikus változó formájában megjelenik a jelenleg el nem fogadott, elbírálás alatt álló kárbejelentések száma is.

A kód első részében itt is előfeltételek ellenőrzésére szolgáló függvények találhatók, majd látható egy olyan metódus, mely attól függően, hogy a felhasználó rendelkezik-e kárbejelentéssel egy igaz/hamis értékkel tér vissza.

Ezt követően elérkezünk az okosszerződés legfontosabb metódusához, mely a kár bejelentésére szolgál. Paraméterként egy címet, két IPFS elérési útvonalat, valamint egy kárösszeget vár. A biztosító által térítendő összeg kiszámolását követően megtörténik a bejelentés blokkláncon történő eltárolása, az aktív kárbejelentések számának növelése, valamint keletkezik egy esemény a digitális tárca publikus kulcsával és a kárbejelentés azonosítójával. Azért csak az azonosítót tartalmazza az esemény, mert ezek igen költséges műveletek tudnak lenni, főleg, ha egy általunk kreált struktúrát próbálunk benne visszaadni, hiába az indexelt paraméterátadás.

A kárbejelentések elbírálása a négy szem elv alapján működik [41]. Ezáltal lehetősége van a felhasználóknak felülvizsgálni más ügyfelek által bejelentett káresetek érvényességét oly módon, hogy egy érvényben lévő kártérítési kérelemhez tartozó, a káresetről készült képet látván eldönthetik, hogy szerintük ez egy valós kérelem, vagy sem. Amennyiben két különálló személy is úgy gondolja, hogy a kérelem helytálló, úgy a kárbejelentés elfogadott státuszba kerül és megtörténhet a kompenzáció kifizetése a biztosítócég részéről.

A gyakorlatban a megfelelő képre navigálva (*4.1.6 bekezdés*) meghívásra kerül egy olyan függvény, mely feladata egy aktív kárbejelentés véletlenszerű kiválasztása után visszaadni a káresetről készült fotó IPFS elérési útvonalát. Fontosnak érzem kiemelni, hogy a prototípus elkészítése során nem biztonságos random szám generátort használtam. Az én megoldásomban a jelenlegi blokk időbélyegéből, az üzenet küldőjének címéből és egy véletlenszerű számból SHA256 algoritmus [42] segítségével hash-t generáltam, amely maradékos osztásából nyertem ki a szükséges véletlenszerű indexet. Ez valójában sérülékenységhez vezethet, külső orákulumok használata ajánlott [43].

Amennyiben a káreset elbírálója jogosnak véli a követelést, egy olyan metódus fut le, amely a megfelelő esemény létrehozása után kibocsát egy *utility token*t a bíráló félnek, majd ellenőrzi, hogy két fél által is elfogadásra került-e a kártérítési kérelem. Ha igen, úgy a státusz megváltoztatásra kerül, valamint törlődik a szótárból a kárbejelentés. A függvénynek a feladata a bíráló fél által leadott szavazatok számának növelése is, ami abban az esetben, ha ez a szám ötre növekedett, beállít egy felfüggesztési időszakot, ami alatt nem engedélyezett a szavazás. Ez a telepített szerződés esetén egy nap. Ha az ügyfél szerint jogtalan a követelés, egy esemény keletkezése után szintén megkapja a jutalomként járó *utility token*t, majd ebben az esetben is ellenőrzésre kerül a leadott szavazatok száma.

Lehetőség van a felhasználókat megkerülve, a kárbejelentés közvetlenül a biztosító által történő elfogadásra is. Ekkor egy, csak a biztosítócég által hívható függvény fut le és végzi el az elfogadáshoz szükséges műveleteket. Ha egy kárbejelentés elfogadott státuszban van, a biztosítócég kezdeményezheti a kártérítés kifizetését, mely a célcím megadását követően egy eseményt generál. Ekkor az eseménykezelő a fizetési átjárón keresztül végrehajtja a kártérítés kifizetését.

### Biztosítási tokent kezelő okosszerződés

Az utolsó, a blokkláncra feltelepített okosszerződés őse a kárbejelentésekért felelős okosszerződés. Tekintve, hogy ez az egyetlen nem absztrakt szerződés, inicializálása során fontos volt az ERC20 program konstruktorának meghívása. Így jön létre az úgynevezett *Gas Insurance Token*, vagy röviden *GIT* token, mely azon *utility token*eket reprezentálja, melyek az alkalmazás használatáért kaphatók. Minden olyan függvény, mely kapcsolatban van az ezekkel a tokenekkel, itt található.

Mint korábban említettem (*4.2.2 bekezdés*) a kárbejelentések érvényességét felülvizsgáló szavazók GIT tokenben részesülnek. Azonban nem ez az egyetlen mód azok gyűjtésére. Szerettem volna, ha flexibilisebb biztosítást tudna nyújtani az applikáció, mint a legtöbb biztosítócég konstrukciói. Ezért úgy valósítottam meg az üzleti logikát, hogy a fokozatok közötti váltás esetében ne kelljen megvárni a befizetett hónap lejártát, hanem abban a pillanatban lehetőség legyen egy másik kategóriába történő átsorolásra, figyelembe véve azt, hogy az ügyfélnek még érvényes biztosítása van. Ez magasabb biztosítási csoportra történő váltás esetén a különbözet kifizetését vonja maga után. Alacsonyabb kategória választásakor pedig a konstrukció kompenzálja a felhasználót GIT tokenekkel. Ekkor figyelembe vesszük, hogy egy, vagy két kategóriával alacsonyabb az újonnan választott biztosítás, valamint azt is, hogy hány nap van még a befizetett intervallumból. Az így kalkulált szabadon felhasználható tokenek jóváírásra kerülnek az ügyfél számláján.

Ha magasabb kategóriára történő váltás esetén az ügyfélnek befizetési kötelezettsége van,

egy eseménnyel jelezzük ezt az eseménykezelő irányába, amely sikeres fizetés lebonyolítása esetén meghív egy megerősítésre szolgáló függvényt. Ebben a metódusban lefutnak a szükséges műveletek, mint a következő fizetési időpont beállítása, szerződésben tárolt kategória megváltoztatása és a sikeres fizetést jelző esemény létrehozása.

A már többször említett GIT tokenek felhasználási módja a következő: amennyiben kárbejelentést szeretne tenni a felhasználó és rendelkezik egy vagy több tokennel, lehetősége van ezeket felhasználni plusz kártérítési százalékokért cserébe. Ha például az ügyfél a keletkezett kár 20 százalékának megtérítésére jogosult és rendelkezik 5 GIT tokennel, amelyeket fel kíván használni, úgy érvényes kárbejelentés esetén a kár 25 százalékának megfelelő összeg kerül átutalásra a számlájára.

Az elkészült okosszerződés a teljes üzleti logikát tartalmazza, melyek az öröklődés lehetőségének köszönhetően külön állományokban találhatók. A Goerli Testnet-re [44] így csupán ez az egy okosszerződés került telepítésre, az alábbi címre: *0xb9A1d17e0251Da8bc3E60D904dd082b13e453A1D[[8]](#footnote-9).*

## Elosztott dokumentumtár

A mobilalkalmazás az elosztottság elvén alapszik. Mint korábban említettem, az üzleti logika blokklánc alapú működése csökkenti a sebezhetőséget, tekintve, hogy nincs egy központi szerver, amely leállása az alkalmazás hibás működéséhez vezethet. Mivel az applikációban állományok kezelése is elengedhetetlen – például a kárbejelentés leadása esetén feltöltendő kép, illetve dokumentum – felmerült az ötlet, miszerint ezek tárolása is történhetne elosztott módon az IPFS platform (*2.4.3 bekezdés*) segítségével. Segítette a helyzetet, hogy a Solidity nyelven íródott okosszerződések IPFS integrációja rendkívül egyszerű feladat. Csupán az IPFS azonosító eltárolására van szükség, mely egyszerű karakterlánc formájában megtehető. Ezért kezdtem el a REST API fejlesztését, mely fájlok IPFS platformon történő megosztásáért és lekérdezéséért felelős.

Ahogyan azt a 2.4.3. fejezetben is említettem, léteznek olyan cégek, amelyek távoli megjelölő szolgáltatást kínálnak a fájljaink számára, ezzel biztosítva azok redundáns tárolását. Mivel ezen szolgáltatások általában nem ingyen vehetők igénybe, ezért az általam készített dokumentumtár csak a saját csomóponton történő tárolást teszi lehetővé. Ezen a csomóponton azonban minden tárolni kívánt adat megjelölésre kerül, megakadályozva ezzel azt, hogy a szemétgyűjtő felszabadítsa azokat. Szerettem volna elkerülni továbbá azt, hogy a megosztott állományok össze-vissza szerepeljenek a csomóponton belül. Ezért, valamint az automatikus megjelölés miatt használtam az IPFS Files [45] nevezetű API-t, mely ezekre a problémákra kínál megoldást.

Használata során hasonló műveleteket végezhetünk, mint egy Linux fájlrendszer esetében [46]. Így merült fel az az ötlet, hogy az összetartozó fájlok külön-külön mappákban kerüljenek tárolásra, így átláthatóbbá téve a gyökérkönyvtárat. Általános esetben a fájlok egy hash érték segítségével kérdezhetők le, azonban az API lehetővé teszi, hogy az elérési út alapján történjen a lekérdezés, melynek szintaktikája a következő: */{mappa neve}/{fájl neve}.*

A Node.js projekt inicializálását követően pár függőség telepítésére volt szükség. Ilyen függőség az Express, amivel nagyon könnyedén definiálhatók végpontok, melyekre http kéréseket várunk. Emellett fontos elem az IPFS főkönyvtára is, mely tartalmazza a korábban említett Files API-t. Többféle IPFS disztribúció közül választhatunk, az általam fejlesztett verzió a go-ipfs [47] 0.12.2-es verzióján lett tesztelve. Elengedhetetlen még az alkalmazás futtatásához a legalább 16-os verziószámú Node.js kliens.

Az elkészült dokumentumtár a BME felhőben futva várja a beérkező http kéréseket.

### Állomány tárolására szolgáló végpont

A tárolásra szolgáló végpont POST jellegű http kéréseket vár. Ahhoz, hogy a beküldött fájlok eltárolása végbemenjen, a kérésnek néhány követelményt teljesítenie kell.

A http műveletnek tartalmaznia kell egy *id* nevű paramétert. A mobilalkalmazás során ez egy véletlenszerűen generált, 20 karakter hosszú azonosító. Fontos még emellett, hogy a kérés tartalmának a típusa *multipart/form-data* értékű legyen. Ez a fejlécben kerül beállításra. Végül, a feltölteni kívánt fájlt tartalmazó mező neve *file* kell, hogy legyen. Amennyiben ezen kritériumok mindegyike teljesül, úgy egy olyan válaszüzenet kap vissza az API használója, amely tartalmazza az eltárolt állomány hash értékét, valamint egy elérési útvonalat is, a fájl metaadatai mellett.

Abban az esetben, ha érvényes kérés érkezik a végpontra, az első dolog, amit az aszinkron lefutású függvény ellenőriz, hogy létezik-e olyan mappa a csomóponton, amely neve megegyezik a paraméterként kapott *id* mező értékével. Ha nem létezik még ilyen, úgy a megfelelő parancs segítségével létrehozásra kerül a könyvtár. Ellenkező esetben átvizsgáljuk a mappa tartalmát, megbizonyosodva arról, hogy a feltölteni kívánt fájlnévvel megegyező nevű állomány még nem szerepel a könyvtárban. Amennyiben ez valóban így van, az applikáció létrehozza azt, máskülönben létrehozás előtt hozzácsatol a fájl nevének végéhez zárójelekben egy számot, mely értéke mindig eggyel növekszik.

Ezt követően a megfelelő paraméterekkel megosztásra kerül az állomány a csomóponton. Ekkor viszont még csak a csomópont gyökerében létezik a fájl, ezért át kell azt mozgatni a felhasználó által megadott névvel rendelkező könyvtárba. Az aszinkron függvények sikeres lefutása esetén az állomány adatainak lekérése után értesítjük a felhasználót a feltöltés sikerességéről és visszaadjuk többek között a fájl elérési útvonalát is, mely segítségével később az lekérdezhető.

### Állomány lekérdezésére szolgáló végpont

Mivel a lekérdezésre szolgáló végponton nem történik adatmanipuláció, ezért a GET jellegű http kérések feldolgozása a végpont feladata. Csakúgy, mint a feltöltés esetén, itt is teljesítenie kell a bejövő kérésnek egy követelményt, mely szerint kötelező egy *path* névvel rendelkező paraméter küldése. Ezen mező értéke a sikeres fájlfeltöltés során visszakapott elérési út kell legyen. Tekintve, hogy csak egy karakter típusú paramétert tartalmaz a kérés, a fejlécek kiszámítása automatikusan megtörténik.

Érvényes paraméterrel rendelkező, GET típusú http üzenet küldése után először megtörténik a fájlnév kinyerése az elérési útvonalból. Ezt úgy kapjuk meg, ha felbontjuk az útvonalat perjelek mentén és az így kapott lista utolsó eleme lesz a fájl neve. Mivel az IPFS Files API segítségével csak a fájl tartalmát tudjuk lekérdezni bájttömbként, így szükség van egy azonos nevű és kiterjesztésű állomány létrehozására lokálisan. Miután ez megtörtént, a bájttömb tartalmát beírjuk az újonnan kreált fájlba. Ezt követően a fájl bináris formában visszaadásra kerül a válaszüzenetben, a fájlnevet pedig a megfelelő fejlécbe írjuk.

## Eseménykezelő

Az eseménykezelő egy Node.js alkalmazás, amely a BME felhőben várja a feldolgozni kívánt eseményeket. Az npm csomagkezelőnek hála több függőséget is fel tudtam használni az eseménykezelő megépítéséhez. A szerver működéséhez ezeken kívül több adatra is szükség van, melyet egy külső konfigurációs fájlban tárolunk.

Ebben a fájlban az első fontos globális változó az okosszerződést tartalmazó hálózat, a Goerli Testnet RPC csomópontja [48]. Ez egy olyan szerver, amely lehetővé teszi a hálózattal történő távoli kommunikációt. Kétféle módon lehet kapcsolatot létesíteni ezzel a csomóponttal: http és websocket protokoll segítségével. Az alkalmazás websocket protokollon keresztül kapcsolódik a Goerli Testnethez, melyben segítségemre volt az Alchemy [49] platformja. Ez egy egyedi csomópontot biztosít a blokklánccal történő kommunikációhoz, ami azért nagy segítség, mivel a fő csomópontok a Testnetekhez gyakran megbízhatatlanok, tekintve a hosszabb leállásokat.

Következő változók a telepített okosszerződés egyedi azonosítója, illetve a használt Alchemy platform API kulcsa. Annak érdekében, hogy az eseménykezelő meg tudjon hívni egy okosszerződés függvényt, szükség van a privát kulccsal történő aláírásra. Ebben az esetben viszont nem lehet minden függvényhívás előtt külső tárcához fordulni, hogy megtörténjen az aláírás, így az eseménykezelő Ethereum fiókjának privát kulcsa is elmentésre került egy változó formájában. Végül a konfigurációs állományt egy szótár zárja. Mivel az elkészült applikáció egy prototípus, a három résztvevő tesztalany – melyek a biztosítócég, illetve két felhasználó – Ethereum fiókjának publikus kulcsát hozzárendeltem az eHuf alapú tárcáikhoz. Ez azért fontos, mivel az eseményből ki tudjuk nyerni a felhasználó Ethereum tárcájának publikus kulcsát, azonban a fizetés eHuf pénznemben zajlik, így szükséges az ezek közötti átváltás. Ezek mellett egy külön fájlban az okosszerződés ABI leírója is elmentésre került.

A szerver elindítása után az első lépés az RPC csomóponton keresztüli csatlakozás a Testnethez. Ezzel létrejön egy web3 objektum, mely segítségével létrehozunk egy okosszerződés példányt, amihez a szerződés címére, illetve leírójára van szükség. Ez a web3-beli okosszerződés példány az események hallgatásáért felel. Emellett az Ether.js könyvtár segítségével szintén létrehozunk egy okosszerződés példányt, amelyekről a függvényhívások történnek majd. Fontos eleme az eseménykezelőnek továbbá az Ether.js segítségével létrehozott tárca, mely inicializálásához a konfigurációs fájlba mentett privát kulcsra van szükség. Ez a tárca fogja lehetővé tenni a tranzakciók privát kulccsal történő aláírását.

Miután létrehozásra kerültek a használni kívánt változók, megtörténik az események feliratkoztatása az alábbi kódrészlet segítségével:

insuranceTokenContract.events.MontlyFeePayment(options)

.on('data', *event* => {

const data = {

    address: *event*.returnValues.\_address,

      value: *event*.returnValues.\_value

    }

    console.log('\x1b[45m%s\x1b[0m', 'MontlyFeePayment:');

    console.log(data);

    payMonthlyFee(data.address, data.value);

})

Az első sorban a web3 könyvtár közreműködésével létrehozott szerződéspéldány eseményei közül a havidíj befizetésére szolgáló eseményt választjuk ki. Itt kérdésként felmerülhet, hogy honnan ismerjük az Ethereum hálózatán található okosszerződés metódusait, eseményeit. Mivel ezek mind szerepelnek az ABI leíró fájlban, melyből a web3 könyvtárral létrehoztam egy példányt, így úgy használhatjuk az okosszerződést, mintha csak egy JavaScriptben írt objektumról beszélnénk.

Továbbhaladva a következő sorra, az *on* függvény első paramétereként megadhatjuk, milyen esetben szeretnénk lefuttatni a második paraméterben megfogalmazott függvényt. Négy lehetőség közül választhatunk. Ha a *data* opciót választjuk, akkor minden alkalommal, amikor egy ilyen esemény keletkezik, lefut az utána lévő függvény, *changed* állapotban csak akkor, ha a hallgatott esemény eltávolításra kerül a blokkláncról, *error* esetében akkor, amennyiben hibával tér vissza az esemény, a *connected* kulcsszót használva pedig egyszer fut le a függvény, csatlakozáskor.

Ezt követően a függvény paramétereként kapott objektum tartalmazni fog több adattagot, köztük az esemény visszatérési értékeit is, melyek ebben az esetben a befizető címe, illetve a befizetni kívánt havidíj. Ezután kiírjuk a keletkezett esemény nevét, illetve a korábban kreált, visszatérési értékeket tartalmazó objektumpéldányt is. Az kiírás első paramétereként látható karakterlánc a háttérszínt módosítja az átláthatóság érdekében.

Abban az esetben, ha szükség van fizetés megerősítésre is, továbbhívás történik a fizetést feldolgozó, eseménykezelőben implementált metódusra, amely ebben a példában a havidíj befizetésére szolgál.

Ebben a függvényben kommunikál az eseménykezelő a fizetési átjáróval, http kéréseken keresztül. Első lépésként minden esetben szükséges lekérni az úgynevezett *nonce* számot, mely egy minden – Fabricon értelmezett – címhez egyedi értékkel rendelkező, automatikusan növekvő változó. Miután kiderült ez a szám, következő lépésként az aláírásgenerálás történik. Ehhez ugyan azt az algoritmust használom, melyet a fizetési átjáró. Rendelkezésre áll a küldő címe, fogadó címe, átutalni kívánt összeg és *nonce* változó értéke is. Miután megtörtént az aláírás generálása, az előbb említett paraméterekkel, valamint az aláírás elemeivel (melyet a *4.5. fejezetben* bővebben kifejtek) megtörténik a pénzösszeg átutalására szolgáló fizetés átjáró által implementált függvény meghívása. Sikeres fizetés esetén hasonló módon, ahogyan az események esetében tettük, meghívjuk az okosszerződés fizetés megerősítő metódusát, hogy érvényesítésre kerüljön a havidíj befizetésének hatása. Sikeres havidíj befizetés után az alábbi szöveget láthatjuk (lásd 11. ábra).

A képen szöveg látható

Automatikusan generált leírás

11. ábra: Havidíj befizetése után keletkező események

## Fizetési átjáró

A fizetési átjáró, valamint a 4.6. bekezdésben található CBDC réteg nem általam létrehozott termékek, az egyetem és a Magyar Nemzeti Bank közös, CBDC alapú projektje során jöttek létre. Ennek ellenére az alkalmazásomba történő CBDC integráció során én is használtam az alábbi komponenseket, ezért úgy döntöttem, hogy röviden bemutatom a működésüket.

A fizetési átjáró egy REST API, mely segítségével kommunikálhatunk a Hyperledger Fabric blokkláncán található CBDC programmal. Az API szintén a BME Felhőben található. Ahhoz, hogy tranzakciókat tudjunk bonyolítani segítségével, szükséges az eHuf pénznemmel rendelkező fiókok létrehozása. Az alkalmazásom esetében három teszt felhasználót regisztráltam, melyek közül az egyik a biztosítócég címe, a másik kettő pedig ügyfelekhez tartozik. A generálás folyamata egy szkript[[9]](#footnote-10) meghívásával kezdődött, mely után a generált értékek – melyek a privát kulcs, publikus kulcs és a blokkláncon értelmezett cím – egy JSON leíró fájlban található tömbbe hozzáadásra kerültek.

Mivel a CBDC réteg blokklánc hálózaton fut, ezért az itt lévő metódusok meghívásához az azonosítás érdekében privát kulccsal történő aláírás szükséges. Az átjáró összes releváns függvénye többek között vár egy *r, s* és *v* értéket is. Ezen paraméterek az aláírásgenerálás során kaphatók meg. Az aláírás ECDSA algoritmust használ, az *r* és *s* értékek a kalkulált aláírás részei. Emellett a *v* érték egy olyan szükséges szám, amely segítségével az aláírásból a publikus kulcs biztonságosan visszaállítható, ugyanis bár nagyon kis eséllyel, de előfordulhat ezen algoritmus használata esetében, hogy az aláíráshoz több publikus kulcs is rendelhető.

A generálás elvégzéséhez szintén egy szkript áll rendelkezésünkre. Első paraméterként minden esetben a már említett JSON leíró fájlban található cím indexét kell megadni, mely után sorban a tranzakció paraméterei következnek. Tárcák közötti utalás esetében ilyenek például a forráscím, célcím és az érték. Miután megadtuk a várt paramétereket, a szkript lefuttatása után megkapjuk a várt *r, s* és *v* értékeket.

Mint említettem, a fizetési átjáró egy REST API, mely végpontjaira http kérések érkezhetnek. A Swagger[[10]](#footnote-11) dokumentációból [50] deríthetjük ki, hogy mely végpontok milyen http kéréseket és paramétereket várnak. Amennyiben minden szükséges paraméterrel rendelkezünk és sikerült az aláírásgenerálás is, megtörténhet a tranzakció lebonyolítása. Az aláírás ellenőrzésre kerül a CBDC oldalon, majd, ha minden érték helyes, sikeresen végbemegy a tranzakció értesítve erről a felhasználót.

## CBDC réteg

A CBDC réteg felelős a digitális jegybankpénzben történő műveletek megvalósításáért. Az eHuf valutát reprezentáló okosszerződések TypeScript nyelven íródtak. A nyelv nagyban hasonlít a JavaScript-hez, annyi különbséggel, hogy támogatja a típusok létrehozását és deklarálását. A Fabric API segítségével történik meg a kommunikáció a blokklánccal. A BME felhőben található, a logikát megvalósító okosszerződés függvényei hasonló logika alapján vannak felépítve. A tranzakciók minden esetben várnak első paraméterként egy indexet, mely a hálózatba regisztrált, a fiókokat tároló tömbben való elhelyezkedésüket mutatja meg. Ezt követően kell megadni az egyéb paramétereket, köztük az r, s és v értékeket, amelyek fontosságát a *4.5. bekezdésben* ismertettem.

Ezután következik a függvények törzse, mely kezdetben ellenőrző függvényekből áll. Ilyen metódusok például az aláírás ellenőrzése és az elegendő egyenleg vizsgálata. Miután ezek megtörténtek, az üzleti logika implementálásra kerül. Ehhez szintén nagy segítség a Fabric által biztosított API. A függvény törzse egy esemény létrehozásával zárul, amit egy, az okosszerződés címére konfigurált eseménykezelőből figyelhetünk. Amennyiben akármilyen hiba történik a függvény futása során, arról hibaüzenet formájában értesül a felhasználó.

Szakdolgozatom szempontjából az okosszerződése négy metódusa releváns. Először fiókok létrehozásával kezdtem, majd kibocsátottam a létrehozott fiókoknak a tesztelés segítése érdekében eHuf valutákat. A fizetést igénylő metódusok pedig a digitális pénzek mozgatásáért felelős függvényt használták, melyhez szükség volt a *nonce* szám lekérdezésére is.

# Összefoglalás

Az alábbi bekezdésben az elkészült alkalmazásról fogalmazok meg egy pár gondolatot, majd kitérek az applikáció továbbfejlesztési potenciáljaira is. A kész mobilalkalmazás Android operációs rendszeren lett tesztelve, a nyilvános forráskód megtalálható a Függelékben.

## Alkalmazás értékelése

Az alkalmazásfejlesztés során elkerülhetetlen egy alapos terv elkészítése. Eddigi tanulmányaim során minden elkészült applikáció közül most töltöttem a legtöbb időt az alkalmazás megtervezésével. Úgy gondolom, hogy ez nagyon jó döntésnek bizonyult, hiszen ez észrevehetően megkönnyítette és gyorsabbá tette a fejlesztés folyamatát. Ezen kívül sokat segített a már elkészült, blokklánc alapú biztosítást kínáló alkalmazások megvizsgálása is.

Összességében elégedett vagyok az elkészült végtermékkel. Ugyan szerepelt a kezdeti tervben egy olyan elem, amely megvalósítása sajnos nem sikerült (ügyfél életkorának hitelesítése), ennek ellenére úgy gondolom, hogy egy jól kezelhető, átlátható alkalmazást sikerült megvalósítanom. Az alkalmazásfejlesztés során nagyon fontosnak éreztem, hogy modern, időtálló technológiákkal tudjak dolgozni. Bár nem teljesen ismeretlen a blokklánc és a CBDC fogalma a modern informatikában, úgy gondolom, hogy még sok lehetőséget tartogat magában, melyek segítségével a közeljövőben felveheti a versenyt az centralizált, központi szerverektől függő applikációkkal szemben. Ugyan a fejlesztés időigényesebb, a rendszer megtervezése bonyolultabb lehet, azonban a tapasztalat növekedésével arányosan kialakulhatnak olyan tervezési minták, melyek felgyorsíthatják, egyszerűbbé tehetik az elosztott adatbázist használó rendszerek implementálását.

A legbüszkébb az elkészült termék esetében a komponensek egymással történő kommunikációjára vagyok. Számos technológiát használtam, melyek egyenként önmagukban működő komponenseket alkotnak. Ilyenek a két különböző blokkláncon futó okosszerződések, az elosztott dokumentumtár, a felhasználói felület is. Ennek ellenére a komponensek elkészítése után az ezek egymással történő kommunikációja elég időigényes volt, szükség volt a rendszer működésének teljes átlátására. Köszönhetően ennek, illetve a számos fejlesztői eszköznek, sikerült egy kész prototípus megalkotása.

Az elkészült végtermék mindezek ellenére nem tekinthető késznek. A több komponens összekapcsolásának következtében az alkalmazás működése lassabb, mint egy centralizált adatbázissal rendelkező applikáció. Amíg nagyobb felhasználói élményt tudnak elérni a központi adatbázisok köré épülő alkalmazások, hiába a nagyobb biztonság és automatizálás a blokkláncok esetében, nagy százalékban a gyorsabb, kevésbé biztonságos, centralizált termék mellett fognak dönteni a felhasználók. A kód optimalizálása nem csak a sebesség növelése, de a felhasználók pénzének megóvása miatt is fontos. Minél kevésbé optimalizált egy okosszerződés, annál nagyobb költségbe fog kerülni a függvényeinek meghívása. Úgy gondolom az applikációmban minden stratégiát bevetve egy valamivel optimálisabb kód is létrejöhetne. Ezek mellett természetesen hiányzik a biztosításkötés jogi oldala is, mely modellezése túlmutat az elkészült prototípus keretein.

## Továbbfejlesztési lehetőségek

Egy blokklánc alapú, digitális jegybankpénzt használó biztosítás esetében rengeteg továbbfejlesztési lehetőség létezik. Sikerült megvalósítanom egy olyan rendszert, mely *utility tokenekkel* jutalmazza a felhasználókat a károk elbírálásában nyújtott szerepükért. Ez azért fontos az alkalmazás továbbfejlesztése szempontjából, hiszen számos ötlet felmerülhet ezek felhasználása során. Egy opció lehet a személyre szabott, egyedi biztosítások kötése. Ezek mellett integrálható lehet az üzleti logikába egy barátok meghívására szolgáló komponens is, mely után tokenekkel jutalmazzuk a meghívó és a meghívott felet is egyaránt.

A legnagyobb potenciált rejtő továbbfejlesztési lehetőség azonban egy lakóközösségek számára kialakítható, közös biztosítás lehet. Ebben az esetben egy lakóközösség – legyen az akár egy egész lakóház, akár csak egy szint – létrehozhat egy közös biztosítást, mely során minden hónapban a befizetett havidíjak egy közös CBDC fiókra kerülnének. Ez azért fontos, mivel kár esetén a fiókról történne a kifizetés, ezzel biztosítva a felhasználókat arról, hogy mindig lesz elegendő fedezet a káresetek kifizetésére. Emellett a felhasználók csoportos kedvezményekben is részesülhetnek.

# Irodalomjegyzék

1. Energy consumption in households, <https://ec.europa.eu/eurostat/statistics-explained/index.php?title=Energy_consumption_in_households>
2. History of insurance in America, [https://www.investopedia.com/articles/financial-theory/08/american-insurance.asp#citation-6](https://www.investopedia.com/articles/financial-theory/08/american-insurance.asp%23citation-6)
3. Overall cryptocurrency market capitalization per week from July 2010 to November 2022, <https://www.statista.com/statistics/730876/cryptocurrency-maket-value/>
4. Consensus mechanisms, <https://ethereum.org/en/developers/docs/consensus-mechanisms/>
5. What is Proof of Stake?, [https://ethereum.org/en/developers/docs/consensus-mechanisms/pos/#what-is-pos](https://ethereum.org/en/developers/docs/consensus-mechanisms/pos/%23what-is-pos)
6. ERC20 Token Standard, <https://ethereum.org/en/developers/docs/standards/tokens/erc-20/>
7. ERC721 Token Standard, <https://ethereum.org/en/developers/docs/standards/tokens/erc-721/>
8. What Is Central Bank Digital Currency?, <https://consensys.net/solutions/payments-and-money/cbdc/>
9. Hyperledger Cactus whitepaper, <https://github.com/hyperledger/cactus/blob/main/whitepaper/whitepaper.md>
10. Central Bank Digital Currency Tracker, <https://www.atlanticcouncil.org/cbdctracker/>
11. Sand Dollar, the Digital Bahamian Dollar, <https://www.sanddollar.bs/>
12. Blockchain in Insurance Market, <https://www.globenewswire.com/en/news-release/2022/06/21/2466343/0/en/Blockchain-in-Insurance-Market-is-estimated-to-be-US-39-5-billion-by-2030-with-a-CAGR-of-72-4-during-the-forecast-period-By-PMI.html>
13. AXA insurance company, <https://www.axa.co.uk/>
14. AXA goes blockchain with fizzy, <https://www.axa.com/en/magazine/axa-goes-blockchain-with-fizzy>
15. AXA Scraps Fizzy Insurance Smart Contract, <https://www.artificiallawyer.com/2020/10/08/axa-scraps-fizzy-insurance-smart-contract-but-still-interested-in-the-tech/>
16. Growing insurance consortium eyes industry-wide blockchain, <https://www.reuters.com/article/us-insurance-blockchain-swiss-re-idINKBN15924K>
17. B3i confirms swich away from blockchain to R3’s Corda, <https://www.reinsurancene.ws/b3i-confirms-switch-away-from-blockchain-to-r3s-corda/>
18. B3i fails to raise new capital, enters insolvency, <https://www.reinsurancene.ws/b3i-fails-to-raise-new-capital-enters-insolvency/>
19. State of the sector: agri-insurance for smallholder farmers, <https://isfadvisors.org/wp-content/uploads/2022/03/ISF_Agri-insurance-for-smallholder-farmers_March-2022.pdf>
20. Introduce the Lemonade Crypto Climate Coalition, <https://www.lemonade.com/blog/crypto-climate-coalition/>
21. Lemonade to use blockchain to deliver “affordable and instantaneous” climate insurance, <https://www.reinsurancene.ws/lemonade-to-use-blockchain-to-deliver-affordable-and-instantaneous-climate-insurance/>
22. Stablecoins: Definition, How They Work, and Types, <https://www.investopedia.com/terms/s/stablecoin.asp>
23. Cross-platform mobile frameworks used by software developers worldwide from 2019 to 2021, <https://www.statista.com/statistics/869224/worldwide-software-developer-working-hours/>
24. The official package repository for Dart and Flutter apps, <https://pub.dev/>
25. Ethereum whitepaper, Vitalik Buterin, 2014, <https://ethereum.org/en/whitepaper/>
26. Remix IDE, The Native IDE for Web3 Development., <https://remix.ethereum.org/>
27. Top 10 Solidity Gas Optimization Techniques, <https://www.alchemy.com/overviews/solidity-gas-optimization>
28. Javascript package manager, npm, <https://www.npmjs.com/>
29. A crypto wallet & gateway to blockchain apps, <https://metamask.io/>
30. Application Binary Interface in a nutshell, <https://fourweekmba.com/application-binary-interface/>
31. Permissioned vs. permissionless blockchains: Key differences, <https://www.techtarget.com/searchcio/tip/Permissioned-vs-permissionless-blockchains-Key-differences>
32. Connect Metamask with a native mobile app built in Flutter, [https://github.com/MetaMask/metamask-mobile/issues/3735#issuecomment-1041351004](https://github.com/MetaMask/metamask-mobile/issues/3735%23issuecomment-1041351004)
33. How gas fees work ont he Ethereum Blockchain, <https://www.investopedia.com/terms/g/gas-ethereum.asp>
34. What is self-sovereign identity?, <https://sovrin.org/faq/what-is-self-sovereign-identity/>
35. Self-soverign Identity, How big is the market oppurtunity?, <https://cheqd.io/hubfs/SSI%20market%20cheqd%20report.pdf>
36. Verifiable Credentials Data Model, <https://www.w3.org/TR/vc-data-model/>
37. Cryptozombies code school for Ethereum development, <https://cryptozombies.io/en/course>
38. Solidty – abstract smart contract[, https://www.geeksforgeeks.org/solidity-abstract-contract/](https://d.docs.live.net/8650d21511cae414/Dokumentumok/szakdolgozat/,%20https:/www.geeksforgeeks.org/solidity-abstract-contract/)
39. OpenZeppelin – Ownable smart contract, <https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/access/Ownable.sol>
40. OpenZeppelin – ERC20 smart contract, <https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC20/ERC20.sol>
41. Four eyes principle, <https://ec.europa.eu/eurostat/cros/content/four-eyes-principle_en>
42. What is SHA-256?, <https://blog.boot.dev/cryptography/how-sha-2-works-step-by-step-sha-256/>
43. Random Number Generation (RNG) in Solidity, <https://blog.chain.link/random-number-generation-solidity/>
44. Goerli Testnet for Ethereum, <https://goerli.net/>
45. IPFS Files API, <https://github.com/ipfs/js-ipfs/blob/master/docs/core-api/FILES.md>
46. Linux file system commands, <https://doc.nuxeo.com/nxdoc/filesystem-commands/>
47. Go-IPFS versions, <https://dist.ipfs.io/go-ipfs/>
48. RPC node, <https://help.coinbase.com/en/coinbase/getting-started/crypto-education/glossary/rpc-node>
49. Alchemy web3 development platform, <https://www.alchemy.com/>
50. CBDC Payment Gateway, [http://vm.niif.cloud.bme.hu:9200/swagger-ui/#](http://vm.niif.cloud.bme.hu:9200/swagger-ui/)

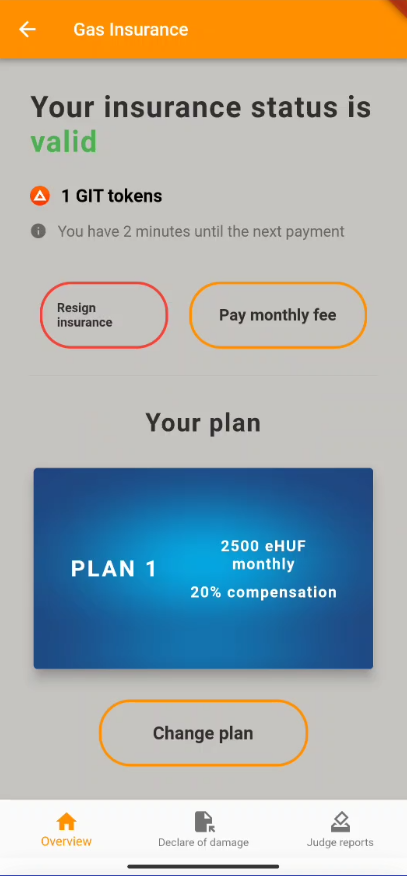
# Függelék

Az alkalmazást tartalmazó Github repository: <https://github.com/pelyhe/thesis>

|  |  |
| --- | --- |
| 12. ábra: A tárca csatlakoztatásáért felelős képernyő | 13. ábra: Az adatok lekérése közben látható töltőképernyő |

|  |
| --- |
|  |
| 14. ábra: Kibocsátott bizonyítvány kártyák |

|  |  |
| --- | --- |
|  |  |
| 15. - 16. ábra: Biztosítás kötése képernyő | |



17. ábra: Főoldal



18. ábra: Biztosítási kategória változtatása nézet

|  |  |
| --- | --- |
| A képen szöveg látható  Automatikusan generált leírás | A képen szöveg látható  Automatikusan generált leírás |
| 19. - 20. ábra: Kárbejelentő nézet | |

A képen szöveg, berendezés, konyhai felszerelés látható

Automatikusan generált leírás

21. ábra: Kár elbírálása nézet

1. REST API: http kérések segítségével kommunikáló, adatok lekérdezésére, manipulálására használt interfész [↑](#footnote-ref-2)
2. frontend fejlesztés: felhasználó számára látható felület programozása [↑](#footnote-ref-3)
3. hash algoritmus: különböző bemenetekre különböző, azonos bemenetekre azonos kimenetet produkáló függvény [↑](#footnote-ref-4)
4. backend szerver: a felhasználó elől elrejtett, üzleti logikát tartalmazó, adatbázist manipuláló szerver [↑](#footnote-ref-5)
5. utility token: az alkalmazás használatáért cserében kapott token, melyek valamilyen kedvezményre felhasználhatók. Részletesebben lásd *4.2.3. bekezdés*. [↑](#footnote-ref-6)
6. singleton minta: olyan tervezési minta, mely során az adott osztályból egy, statikus példány hozható létre [↑](#footnote-ref-7)
7. türelmi időszak: a fizetés határidejének lejárta után ennyi ideig még nem függeszthető fel a biztosítás [↑](#footnote-ref-8)
8. A szerződés azonosítója a Goerli Testneten, mely segítségével megtudhatunk pár információt a telepített okosszerződésről a felfedező (<https://goerli.etherscan.io/>) segítéségével [↑](#footnote-ref-9)
9. szkript: valamilyen programozási nyelvben (jelen esetben JavaScript) írt futtatható utasítássorozat [↑](#footnote-ref-10)
10. Swagger: REST API-k fejlesztésére és dokumentálására használt fejlesztői eszköz, <https://swagger.io/> [↑](#footnote-ref-11)