Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar

Piller Trisztán

FÓRUM ALKALMAZÁS FEJLESZTÉSE AZURE SERVERLESS KÖRNYEZETBEN

KONZULENS

Tóth Tibor

BUDAPEST, 2022

Tartalomjegyzék

HALLGATÓI NYILATKOZAT

Alulírott **Piller Trisztán**, 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, 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. 08.

.....

Piller Trisztán

Összefoglaló

Egy üzletileg is életképes szoftveres megoldás tervezésénél fontos szerepet vállalhat az, hogy milyen platformon szeretnénk felépíteni és a végén üzemeltetni a rendszert. A választott irány meghatározhatja a fejlesztés és telepítés hatékonyságát, valamint piacra kerülésének gyorsaságát.

A piac egyre inkább a felhő alapú számítástechnika lehetőségei felé terelődik. A felhő alapú platformok dinamikus fejlődést mutatnak, általuk egyre szélesebb körben válnak elérhetővé szolgáltatások a fejlesztők felé, az informatika szinte minden területén alkalmazásra kerülnek felhőplatformok.

Szakdolgozatom célja demonstrálni egy Microsoft Azure felhőszolgáltatásokon alapuló webalkalmazáson keresztül a platform adta lehetőségeket és kihívásokat. Alkalmazásom, a Re:mark egy közösségi fórum, amelynek felhasználói szabadon oszthatják meg és vitathatják meg egymással gondolataikat.

Elejétől végéig bemutatásra kerül a TypeScript alapú full-stack serverless webalkalmazás, amely fejlesztéséhez számos izgalmas és élvonalbeli technológiát alkalmaztam, infrastruktúrájának menedzselésére pedig Terraformot használtam.

Abstract

When architecting a commercially viable software solution it might play an important role, on what type of platform we want to build and eventually operate the system. The chosen direction can define the efficiency of development and deployment, also the speed of entering the market.

The market is headed increasingly towards using the possibilities of cloud computing. Cloud based platforms show a dynamic development, the range of available services for developers is getting wider and wider, we can see cloud platforms used in every field of IT.

The aim of my thesis is to demonstrate possibilities and challenges given by the platform through a Microsoft Azure cloud services based web application. The application called Re:mark is a community forum on which users can share and debate ideas with each other.

I present the TypeScript based full stack serverless application from start to finish. For its development I used numerous exciting and cutting-edge technologies, for managing its infrastructure I used Terraform.

1 Bevezetés

1.1 Motiváció

Dolgozatomban – az általam *Re:mark* névvel ellátott – közösségi fórumként szolgáló webalkalmazás fejlesztését mutatom be.

Ma már szinte mindenhonnan körbe övezik az embert webes szolgáltatások, ezek közé tartoznak az internetes közösségi média oldalak és webes fórumok. A *Re:mark* a *Reddit* [1] weboldalból merít ötleteket.

Webalkalmazásom segítségével a felhasználók képesek csatornákon keresztül posztbejegyzéseket készíteni, amelyeken további véleményformálás kommentek formájában, illetve le- és felszavazással lehetséges. Használata során az egyes interakciókról böngészőben megjelenő értesítéseket kap a felhasználó.

Számos rendszerkomponensből áll össze, a komponensek felelősségei közé tartoznak a következők: felhasználók és hozzáférési jogaik kezelése, képi és szöveges erőforrások tárolása, valamint felület biztosítása értesítések kezelésére és az erőforrások manipulálására.

A projektet egy részről az motiválta, hogy betekintést szerezzek webes fórumok fejlesztése során előkerülő kihívásokba. Más részről pedig az, hogy tapasztalatot szerezhessek széles technológiai palettával rendelkező projektek fejlesztésében és ilyen rendszerek felhőben való üzemeltetésében.

1.2 A feladat körvonalazása

A bemutatásban szereplő webalkalmazás serverless [2] szolgáltatásokra épülő mikroszolgáltatásos rendszerben került megvalósításra.

A munkát a követelmények pontos kialakításával kezdtem, megrajzoltam a felhasználói felület vázát.

Többrétegű rendszerarchitektúrát követtem: a kliens oldali rétegben, illetve a szerver oldali rétegekben használt technológiák megválasztása után megterveztem a rendszer moduláris szintű modelljét, illetve definiáltam a legfontosabb rendszer szintű folyamatok kommunikációs modelljét.

Az implementáció *JavaScript* technológiák felhasználásával került kivitelezésre, ügyelve a magas fokú esztétikai és felhasználói élmény nyújtására és a reszponzív felhasználói felület tervezési szemléletre.

Az üzemeltetést alapos megfontolásokkal választott *Microsoft Azure* [3] serverless szolgáltatásokkal és *Terraform* [4] használatával oldottam meg.

1.3 Szakterületi kihívások

A felhőben való fejlesztésnek több kihívása is elém tárult. Ezek közé tartozik a választott Microsoft Azure felhőplatform lehetőségeinek felfedezése, a választott felhőszolgáltatások menedzsmentje, ezen szolgáltatások felkonfigurálása saját céljaimnak megfelelően.

A szolgáltatások megválasztásánál több szempontot kellett megvizsgáljak. A legfontosabbak közé tartozik: a szolgáltatás gazdaságossága, a biztonság, illetve a megbízhatóság teljesítmény és elérhetőség szempontjából. Nem elhanyagolható szerepet vállal egy szolgáltatás megválasztásánál, hogy milyen szinten segíti elő a mérnököket az üzemeltetés leegyszerűsítésében. Ide kapcsolódik, hogy például mennyiben csökkenti a szükséges erőfeszítéseket a jövőbeli forgalomváltozások tekintetében egy saját üzemeltetésű szerverrel szemben.

A *DevOps*[1] világában egészen újnak és feltörekvőnek számít az *IaC (Infrastructure-as-Code)*[2] [5] alkalmazása. Ezt a gyakorlatot is beemeltem az üzemeltetési kihívások közé a Terraform szoftver használatával.

A kihívásoknak köszönhetően sokat tanulhattam a felhőben való fejlesztés előnyeiről, a serverless szolgáltatások jellemzőiről.

[1] Developer Operations: azon feladatokat, eszközöket és gyakorlatokat foglalja össze, amelyek rendszerek fejlesztéséhez és üzemeltetéséhez használatosak.

[2] Üzemeltetési gyakorlat, nyersen fordítva: „Infrastruktúra mint kód”. Egy kiterjedt rendszer erőforrásainak kódból való definiálása és felkonfigurálása.

2 Feladat specifikáció

A projektfeladat specifikációját funkcionális és nem funkcionális követelményekkel lehet leírni. A funkcionális követelmények arra szolgálnak, hogy bemutassák, a rendszernek milyen működésbeli funkciókat, szolgáltatásokat kell biztosítania a felhasználók felé. A nem funkcionális követelmények pedig arra szolgálnak, hogy bemutassák, hogy a rendszernek milyen működési feltételekkel kell rendelkezni, milyen minőségi tulajdonságokkal kell rendelkeznie.

2.1 Funkcionális követelmények

2.1.1 Felhasználókezelés

A felhasználó képes már létező Google felhasználói fiókjával regisztrálni, illetve bejelentkezni a rendszerbe. A rendszer a felhasználói fiók adatait (név, e-mail cím) tárolja. A felhasználó fiókjához képes profilképet feltölteni.

2.1.2 Csatornák kezelése

A csatorna egy tematikus csoport címmel, leírással, amely gyűjti a csatorna követőit, illetve az ide közölt posztbejegyzéseket. A csatorna leírása Markdown nyelven írható, annak megfelelően kerül formázásra. A csatornába való posztoláshoz a felhasználónak regisztrálnia kell a rendszerbe, illetve a csatornát követnie kell.

A csatornát létrehozó felhasználó (tulajdonos) a csatornát bármikor törölheti, a leírást módosíthatja. Kioszthat a követők között moderátori szerepkört, amely lehetővé teszi a csatorna posztjainak módosítását, illetve törlését. A tulajdonos a moderátori jogot bármikor visszavonhatja.

2.1.3 Posztok kezelése

A poszt egy szöveges bejegyzés címmel és tartalommal. A poszt tartalma Markdown nyelven írható, annak megfelelően kerül formázásra. A posztot a csatorna követői fel- és leszavazhatják, ezzel kifejezve tetszésüket. Egy posztot egyszer képes egy felhasználó vagy fel-, vagy leszavazni.

Posztot egy csatorna alá a csatorna követői posztolhatnak. A poszt tulajdonosa bármikor módosíthatja, illetve törölheti a posztot, illetve adhat hozzá képet.

2.1.4 Kommentek kezelése

A komment egy szöveges hozzászólás egy poszthoz. A komment tartalma Markdown nyelven írható, annak megfelelően kerül formázásra. A kommentet a poszt csatornájának követői fel- és leszavazhatják, ezzel kifejezve tetszésük. Egy kommentet egyszer képes egy felhasználó vagy fel-, vagy leszavazni.

Komment akkor írható poszt alá, ha a poszt csatornáját már követjük. A komment tulajdonosa bármikor módosíthatja, illetve törölheti a kommentet. A poszt csatornájának moderátorai módosíthatják, illetve törölhetik a kommentet.

2.1.5 Értesítések kezelése

A weboldalon valós időben értesítéseket kap a felhasználó mindenféle eseménnyel kapcsolatosan. Értesítést előidéző események a következők:

- Moderátor szerepkört kaptál egy csatornában
- Csatlakoztak a csatornádhoz
- Elvették a moderátorjogod a csatornából
- Kommentet fűztek a posztodhoz
- Szavazatot kapott a kommented
- Új poszt készült a csatornában, amit követsz
- Új poszt készült a csatornában, aminek moderátora/tulajdonosa vagy
- Szavazatot kapott a posztod
- Törölték a csatornát, amelynek tagja voltál.

2.2 Nem funkcionális követelmények

2.2.1 Üzemeltetés

A mikroszolgáltatások szoftverarchitektúra előnye, hogy magas rendelkezésre állású, stabil, tranziens hibáknak ellenálló, jól skálázható rendszereket építhetünk. Jól eloszlanak a felelősségek a komponensek között, valamint biztosítható akár még az is, hogy idővel a komponenseket kicseréljük, csupán jól megtervezett API szerződésekre van szükségünk a részegységek között.

Az alkalmazás üzleti logikáját úgy kell megtervezni, hogy az jól elkülöníthető felelősségi körökkel rendelkező alrendszerekre, azaz mikroszolgáltatásokra bontható

lehessen. Valamint ezen követelményeket kiszolgáló üzemeltetési platformot és eszközöket kell választani.

A felhasználó böngészőn keresztül egy weboldalon keresztül képes elérni az alkalmazás felületét és funkcióit.

2.2.2 Konfiguráció menedzsment

Az *Infrastructure-as-code* (IaC) szemlélet a szoftverfejlesztésben nagy segítséget tud nyújtani. Lehetővé teszi komplex szoftveres és hardveres infrastruktúránk kódból való telepítését és felkonfigurálását. Így a kód és a konfiguráció együtt képes fejlődni a kezünk alatt, és könnyen követhető lesz a változások története egy kódbázis verziókezelő rendszer alkalmazásával.

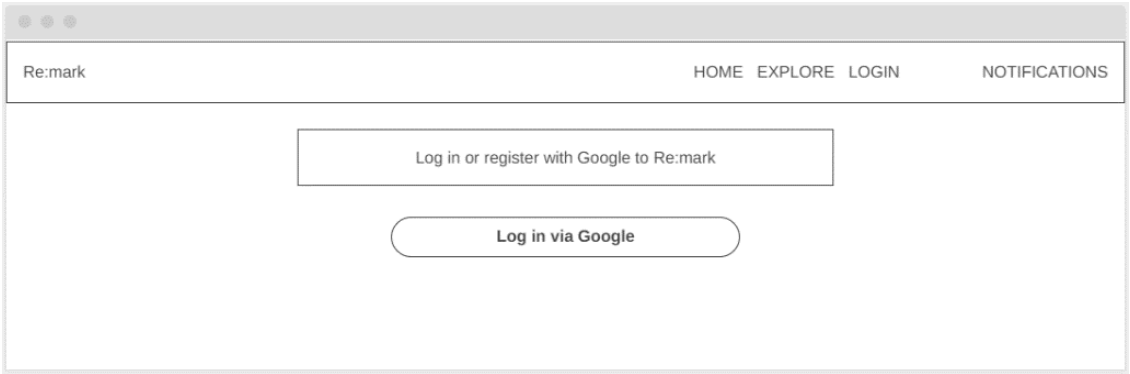
Egy ilyen IaC eszköz segítségével kell üzemeltetésre és menedzselésre kerüljön a Re:mark szoftver minden infrastrukturális eleme és konfigurálása.

2.3 Wireframe-ek

A wireframe (drótváz) a felhasználói felület tervezésének egy eszköze. A wireframe-ek leegyszerűsített rajzok a felületről, a hasznosságot teszik előtérbe, segítségükkel átlátható lesz a felhasználói felület struktúrája, felhasználása. Minden fél számára egyértelműsíti a weboldal egyes nézeteinek a jellemzőit.

2.3.1 Bejelentkezés

A bejelentkezés és regisztráció egyben történik meg. Ha még nem volt felhasználói fiókunk Re:markon, akkor egyszerűen a háttérben létrejön a fiók, így nincs szükség regisztrációs felületre (1. ábra).



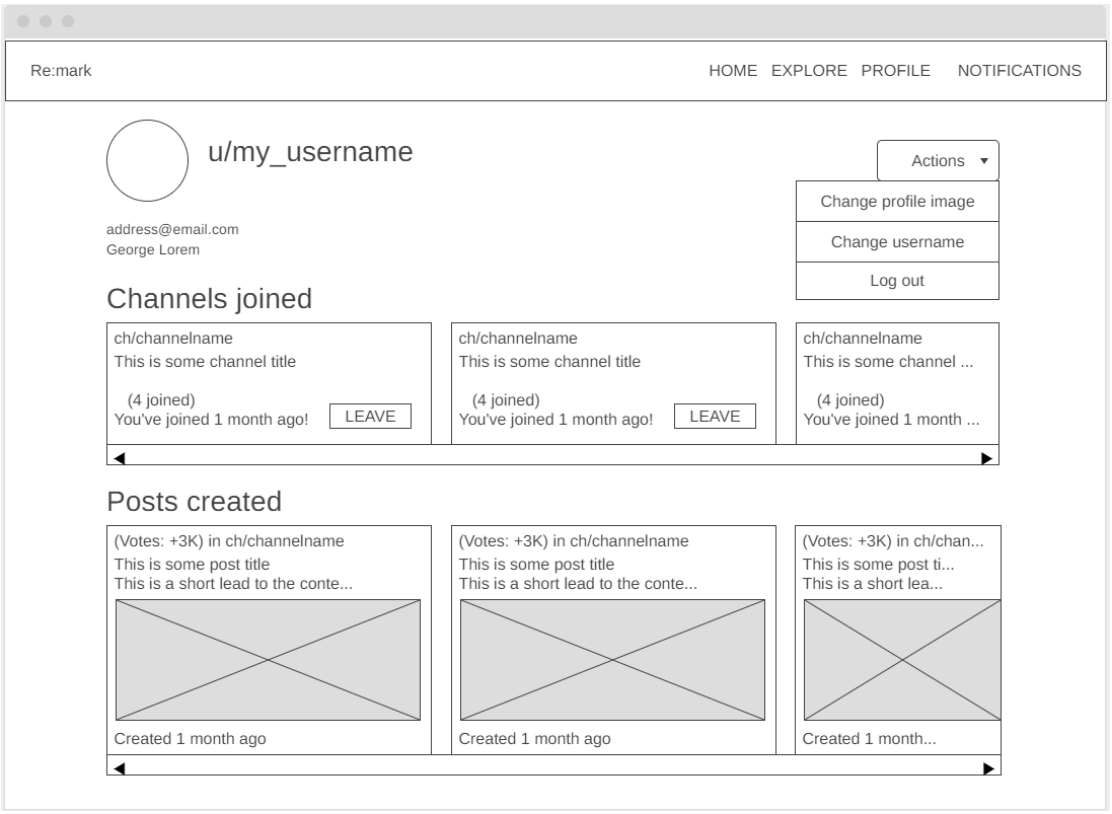
1. ábra. Bejelentkezési oldal nézete

2.3.2 Profil nézete

A profilunkban láthatjuk a választott profilképünk és mellette a felhasználói nevet, alatta az e-mail címünket és nevünket. Az általunk követett csatornák és készített posztjaink pedig horizontálisan görgethető kártyasorokban kerülnek elhelyezésre. A horizontálisan görgethető kártyasorok által kompakt marad az oldal, egyben látható minden szekciócím. A posztoknak a kártyáin a képek kapnak kiemelt szerepet, hogy a posztok maguk könnyen beazonosíthatóak legyenek.

Ha a bejelentkezett felhasználó profilját látjuk (azaz a sajátunkat), abban az esetben egy lenyitható menü is megjelenik a jobb felső sarokban, amelyben felhasználónév és profilkép módosítási, illetve kijelentkezési lehetőségek kerülnek elő. Így az oldalon elérhető akciók egy helyen kerülnek összegyűjtésre. Ebben az esetben mellesleg a csatorna kártyáin „Leave” gomb is található, ha megszüntetni kívánnánk a követését a csatornának.

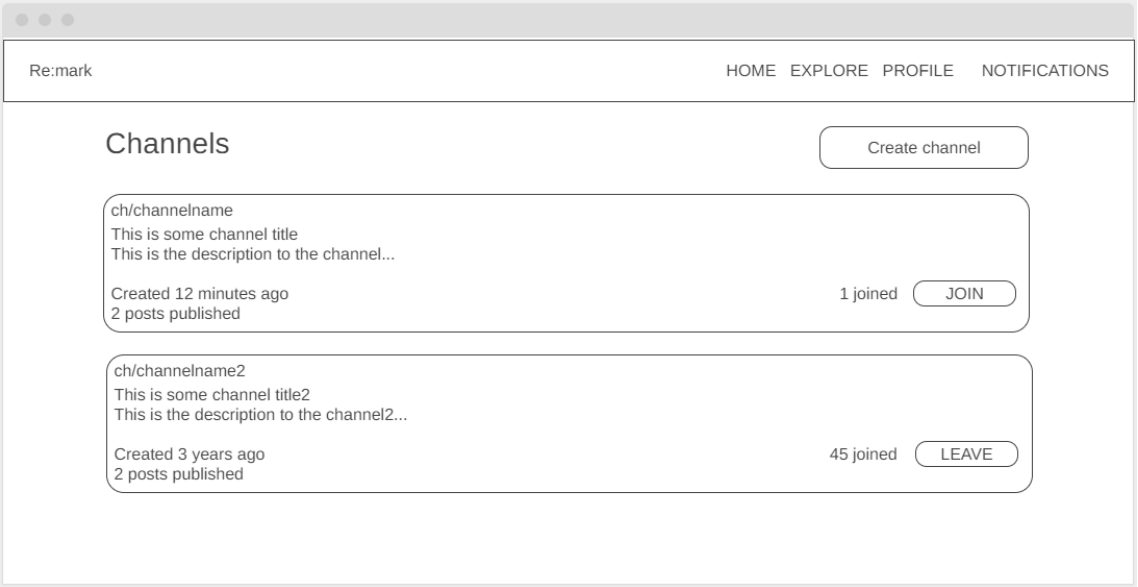
A leírásnak megfelelő wireframe-et a 2. ábra mutatja be.



2. ábra. Profil oldal nézete

2.3.3 Csatornák nézete

A kezdőlapon kap helyet a csatornák listázási nézete. A csatornák kattintható kártyákként jelennek, alapadataikat tartalmazzák és a követési gombot láthatjuk rajtuk (3. ábra).



3. ábra. Csatornák listázási nézete

2.3.4 Csatornán található posztok nézete

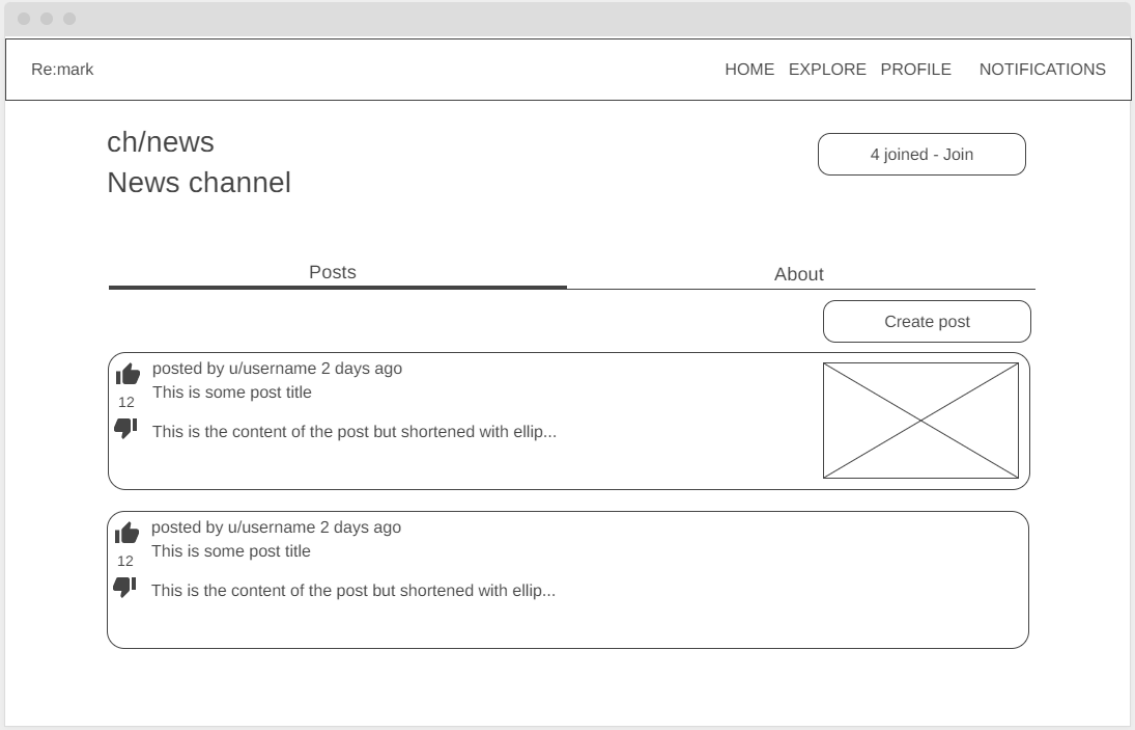
A csatorna követésére gomb (eddigi követők számával) az oldal tetején lévő fejlécben kerül elhelyezésre az egyedi név és a cím mellett.

A csatorna posztjait és a leírását két külön tabon tesszük elérhetővé, amelyek között a csatorna megnevezése és a követési gomb alatti tab navigátor gombokkal válthatunk.

Így könnyedén eloszthető két részre az információfolyam: egyik részén csak a posztok vannak egy végtelenségig görgethető listában; másik részén a csatorna

alapadatai. Nincs szükség így felugró ablakokra, és az alapadatok sem veszik el a figyelmet a posztokról, ugyanis alapvetően a posztok listája kerül megnyitásra.

A posztok tabján a csatornákhoz hasonlóan kártyákon vannak az ízelítő információk feltüntetve (4. ábra).

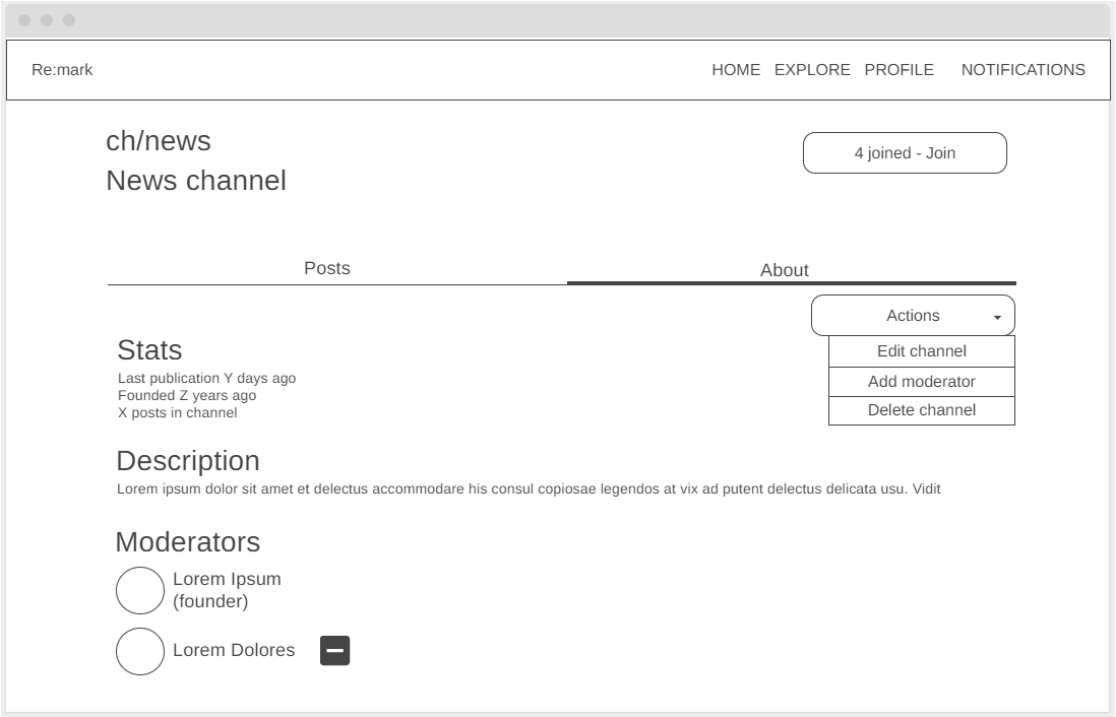


4. ábra. Posztok listázási tabjának nézete

2.3.5 Csatorna részleteinek nézete

A csatorna oldalán az „About” tab gombjára kattintva kerülhetünk a részleteket ismertető nézetre. Tartalmaz statisztikát, leírást, moderátorok listáját és lenyíló menüben akciógombokat a tulajdonos számára, hogy módosítsa, törölhesse a csatornát és további moderátorokat adhasson hozzá.

A moderátorok listájában egy-egy listaelem mellett az adott moderátor kirúgására is gombokat láthatunk (5. ábra). Így a tulajdonosi akciók két területre fokuszálódnak csak: a lenyíló menüre és a moderátorok melletti gombokra.



5. ábra. Csatorna részleteit mutató tab nézete

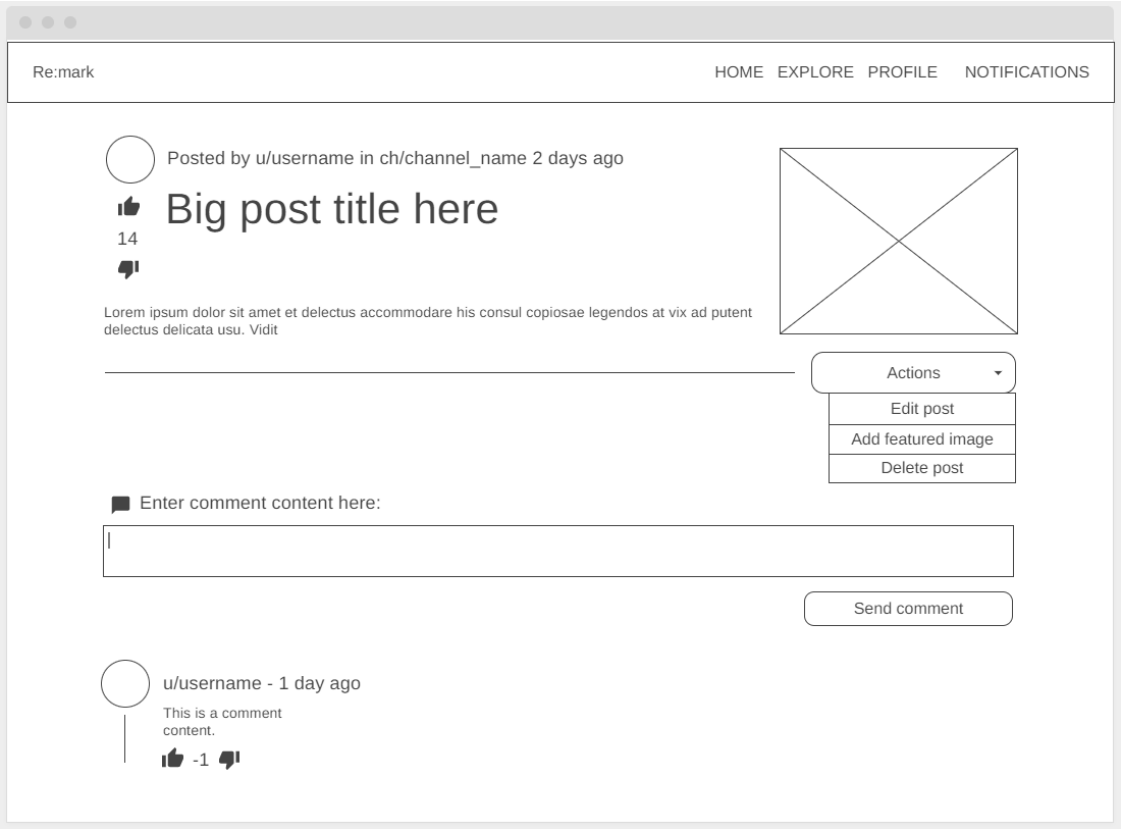
2.3.6 Poszt nézete

A poszt nézete több részből épül fel. Az első szekcióban helyet kap a posztolóról és a posztolás körülményeiről (mikor, milyen csatornán) egy fejléc. Ezek alatt jelenik meg a poszt címe és a poszthoz rendelt kép, ha van ilyen. Az szavazáshoz szükséges akciógombok ennek a szekciónak a bal oldalán jelennek meg. A poszt tartalma mindezek alatt jelenik meg.

Ezt a részt elválasztja egy vonal, amelynek jobb végében a poszt létrehozója számára jelennek meg lenyíló menüben a poszt szerkesztésére, törlésére és kép hozzáadására akciógombok.

Az elválasztóvonal biztosítja a kommentek és ahhoz tartozó komponensek elkülönítését a poszt saját komponenseitől. A vonal alatt jelenik meg egy újabb szekcióban a komment létrehozásához egy szövegbeviteli terület és a küldés gomb. Ez biztosítja, hogy a poszthoz azonnali reagálási lehetősége legyen a felhasználónak, központi területen helyezkedik el, nem kell keresni. Végül az utolsó szekcióban mindezek alatt a már létrehozott kommentek listája látható. Minden komment rendelkezik közzétételi fejléccel (ki tette közzé, mikor) és a hozzászólás tartalmával, a tartalom alatt szavazati akciógombokkal (6. ábra). Egy-egy komment bal oldalán található egy

függőleges vonal, hogy könnyen lekövethető legyen egy komment tartalmának a folyása, ha átugraná a felhasználó teljes egészében.



6. ábra. Poszt oldal nézete

3 Technológiák ismertetése

A fejlesztés során törekedtem olyan technológiákat alkalmazni, amelyek a szakdolgozat írásakor a piacon széleskörben használatosak, illetve magas fokú támogatottsággal rendelkeznek. Érdemesnek véltem JavaScript alapú technológiákat használni mind kliens, mind szerver oldalon, hogy csökkenjen a nyelvek különbözősége miatti kontextusváltások szükségessége.

3.1 Kliens oldali technológiák

A Re:mark kliens oldala weboldalon keresztül érhető el. Egy weboldal elkészítéséhez HTML, CSS és JavaScript technológiák bevetése szükséges. Azonban komplex követelményekkel rendelkező és összetett felépítésű weboldalak esetén érdemes lehet JavaScript keretrendszert vagy könyvtárat használni. A Re:mark esetében is egy JavaScript könyvtárral építettem a weboldalt, a Reacttel.

3.1.1 Trendek

A piacon találhatóak közül még két másik hasonlóan releváns JavaScript keretrendszert/könyvtárat vizsgálhatunk meg: az Angulart, és a Vue.js-t [6]. Mivel a három közül a legtöbb tapasztalatom Reacttel volt, így arra esett a döntésem.

Az Angular egy sok szempontból teljes JavaScript keretrendszer, ugyanis célja megkönnyebbíteni vállalati szintű webalkalmazások készítését jól előkészített alapokkal. Kötelezővé teszi a TypeScript nyelv használatát, tartalmaz űrlap validációt, navigációs rendszert, az állapotkezelést, szolgáltat függőség injektálást és build eszközöket is. 2016 előtti verzióját *AngularJS*-nek hívták, azt a verziót azonban teljesen újraírták, mivel a kódbázisa nem volt fenntartható. Kódbázisának fő karbantartója a *Google*.

A Vue.js és a React egyszerűbbek, de nagyon rugalmas és könnyen használható könyvtárak. Nagyobb szabadságot adnak a fejlesztőnek, mint az Angular. *Virtuális DOM*-ban (Document Object Model) dolgoznak, ami lehetővé teszi, hogy változtatásokat tegyünk a DOM fájában anélkül, hogy az egészet kéne újrarenderelni. Emiatt gyorsabb a működésük.

Mindkettőnél a komponensek egymásba ágyazhatóak, a weboldal felépítése a komponensek egymásba ágyazásával jön létre. Komponens alapú felépítésük miatt könnyen bővíthetők és újra felhasználhatóak lesznek felhasználói felületünk egyes részei.

A React és a Vue közötti legfőbb különbség az adatkötés iránya. Vue esetében a komponens és a szülője között kétirányú adatkötés lehetséges, míg Reactben csak a szülő adhat adatot a gyermeknek.

3.1.2 React

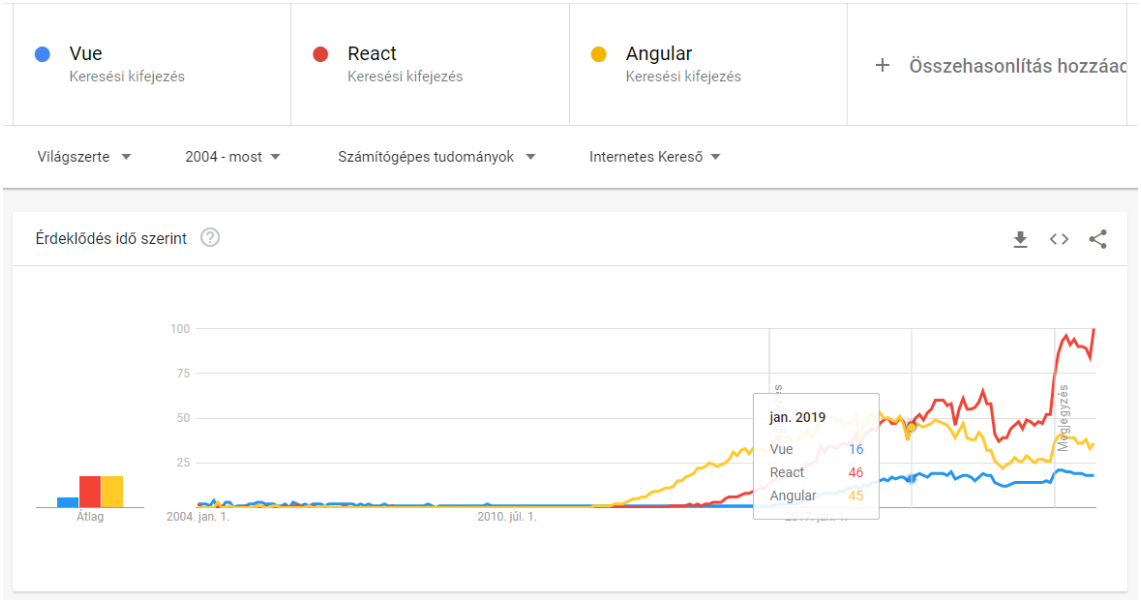
A React komponensei *JSX* (JavaScript XML) nyelven íródnak, amely egy JavaScript nyelv kiterjesztése. Segítségével a React komponensek maguk HTML-lel íródnak, amelyet majd a React motorja alakít át JavaScript kóddá (pl.: `createElement()` és `appendChild()` hívásokká).

A komponensek közötti egyirányú adatkötés a komponensek *props* objektum alatti attribútumaival történik. A szülő felé csupán akciókon keresztül, azaz például egy eseménykezelő függvényen keresztül tud adatot küldeni a gyerekkomponens. Ezt a mechanizmust könnyen a következő kifejezéssel lehet összefoglalni: „*properties flow down, actions flow up”.*

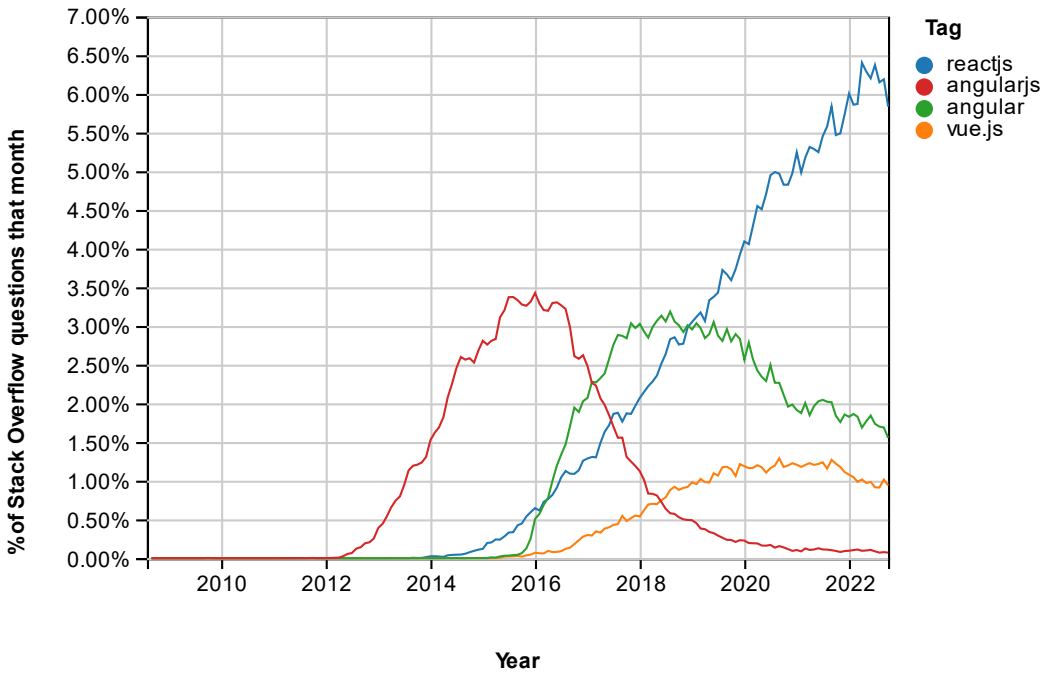
Create React App segítségével készítettem elő *SPA* (Single-Page Application) weboldalam. Az SPA dinamikusan frissülő tartalmú weboldal. Első betöltésekor minden letöltésre kerül, hogy további interakciókra JavaScript kód segítségével frissüljön a megjelenő tartalom, így a szerverrel nem kell újrarendereltetni weboldalunkat. Tekintettel arra, hogy a React önmagában nem szolgáltat megoldást minden webfejlesztés során előforduló problémakörre, így segítségül hívtam jó néhány könyvtárat:

- *React Router*: segítségével könnyen lehet a weboldalakat útvonalakra bontani
- *React Hook Form:* űrlapok építését és validációját könnyítette meg
- *React Query*: adatok lekérdezését és manipulálását segítette a szerverrel való kommunikáció során.

A Google Trends szolgáltatás által vizsgált Google kereséseket összehasonlító (7. ábra), illetve a Stack Overflow Trends szolgáltatás által vizsgált kereséseket [7] összehasonlító (8. ábra) grafikonok leolvasásával egyértelműen megállapítható, hogy a 2019-es évtől kezdve a React vette át a stafétát az Angulartól mint a legnépszerűbb hármuk közül, és azóta is tartja ezt a tendenciát.



7. ábra. Google Trends – Google keresések összehasonlítása



8. ábra. Stack Overflow Trends – Kérdések összehasonlítása adott címkék alapján

3.1.3 Chakra UI

A Chakra UI [8] egy React UI (User Interface, azaz felhasználói felület) komponens könyvtár. Felajánl alapvető előre felkészített komponenseket, amelyekkel felgyorsulhat a fejlesztés és még az esztétikai minősége is biztosított weboldalunknak általuk. A TailwindCSS színpalettáját és alapvető stílusvilágát követi. A piacon található

UI könyvtárak közül ez a "legcsupaszabb", azaz nem kínál túl sok beépített komponenst, viszont azokhoz, amiket kínál, széles körben biztosít személyre szabási lehetőségeket.

Támogatja nappali és éjszakai módú stílusok definiálását, a téma személyre szabását központilag, és a *CSS-in-JS* megközelítést. A Chakra UI mellett nagy népszerűségnek örvendenek a Material UI, a Bootstrap, és az Ant Design könyvtárak is.

3.2 Szerver oldali technológiák

A többrétegű adatvezérelt rendszerekben az üzleti logika szerver oldalon kerül megvalósításra. A szerver oldali technológiák közül a legnépszerűbb futtatási környezetek a Java EE, a PHP, a Python, a .NET és a Node.js. Ezek mindegyikéhez számos keretrendszer tartozik, a legismertebbek a Spring, a Laravel, a Django, az ASP.NET és az Express. [9] A keretrendszerek segítségével gyorsabban és egyszerűbben lehet fejleszteni, és a kód átláthatósága is javul, ugyanis jól bevált tervezési mintákat és gyakorlatokat lehet velük alkalmazni.

3.2.1 Node.js

A Node.js egy JavaScript futtatási környezet, amelyet a Google Chrome V8 JavaScript motorja hajt. A Node.js egyszálas, ámde aszinkron, eseményvezérelt nem-blokkoló I/O modell alapján működik. [10]

Nem-blokkoló I/O modelljének természetéből adódóan alkalmas egyszerre több ezer bejövő egyszerű kérés kezelésére. Ezzel szemben egyetlen hosszú, számításigényes utasítássorozat végrehajtására nem tűnik éppen ideális választásnak. Éppen ezért kis terhelésre, webes applikációk API-jának (Application Programming Interface), videó streamingre és gyors adatátalakításra használják. Rengeteg beépített modullal rendelkezik különféle hálózati kommunikációs protokollok megvalósítására.

Az Azure Function App platformja támogatja a legtöbb modern futtatási környezet használatát a függvények felprogramozására, ezek közé tartozik a Node.js is. Node.js mellett való döntésemben az abban szerzett mély tudásom és még az erősített meg, hogy az Azure szolgáltatások Node.js környezettel való használatához igényes és kimerítő dokumentáció áll rendelkezésre, valamint egészen kiterjedt szoftveres közössége van. A Function Appok programozásakor nem vettem igénybe Node.js keretrendszereket, ugyanis az Azure Node.js SDK-ja (Software Development Kit) bőven elég volt a szükséges funkciók megvalósításához.

3.3 Közös technológiák

3.3.1 TypeScript

A Node.js és a böngésző programozási nyelve a JavaScript, azonban ez a nyelv önmagában sok buktatót és kevés szintaktikai segítséget nyújt, hiszen dinamikusan és gyengén típusos. A TypeScript [11] nyelv a JavaScript nyelv superset-je, vagyis a JavaScript nyelv egy kiterjesztése, amelyet a Microsoft fejlesztett. A TypeScript nyelv egy statikusan típusos, objektumorientált nyelv, amelyet a JavaScript nyelvre fordíthatunk. Legnagyobb előnye, hogy javít a kód átláthatóságán, lehetőséget ad kódkiegészítés alkalmazására és a hibakereshetősége is javul.

3.3.2 NPM

A Node.js egyik csomagkezelője az NPM (Node Package Manager) [12], amellyel kényelmesen listázhatjuk, telepíthetjük, frissíthetjük és eltávolíthatjuk projektünkben használt JavaScript moduljainkat, valamint mi magunk is publikálhatjuk az általunk fejlesztett modulokat az NPM rendszerébe. Több JavaScript projektből álló rendszerekben egy jó módszer NPM csomagokba rendezni az újra felhasznált kódrészleteket.

3.4 Kommunikációs interfész: REST

A REST (Representational State Transfer) [13] egy szoftverarchitekturális módszer, amely nem definiál konkrét protokollt, csupán iránymutató elveket definiál, amelyek betartásával RESTful, azaz REST kényszereinek megfelelő lesz a kommunikációnk alrendszereink között. A REST architektúra 6 elve közül az egyik legfontosabb elvárás, hogy a kliens és a szerver közötti kommunikáció során a kliensnek nem kell tudnia a szerver oldali állapotról (állapotmentes a kommunikáció), a szervernek nem kell tárolnia a kliens állapotát.

A legelterjedt módszer a hálózati kommunikáció alapjául a HTTP protokollt használni REST megszorításokkal, valamint a JSON (JavaScript Object Notation) formátumot használni az adatok átviteli formájául. Azure Function Apps platformján Node.js alatt könnyű RESTful API-t tervezni és fejleszteni.

3.5 Felhasznált felhőszolgáltatások

3.5.1 Kódfuttatási platform

Az üzleti logikánk futtatása, a kérések feldolgozását végző szerverünk többféle felhőszolgáltatási típusban valósulhat meg. Akár egy saját virtuális gépet is szerezhetünk a felhőben, de ha kevesebb üzemeltetési felelősségre vágyunk, akkor akár *PaaS* (Platform-as-a-Service) szolgáltatásokra is korlátozódhatunk. Ekkor megszabadulunk az operációs rendszer, hálózati konfigurációs és védelem, valamint a szoftverfrissítések felelősségétől.

A PaaS szolgáltatások közül a legnépszerűbbek a kódfuttatási platformok, amelyek a kód futtatására szolgálnak, és a kód futtatásához szükséges erőforrásokat biztosítják. Ezek közül a legnépszerűbbek az Azure világában az App Service és a Function App [14]. A Function App még szűkebben értelmezve egy *FaaS* (Function-as-a-Service) szolgáltatás, amely függvényként viselkedik, kódot futtat külső események (*triggerek*) hatására. Teljesen *serverless*, ami annyit takar, hogy nem kell szervereket menedzselnünk, nincs folytonosan a háttérben futó szerverpéldányunk, magától példányosodik, skálázódik.

Az Azure Function App lehetővé teszi a kód futtatásához szükséges erőforrások automatikus skálázását, és hogy roppant egyszerűen tudjunk kódot futtatni a felhőben anélkül. Fő hátránya a *cold start*, azaz a hideg indítás, amikor a háttérben futó példányosított szervernek először kell futtatnia a kódot, ami sokáig eltarthat tekintettel arra, mennyi erőforrást kell frissen előkészítenie. A későbbi futtatások már gyorsabbak, de az első futtatásokat mindenképpen figyelembe kell venni, ha a teljesítményt vizsgáljuk.

3.5.2 Adatbázis

Adatvezérelt rendszerek központi eleme az adatbázis és az arra használt adatbázis-kezelő rendszer, DBMS (Database Management System). Adatmodellek legelterjedtebb típusai a relációs, objektumorientált, hálós és szemi-strukturált (avagy dokumentum-orientált) modellek. Ma elterjedt kifejezés a "NoSQL adatbázis", amely nem-relációs adatbázisokat jelöl. Gyakran a relációs adatbázisokra SQL adatbázisokként referálunk, ugyanis relációs DBMS-eknek a megszokott nyelve az SQL.

A NoSQL adatbázisokat azonban nem lehet egyszerűen a relációs adatbázisokkal összehasonlítani, mivel a NoSQL adatbázisok sokkal rugalmasabbak, sémamentesek, és sokkal több funkciót támogatnak. Szimpla API-t biztosítanak az adatmanipulálásra és a lekérdezésekre. Fő céljuk a nagy mennyiségű adatok gyors és hatékony tárolása és kezelése, valamint elosztott működésben való alkalmazhatóság.

DBMS-ünket akár a felhőszolgáltatók is biztosíthatják, de saját adatbázis-kezelő rendszerünket is használhatjuk. Egy az Azure által menedzselt globális elosztottságot is támogató NoSQL (MongoDB alapú) dokumentum-orientált adatbázis-kezelő rendszerrel a Cosmos DB [15] is. Könnyen skálázható, automatikus indexelést képes alkalmazni, többféle API nyelvet támogat (pl.: SQL, MongoDB, Cassandra), és magas rendelkezésre állást.

3.5.3 Képtárolás

Serverless környezetben felmerülhet a probléma, hogyha nincs szerver, akkor nincs perzisztens tárolásra az alkalmazásunk mellett lehetőség. Egyfajta megoldást jelenthet az, ha képfájljaink bináris formátumban adatbázisunkban tároljuk. Azonban ez sok kényelmetlenséggel és körülménnyel járhat, így érdemes felhőszolgáltató által felajánlott objektum tárolókat használni.

Az Azure Blob Storage szolgáltatás konténerei képesek tárolni bármilyen típusú fájlt – köztük képeket –, és azokat saját API-jával lehet kezelni. A Blob Storage szolgáltatás az Azure Storage szolgáltatás része, a blobok tulajdonképpen binárisan tárolt nagyobb méretű objektumok.

3.5.4 Eseményvezérelt rendszerek

Felhő alapú mikroszolgáltatásos rendszerek gyakran szükségelik az aszinkron üzenetkezelési infrastruktúra létezését [16]. Az aszinkron üzenetkezeléssel képesek vagyunk a rendszerünkben lévő különböző komponenseket egymással laza kapcsolásba hozni, és az egyes komponensek nem kell, hogy egymásra várjanak. Ezzel lehetőséget adunk a skálázhatóság és megbízhatóság növelésére.

Aszinkron üzenetkezelés egy fontos résztvevője az üzenetbróker, amelynek feladata az üzenet ellenőrzése, transzformálása és útvonalválasztás a feldolgozók felé, üzenetsorral akár megbízható tároló is lehet üzeneteknek.

Azure platformon az üzenetkezelés és az ahhoz közel álló eseménykezelés körében használt szolgáltatások közé tartozik az Azure Service Bus, az Azure Event Grid, az Azure Event Hub és az Azure Notification Hub.

3.5.4.1 Azure Service Bus

Az Azure Service Bus [17] egy menedzselt üzenetbróker, képes serverless is működni. Beépített funkcionalitásai közé tartozik a terheléselosztás, üzenetsorral vagy topic-subscription modellben működés. Koncepciója azonos más nem felhő alapú üzenetbrókeréval, ilyen például az Apache ActiveMQ. Mivel Azure platformon működik, így a szolgáltatások közötti integráció is egyszerűbb.

3.5.4.2 Azure SignalR Service

Az Azure SignalR Service egy menedzselt valós idejű webes kommunikációt lehetővé tevő szolgáltatás, tulajdonképpen SignalR alapú szerveralkalmazást imitál. A szolgáltatás a kliensek és a szerver közötti kommunikációt kezeli, és a klienseknek lehetővé teszi, hogy a szerverhez csatlakozva hubokon keresztül valós időben kommunikáljanak egymással. Sokféle kommunikációs protokollt biztosít a kliensek számára, ezek közé tartozik a WebSocket, a Server-Sent Events és a Long Polling.

3.5.5 Statikus webhoszting

Statikus weboldalak tárhelyeként szolgálnak a webhoszting szolgáltatók. Azure alatt egy jó megoldás a Static Web App szolgáltatást igénybe venni ilyen célból, nagyban leegyszerűsíti weboldalak élesítését. A Static Web App platform képes minden új commit esetén is a GitHubról letölteni a kódot, és a kódból generálni a statikus weboldalt. Ingyenes SSL tanúsítványt állít ki, beállíthatunk saját domaint is. A weboldal tartalma globálisan is el van osztva, így weboldalunk gyorsabban tud betöltődni.

3.5.6 Egységes API kezelés

Alkalmazásaink jól definiált API-kon keresztüli nyilvánossá tételével lehet egyszerűen kezelni szolgáltatásaink és adataink elérését. Az Azure API Management (APIM) szolgáltatás lehetővé teszi az API-k együttes és konzisztens kezelését, hozzáférési szabályok beállítását, kártékony cselekvések elleni védelmet, a hibakezelést, a mérést és a monitorozást. Ezen képességeivel biztosítja az APIM szolgáltatás a hatékony munkát, növeli a rendszer kifelé látszódó letisztultságát.

3.6 Terraform

A *Hashicorp* terméke, a Terraform [4] egy saját leírónyelvet használó IaC eszköz. Segítségével biztonságosan és hatékonyan tudunk telepíteni, változtatni és verziózni erőforrásokat felhőbéli, valamint saját helyi platformon is. Platformagnosztikus eszköz, támogatja az Azure platformon való üzemeltetés megoldását kóddal.

A Terraform a kód alapján végzi el a telepítést, állapotfájlokban tárolja az általa menedzselt erőforrásaink. A deklaratív megközelítését követi a konfigurációmenedzsmentnek, könnyedén lehet vele ember által is olvasható és hamar értelmezhető kódot írni.

Egy hatékony munkafolyamat által érhetjük el, hogy erőforrásaink élesítésre kerüljenek a Terraform menedzselésében. Ez a folyamat a „*write-plan-apply*” hármas: deklaráljuk kódból az infrastruktúra elemeit, majd futtatunk egy tervezést, amely kimutatja, milyen változások várhatóak, végül pedig ezt a tervet kiviteleztetjük a Terraform klienssel és onnantól már a Terraform gondoskodik erőforrásaink életciklusáról.

4 Architekturális terv

A felhőszolgáltatók (pl. Amazon Web Services, Microsoft Azure, Google Cloud Platform) biztosítják, hogy a szoftverarchitektúra kivitelezéséhez ne kelljen saját infrastruktúrát üzemeltetni. Ennek megfelelően a különböző mikroszolgáltatásainknak a Microsoft Azure felhőszolgáltatót választottam. A magas rendelkezésre állás, a jó skálázhatóság, illetve a tranziens hibák hatékony kezelésének megoldására pedig a felhőplatform által felajánlott serverless megoldásokat használtam fel.

A magas szintű, azaz architekturális tervezés során törekedtem vállalati szinten is elfogadható, kis granularitású rendszert összeállítani a Re:mark mögé.

4.1 Logikai nézet

Az architekturális tervezés egyik kezdő lépése a terv logikai nézetének kialakítása, azaz logikai (függőségi) kapcsolatok kialakítása a teljes rendszer moduljai között. [16]

A készülő szoftverrendszert a következő modulcsoportokra osztottam felelősségi köreik szerint:

- *User services*: felhasználókhoz kapcsolódó szolgáltatások,
- *Content services*: tartalomhoz kapcsolódó szolgáltatások,
- *Blob services*: fájlokhoz kapcsolódó szolgáltatások,
- *Notification services*: értesítésekhez kapcsolódó szolgáltatások,
- Kliens,
- Adatbázis.

Ezek a 9. ábra bemutatásában is láthatóak, amelyben tisztázásra került az is, hogy az egyes modulok közötti kapcsolatok milyen irányúak és minőségűek, azaz melyik melyiktől milyen miként függ.

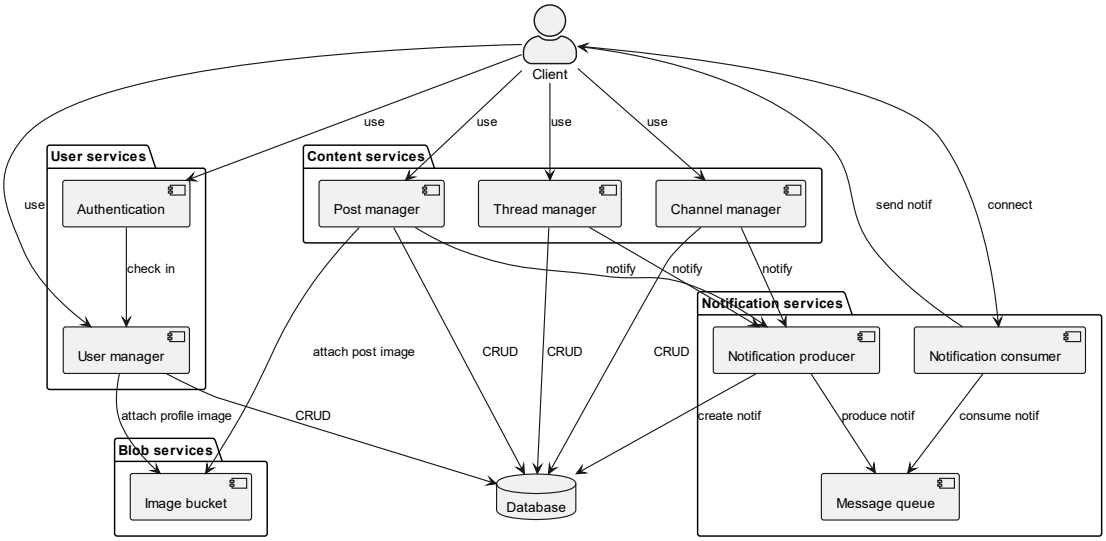
A felhasználókat és tartalmakat menedzselő modulok az adatbázisban lévő entitásokon CRUD műveleteket (Create-Read-Update-Delete), azaz létrehozást, olvasást, módosítást és törlést tudnak végrehajtani.

A tartalomkezelő modulok valamilyen módon értesítik az üzenettermelő modult, amely generál egy üzenetet, leküldi az üzenetsorba, ahonnan a fogyasztó pedig ezt

kiveheti, és a klienssel való oda-vissza kapcsolaton keresztül továbbítja az üzenetet értesítés formájában.

A logikai nézetben jelzésszerűen benne van, hogy valamilyen autentikációs modul is használatra kerül a rendszerben, amely a kliens beazonosítására szolgál a kérések során.

A posztokat és felhasználókat menedzselő modulok képek tárolására egy objektumtárat használ.



9. ábra. Rendszerarchitektúra logikai nézete UML diagrammal ábrázolva

4.2 Fizikai nézet

Az architekturális tervezés végső lépése a fizikai nézet kialakítása. A fizikai nézet feladata a logikai nézetbeli modulok konkrét mikroszolgáltatásokra bontása és azok platformspecifikus ábrázolása, fizikai kapcsolatok bemutatása. A mi esetünkben a mikroszolgáltatások konkrét felhőbéli szolgáltatásokkal kerülnek megfeleltetésre. A 10. ábra is ezt szemlélteti, illetve a szolgáltatások egymásra rétegződését, szintezettségét is jól ábrázolja.

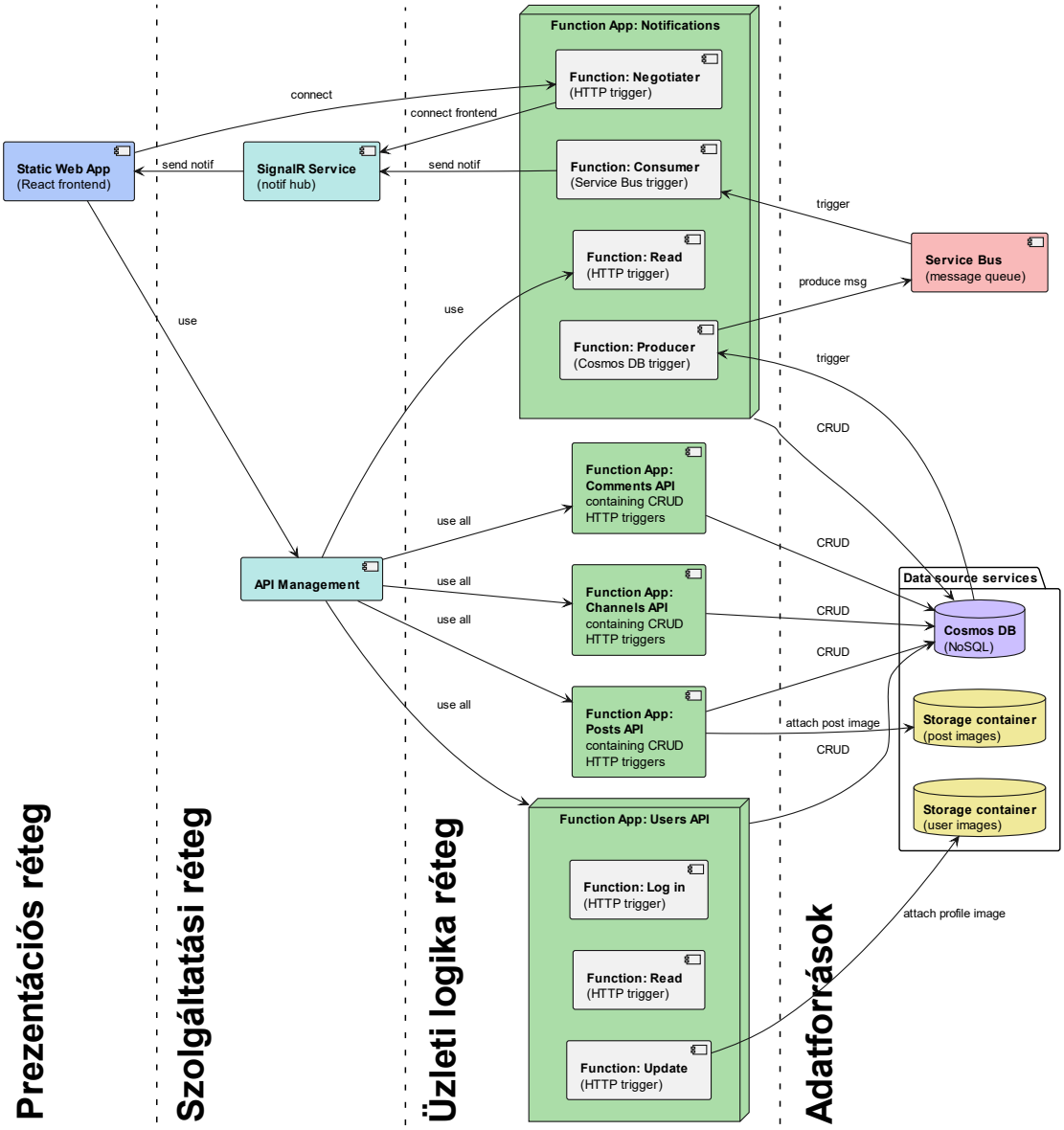
A weboldal hosztolására Static Web App került alkalmazásra, mivel roppant könnyűvé válik vele az üzemeltetés és a folyamatos integráció. A kliens felé felkínált nyilvános API-k összegyűjtésére és hatékony együttes kezelésére egy serverless üzemmódú API Managementet példányosítottam.

Tekintettel arra, milyen könnyű élesíteni és mennyire sokféle triggerfajtát (Service Bus trigger, Cosmos DB trigger, HTTP trigger, timer trigger stb.) képes

biztosítani, ezen indokok mentén az üzleti logika futtatási környezetének Function Appokat választottam.

A valós idejű értesítéskezelésre kliens oldalról jól ismert módszer WebSocketes megoldás bevetése, éppen ezért a SignalR Service-t választottam, azt is serverless üzemmódban példányosítottam. A termelő és fogyasztó oldali üzenetkezelő komponensek közötti üzenetáramlásért a Service Bus felel, mely üzenetbróker és üzenetsor is egyben, egyértelmű választás volt, hiszen könnyedén integrálható Azure Function App függvényekkel.

A webes tartalmak entitásainak tárolására a Cosmos DB-t választottam SQL API nyelvvel, amely habár NoSQL DBMS tulajdonságából fakadóan sémát nem képes tartani, viszont globális tartalomdisztribúciót, cache-elést, automatikus indexelést és skálázódást biztosít. Ezek egy potenciálisan globális méreteket öltő, vállalati fórumalkalmazás készítésénél mind erősebb tulajdonságoknak bizonyulhatnak. Képek adatbázisban történő tárolása nem a legjobb megoldás, mivel a képek nagy méretű bináris fájlok, így a Blob Storage volt erre a legjobb választás.



10. ábra. Rendszerarchitektúra fizikai nézete UML diagrammal ábrázolva

4.3 Adatbázisséma

A Cosmos DB sok jó tulajdonsága ellenére NoSQL létére sémát nem képes tartani, a sémakényszereket nekünk kell kódból megoldani. A perzisztensen tárolandó entitások a követelményekből könnyen kitalálhatóak. Az entitások közé tartozik a csatorna (*Channel*), poszt (*Post*), felhasználó (*User*), komment (*Comment*) és az értesítés (*Notification*). Az entitások kapcsolatait a 11. ábra jól bemutatja.

A csatornakövetést a felhasználók és a csatornák közötti kapcsolatra ékelődő *ChannelJoin* nevű entitással valósítottam meg. A csatorna tulajdonosának és a moderátoroknak nem volt szükséges külön az asszociációt megoldani, ugyanis ezen felhasználók egyben követői is lesznek a csatornának, így érdemes volt ezen

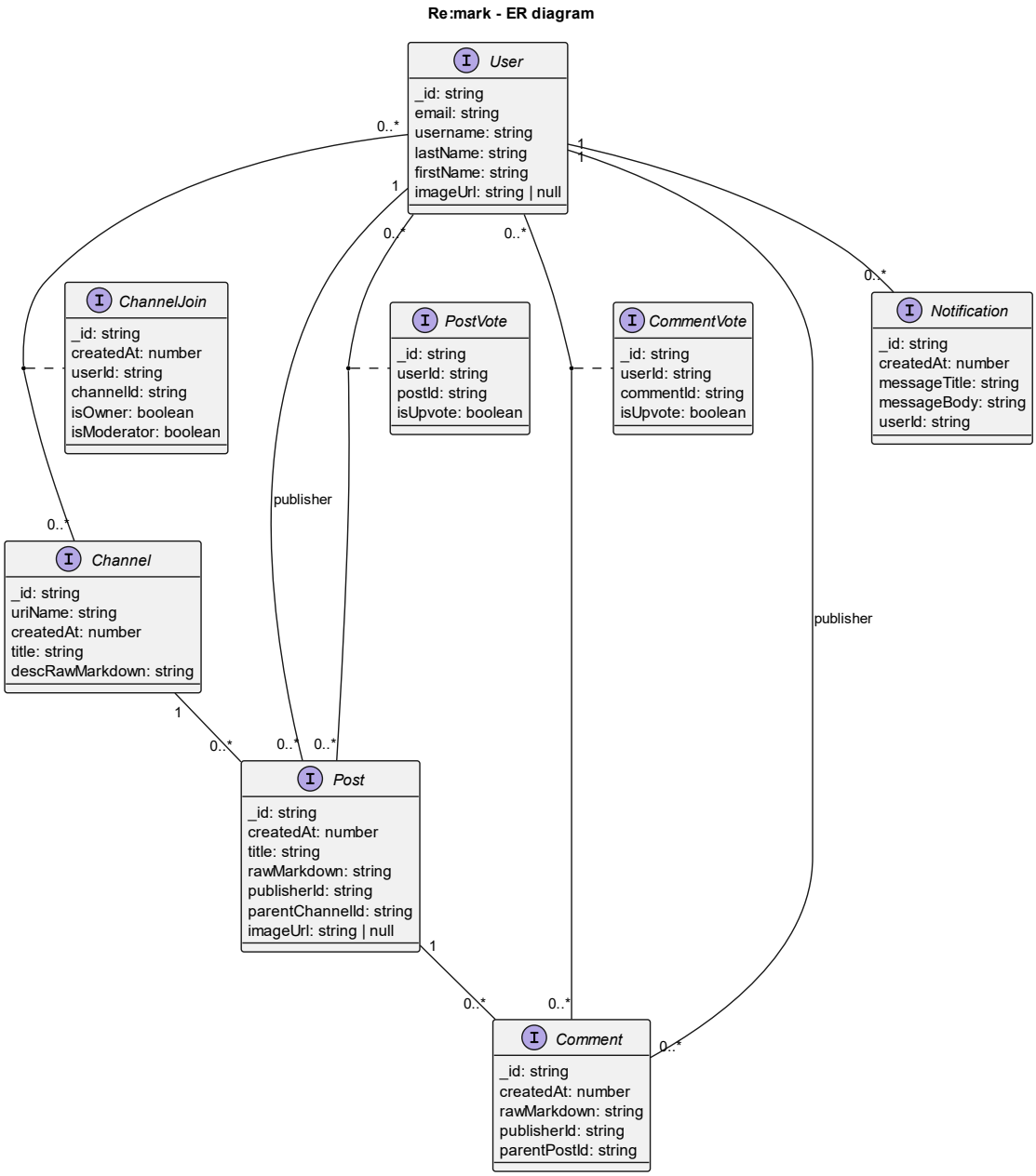
tulajdonságokat is `boolean` típusú mezőkként (`isModerator`, `isOwner`) tárolnom a felhasználókról a *ChannelJoin* entitásokban.

Poszt és csatorna között tartalmazási kapcsolat van, több-az-egyhez kapcsolat a *Post* entitás `parentChannelId` mezője által biztosított. Hasonlóan komment és poszt között is ez a helyzet, a több-az-egyhez kapcsolat a *Comment* entitás `parentPostId` mezője által biztosított.

Értesítéseket felhasználóhoz kötjük. Értesítések több-az-egyhez kapcsolata a felhasználóval a *Notification* `userId` mezője által jön létre. Posztok és kommentek esetén fontos tudnunk, ki hozta létre a tartalmat, így ezek esetén is a több-az-egyhez kapcsolatot az egyes entitások `publisherId` mezője biztosítja.

A szavazatok felhasználó és komment, illetve felhasználó és poszt között több-a-többhöz kapcsolatokban formalizálhatóak, ennek megoldására a kapcsolatokra ékelődő *PostVote* és *CommentVote* entitásokat használtam.

A `createdAt` mezők epoch időbélyeget jelölnek egész számként, így megakadályozzák, ha esetleg más vagy több szerver oldali technológiát is bevetünk a jövőben, ne okozhasson az új rendszerbe való konvertálás problémákat a dátumkezelésnél. Az `_id` mezők mind a Cosmos DB által generáltak, így nekünk nem kell ezt a mezőt menedzselni.



11. ábra. Az adatbázisséma ER diagrammal ábrázolva

5 Az alkalmazás fejlesztése

A szakdolgozathoz készült projektkódok mind Git verziókezelő rendszer segítségével tárolom, egy repository alatt, azaz monorepo stratégiát alkalmazva. Git szolgáltatóként a GitHubot választottam, ezen a platformon nyilvánosan elérhető a teljes kódbázis. Az itt található alprojektek a következők: a React kliensalkalmazás, az öt Azure Function App, két fejlesztést segítő NPM csomag és a Terraform projekt.

A Function Appok mind Node.js alatt futó, TypeScriptben írt, kis méretű függvényeket gyűjtenek egy-egy NPM projekt alatt. A függvények főként HTTP triggerek, de alkalmazásra kerültek Service Bus és Cosmos DB triggerek is.

5.1 Saját fejlesztésű NPM csomagok

A Re:markhoz köthető alprojektek – a Terraform projekten kívül – mind JavaScript platformon készült projektek. Két NPM csomagot fejlesztettem, amelyeket a projektek közösen tudnak használni. A cél a kódduplikáció elkerülése volt.

Az egyik ilyen csomag a @triszt4n/remark-auth. Célja közös autentikációs, azaz felhasználóhitelesítési logikát biztosítani a szerver oldali Function App függvények részére. Az API-ja egyszerű, két függvényt biztosít: a `createJWT` függvényt, amellyel Re:markos saját privát kulccsal aláírt JWT (JSON Web Token) [18] készíthető, illetve a `readUserFromAuthHeader` függvény, amellyel a HTTP kérés fejlécében lévő saját JWT-nkből kideríthetjük a kérés intézőjét. Csupán egy függősége van JWT aláírás miatt: a jsonwebtoken.

```
// User data from Authorization
const result = readUserFromAuthHeader(req, process.env.JWT_PRIVATE_KEY)
const user =
  result.isError ? null :
    (result.userFromJwt as { id: string; username: string; email: string })
```

A példakódban láthatjuk, hogy kerül kiolvasásra egy Function App HTTP trigger függvényben a bejövő kérésből a kérés intézője, ha van egyáltalán.

A JWT egy nyílt standard, amely kompakt információ szállítását teszi lehetővé JSON objektumokként. Digitálisan aláírt, így ellenőrizhető és megbízható.

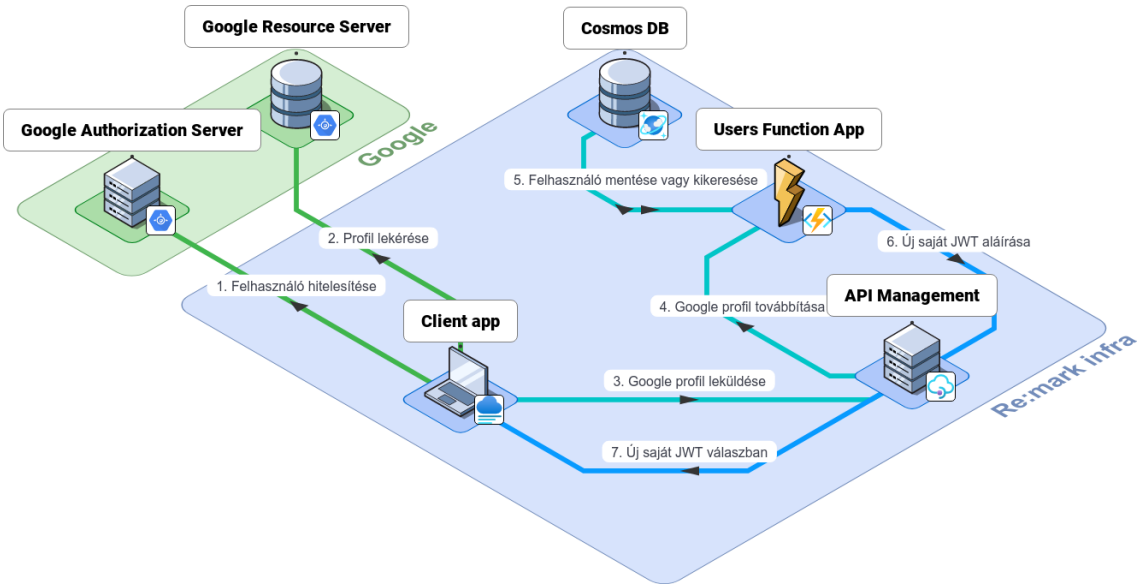
A másik csomag a @triszt4n/remark-types. Ez a csomag a projekteken átívelő TypeScript típusokat tömöríti és ajánlja ki. Ezen típusok közé tartoznak az

adatbázissémához tartozó *modell interfészek*, illetve a szerver és a kliens között utazó *DTO*-kat (Data Transfer Object) jellemző interfészek. Ezt a csomagot mind a Function Appok és a React applikáció beemeli.

5.2 Rendszer szintű folyamatok implementációi

5.2.1 Felhasználóhitelesítés

A felhasználók – követelménynek megfelelően – Google fiókjukkal tudnak bejelentkezni a weboldalra. A 12. ábra a folyamat lépéseit illusztrálja.



12. ábra. A bejelentkezési folyamat rendszerszinten Isoflow-ban ábrázolva

A Google-lel való hitelesítés után a profilt leküldjük a **POST /login** végpontra, a kérést a megfelelő Azure Function App feldolgozza. Feldolgozás során a kapott profilból kifejti az e-mail címet, lekérdezi az adatbázisból ezzel az e-mail címmel a felhasználót, ha nincs eredmény, készít felhasználó egyedet és az adatbázisba lementi. A kapott felhasználói egyedet (a Re:mark profilt) pedig visszaküldi a kliensnek a Function App függvény.

Mind a kapott Google profil és a visszaküldött Re:mark profil tulajdonságai aláírt JWT-ként kerülnek szállításra. A hagyományos munkamenet alapú felhasználókezeléssel szemben a JWT alapú annyival előnyösebb, hogy nem akadályozza a horizontális skálázódást.

5.2.1.1 Kliens oldali megvalósítás

A gond nélküli Google OAuth 2.0-s bejelentkeztetést a `GoogleLogin` React komponens oldja meg, amelyet a @react-oauth/google [19] NPM csomag biztosít. Az alábbi roppant egyszerű kódrészlet szemlélteti használatát:

```
<GoogleLogin
  onSuccess={onLoginSuccess} onError={onLoginFailure} useOneTap
/>
```

A komponens megoldja a felhasználó igazoltatását és megszerzi a felhasználótól a szükséges engedélyeket alapvető adatainak (név, e-mail cím) megszerzéséhez. Sikeres autentikálás végén az `onSuccess` propjában kapott függvényt hívja meg.

```
const mutation = useMutation(userModule.loginUser, {
  onSuccess: async ({ data }) => {
    const { jwt, user } = data
    Cookies.set(CookieKeys.REMARK_JWT_TOKEN, jwt, { expires: 2 })
    await queryClient.invalidateQueries('currentUser', {
      refetchInactive: true
    })
    setIsLoggedIn(true)
    navigate('/profile')
  },
  onError: (error) => { /* ... */ }
})

const onLoginSuccess = (response: CredentialResponse) => {
  const { credential } = response
  mutation.mutate(credential)
}
```

Az `onLoginSuccess` függvény csupán annyit tesz, hogy kiveszi az aláírt JWT-t a válaszból. Ez a JWT itt stringként a credential változóban kerül átpasszolásra a mutációs függvénynek. A mutációs `userModule.loginUser` függvényt a *React Query* menedzseli, a kapott JWT szerver oldalra való leküldése után az általunk generált új JWT sütiben letárolásra kerül, és innentől a lejáratáig minden HTTP kérésnek az *Authorization* fejlécében lesz továbbítva szerver oldal kérések intézése esetén.

Kijelentkezéskor törlődik a JWT tokent tároló süti, így már nem tud a kérésekkel együtt utazni.

5.2.1.2 Szerver oldali megvalósítás

Szerver oldalon a beérkezett Google JWT-t dekódolni kell, a kapott e-mail cím alapján lehet megszerezni, vagy éppen készíteni egy új felhasználót az adatbázisban. Az új adatbázisbeli felhasználó adatait JWT-be írjuk 2 napnyi lejárati idővel, majd a tokent

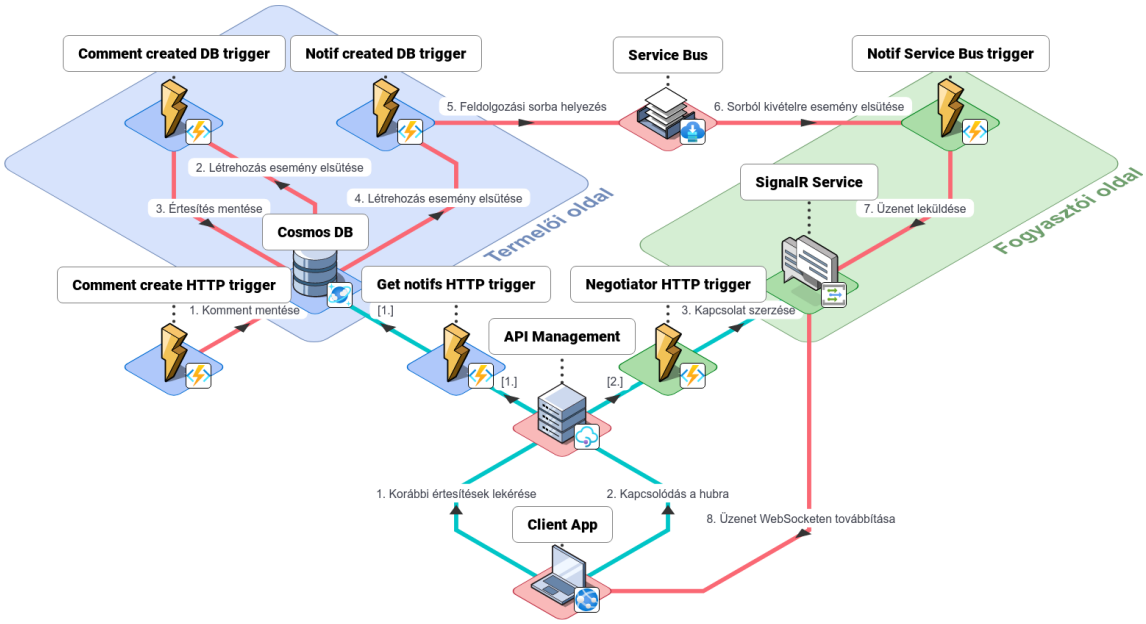
a HTTP válaszban visszaadjuk. A jsonwebtoken [20] NPM csomag került használatba a dekódolására.

```
import { createJWT } from '@triszt4n/remark-auth'
import * as jwt from 'jsonwebtoken'

const httpTrigger: AzureFunction = async function (context: Context, req:
HttpRequest): Promise<void> {
  /* ... */
  const { credential } = req.body
  const googleUser = jwt.decode(credential) as GoogleUser
  const usersContainer = fetchCosmosContainer('Users')
  const dbUser = await getOrCreateUserByEmail(usersContainer, googleUser)
  const jwtToken =
    createJWT(dbUser, process.env.JWT_PRIVATE_KEY, 60 * 60 * 24 * 2)
  context.res = { body: { jwt: jwtToken, user: dbUser } }
}
```

5.2.2 Értesítések kiküldési folyamata

Ahogy azt a 13. ábra is szemlélteti, az értesítésküldés folyamata a termelő-fogyasztó mintát alkalmazza. Termelői oldalon a folyamat ott kezdődik el, amikor egy HTTP trigger beleír az adatbázisba, az új dokumentum az adatbázisban pedig elsüt egy Cosmos DB triggert.



13. ábra. Az értesítésküldési folyamat rendszerszinten Isoflow-ban ábrázolva

Ez a trigger függvény feldolgozza az újonnan készült egyedet, készít értesítéseket a megfelelő felhasználóknak célozva, majd a dokumentumokat lementi az adatbázisba. Ezek után egy másik Cosmos DB trigger ismét elsütésre kerül, ez viszont már a központi

csak értesítés dokumentumokkal foglalkozó függvény, amely a létrejött értesítés példányt szimplán továbbítja a Service Bus üzenetsorába.

Az üzenetsorba helyezés egy újabb, Service Bus trigger típusú függvényt fog élesíteni, amely kiveszi a sorból az üzenetet és azt lepasszolja a SignalR szolgáltatásnak.

Az egész folyamat végbeérésének feltétele, hogy a kliens már fel legyen csatlakozva a hubra és aktívan várja WebSocket kapcsolatán a bejövő értesítéseket a SignalR példánytól. Ehhez kell a kliens és a SignalR közé egy negotiate (tárgyaló) szervervégpont, amely a kapcsolat megkötéséhez a közeget biztosítja. Ezt egy HTTP trigger biztosítja.

Ha a WebSocket kapcsolat már él, a SignalR Service a felé Service Bus triggerből továbbított üzeneteket tudja továbbítani a megfelelő klienseknek.

A kliens böngészőjében a weboldal első betöltésekor (bejelentkezés után) az összes korábbi nem törölt értesítés is letöltésre kerül egy HTTP triggeren keresztül.

5.2.2.1 Termelői oldali megvalósítások

Vizsgáljuk meg a kommentek létrehozásakor elsütött trigger függvény kódját:

```
const cosmosDBTrigger: AzureFunction = async (
  context: Context, documents: ModifiedCommentResource[]
) => {
  // initialization ...

  await Promise.all(
    documents
      .filter((comment) => !comment.isDeleted && !comment.isUpdated)
      .map(async (comment) => {
        const { resource: parentPost } = await postsContainer
          .item(comment.parentPostId, comment.parentPostId)
          .read<PostResource>()
        // omitted code ...

        const { resource: commenterUser } = await usersContainer
          .item(comment.publisherId, comment.publisherId)
          .read<UserResource>()

        await notificationsContainer.items.create<NotificationModel>({
          createdAt: +new Date(),
          messageBody: `Your post titled
[${parentPost.title}](/posts/${parentPost.id}) received a comment by
[u/${commenterUser.username}](/u/${commenterUser.username}).`,
          messageTitle: 'Someone commented on your post',
          userId: parentPost.publisherId
        })
      })
  )
}
```

A függvény paramétereiben is található `documents` tömb tartalmazza a létrehozott vagy megváltozott dokumentumokat. A függvény számára a TypeScript kódja mellett egy *function.json* fájlban azt is definiálhatjuk, hogy milyen Cosmos DB konténerben való változás esetén fusson le. Az viszont sajnos nem adható meg, milyen műveletre korlátozódjon, ezért szükséges a törölt és frissített dokumentumok kiszűrése (`documents.filter(...)`).

Az értesítések létrejöttére kötött Cosmos DB trigger kódja még egyszerűbb. A függvény felépíti a kapcsolatot az üzenetbrókerrel, és az üzenetsorba egy `ServiceBusSender` példányon hívott metódusokkal továbbítja az üzeneteket.

```
const fetchServiceBus = () => {
  // ...
  const serviceBusClient = new ServiceBusClient(/*connectionString*/)
  const serviceBusSender = serviceBusClient.createSender(/*queueName*/)
  return { serviceBusClient, serviceBusSender }
}

const cosmosDBTrigger: AzureFunction = async (context, documents:
NotificationResource[]) => {
  const messages: ServiceBusMessage[] = documents
    .map((notification) => ({ body: notification }))
  const { serviceBusClient, serviceBusSender } = fetchServiceBus()
  try {
    await serviceBusSender.sendMessages(messages)
    await serviceBusSender.close()
  } finally {
    await serviceBusClient.close()
  }
}
```

5.2.2.2 Fogyasztói oldali megvalósítások

A Service Bus trigger *binding*okkal operál, azaz más felhő erőforrások rákötésével. A függvény *input binding*ja a Service Bus üzenetsora (`connection` és `queueName` definiálja), azaz a bemenetről ez van rákötve, *output binding*ja a SignalR hub (`connectionStringSetting` és `hubName` definiálja), azaz a kimenetére az van kötve. A *function.json* függvénykonfigurációs fájl ennek megfelelően van írva:

```
"bindings": [
  {
    "name": "queueItem",
    "type": "serviceBusTrigger",
    "direction": "in",
    "queueName": "%SERVICE_BUS_QUEUE_NAME%",
    "connection": "SERVICE_BUS_CONNECTION_STRING"
  },
  {
    "type": "signalR",
    "name": "signalRMessages",
```

```
      "hubName": "%SIGNALR_HUB_NAME%",
      "connectionStringSetting": "SIGNALR_CONNECTION_STRING",
      "direction": "out"
    }
  ]
```

A negotiate végpontként működő HTTP trigger függvénykonfigurációja a SignalR példánnyal való kapcsolódáshoz a megszokott `httpTrigger` típusú input bindingon és a `http` típusú output bindingon kívül szükségel egy `signalRConnectionInfo` típusú input bindingot is:

```
"bindings": [
  {
    "type": "signalRConnectionInfo",
    "name": "connectionInfo",
    "hubName": "%SIGNALR_HUB_NAME%",
    "connectionStringSetting": "SIGNALR_CONNECTION_STRING",
    "direction": "in"
  }, ...
]
```

A serverless SignalR Service pedig már egy a felhőszolgáltató által menedzselt szolgáltatás, így annak működését már nem kell felprogramoznunk, innen már működik céljának megfelelően.

5.2.2.3 Kliens oldali megvalósítás

A React alkalmazás a @microsoft/signalr [21] NPM csomagot használja a SignalR hubbal való kapcsolat böngészőből való kezelésére. A kapcsolat felállításához szükséges rövid kódrészlet a következő:

```
import {
  HubConnection, HubConnectionBuilder, LogLevel
} from '@microsoft/signalr'

export const fetchSignalrConnection = (): HubConnection => {
  return new HubConnectionBuilder()
    .withUrl(`${API_HOST}${NOTIFICATION_PATH_PREFIX}`)
    .withAutomaticReconnect()
    .configureLogging(
      process.env.NODE_ENV == 'production' ?
        LogLevel.Information : LogLevel.Debug
    ).build()
}
```

A `HubConnectionBuilder` osztály néhány beállítás (szerver URL, automatikus újracsatlakoztatás, naplózási szint) elvégzése mellett felépíti a kapcsolatot. A `fetchSignalrConnection` függvény által visszaadott `HubConnection` példányon tagmetódusok hívásával innentől kezdve indíthatjuk az üzenetfogadást.

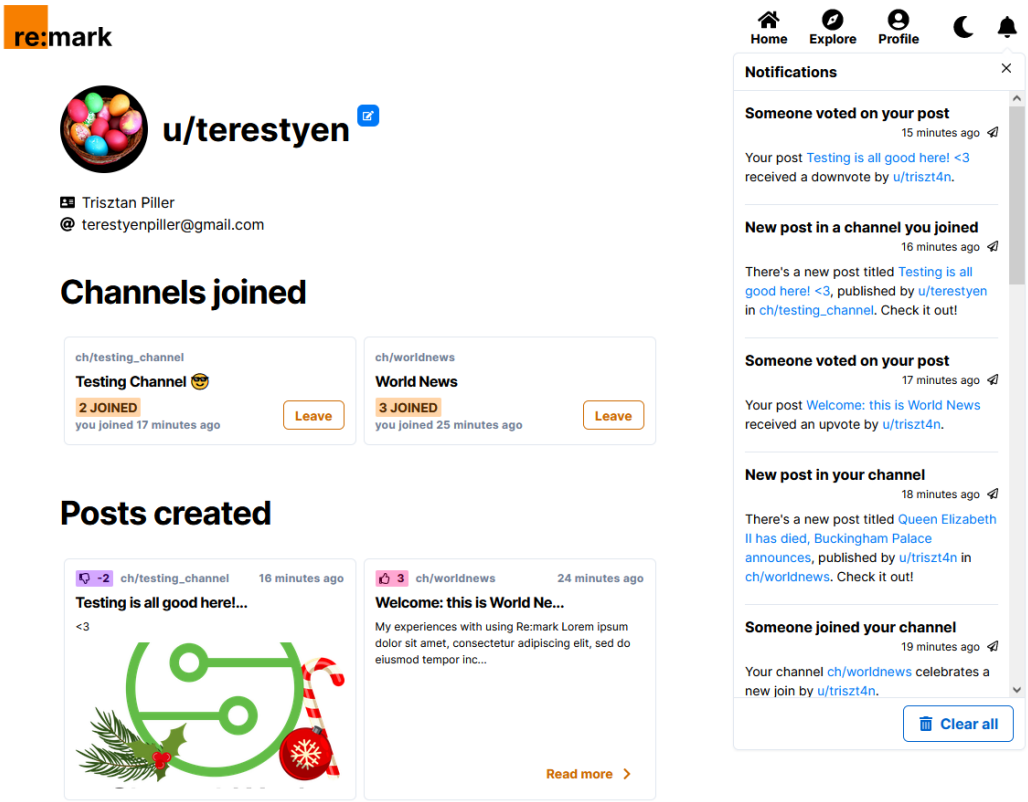
```
const signalrConnection = fetchSignalrConnection()

const notifHandler = (notification: NotificationView, userId: string) => {
  if (notification.userId === userId) {
    setNotifications((oldArray) => [...oldArray, notification])
    setShowNotificationCircle(true)
  }
}

const startConnection = async (userId: string) => {
  try {
    await signalrConnection.start()
    signalrConnection.on(
      `newNotification`,
      (notif) => notifHandler(notif, userId)
    )
  } catch (err) {
    setTimeout(startConnection, 5000)
  }
}
```

Az alkalmazásban az értesítések globális kezelésére egy külön kontextus készült a React *Context API*-ját használva, amely biztosítja, hogy az értesítések adatait ne csak propokon keresztüli top-down adatkötéseken keresztül tudjuk komponensekből elérni.

A fenti kódrészletek is ebben a kontextus komponensben kerültek elhelyezésre. A `HubConnection` típusú objektumon hívható `on(hubMethodName, eventHandler)` tagmetódussal eseménykezelő függvényt adunk bejövő üzenetek kezelésére. Az eseménykezelő (`notifHandler`) leválogatja a bejelentkezett felhasználónak szánt üzeneteket és frissíti az eddigi értesítések tömbjét, valamint egy narancssárga pöttyöt tesz láthatóvá a csengő ikonnál jelezve a felhasználó felé új értesítés érkezését.



14. ábra. Megvalósított profil nézet az értesítések felbukkanó ablakával

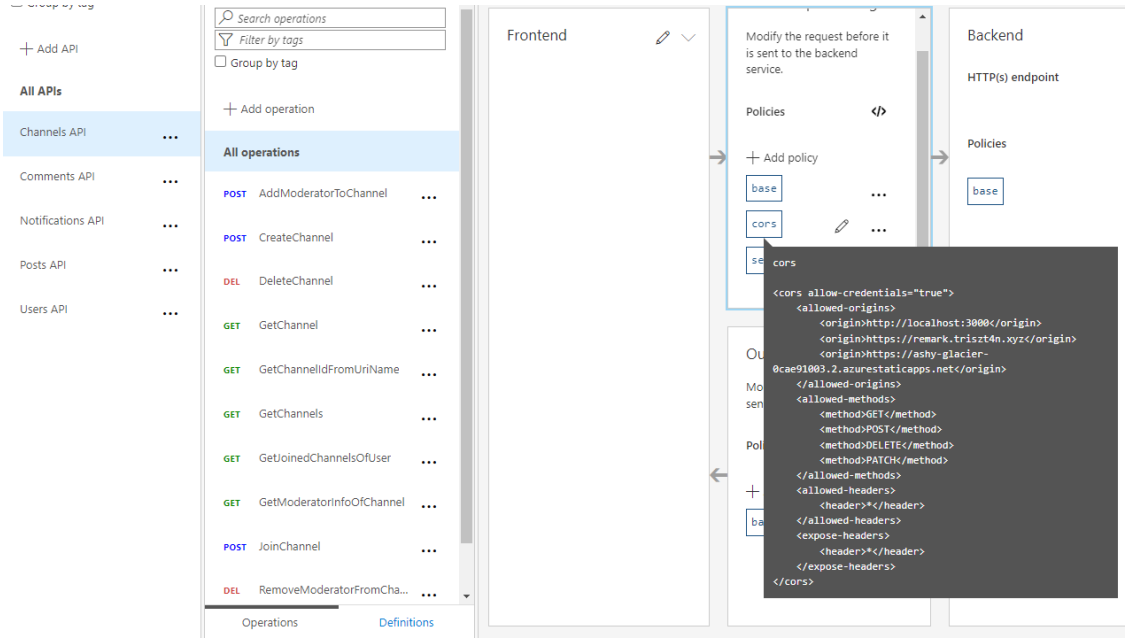
5.3 További szerver oldali megoldások

A szerver oldal HTTP végpontjai egy serverless API Management példány mögött fogadják a kéréseket. Az 15. ábra a csatornákat kezelő Function Appnak a végpontjait mutatja az API Management kezelésében.

Az ábrán a CORS (Cross-Origin Resource Sharing) beállításokban is látható, mely kezdeményező felek bejövő kéréseit, valamint milyen metódusokat engedélyez feldolgozni az kiszolgáló.

Az API-k mögött pedig a Function Appok HTTP trigger függvényei lelhetőek. Tulajdonképpen minden függvény egy-egy CRUD funkcionalitásért felelős entitásonként.

Egy-egy Function App TypeScript projektje a Git repository *functions* könyvtára alatt *remark-functions-XYZ* könyvtárban foglal helyet, ahol az XYZ kicserélhető a fő entitások angol nevére: *channels*, *comments*, *notifications*, *posts*, *users*.



15. ábra. A csatornákat kezelő API Management operációk és a közös CORS beállítások

Egy-egy ilyen TypeScript projekt úgy épül fel, hogy a függvények által közösen használt kódok a *lib* mappában kerültek elhelyezésre. Az itt található fájlok a következőket valósíthatják meg:

- adatbáziskapcsolat felépítését segítő függvények,
- egyedi Cosmos DB lekérések SQL nyelvű összeállításai,
- Blob Storage és Service Bus API használathoz implementált saját függvények,
- modelltípusokból származó saját felhasználású típusok deklarációi,
- bemenet validációs kód.

5.3.1 Adatelérési réteg

A 5.3 szülőfejezetben leírt *lib* mappában *dbConfig.ts* fájlokban írtam meg a közös felhasználású adatbáziskapcsolat inicializációjának logikáját, amelyet függvényekbe rendeztem, és így ajánlottam ki a felkonfigurált kliens objektumokat az üzleti logika felé. A *Cosmos DB SDK*-ja könnyen érthető és használható, következő inicializációs függvényeket tudtam készíteni vele:

```
import { CosmosClient, Database } from '@azure/cosmos'
import { RemarkDatabaseContainerId } from '@triszt4n/remark-types'

export const fetchCosmosDatabase = () => {
  const client = new CosmosClient({
    endpoint: process.env.COSMOS_DB_ENDPOINT,
    key: process.env.COSMOS_DB_KEY
  })
```

```
  return client.database(process.env.COSMOS_DB_DATABASE_ID)
}

export const fetchCosmosContainer = (
  database: Database, containerId: RemarkDatabaseContainerId
) => {
  return database.container(containerId)
}
```

Csupán a kulcsokat kellett átadni a kliens objektumok generálásakor, amelyek környezeti változókból származnak. Egy példa arra, milyen letisztulttá vált így egy HTTP triggerben egy adott id-vel rendelkező poszt lekérése *Cosmos DB SDK*-val és a kiajánlott *fetch* kezdetű függvényekkel:

```
const id = context.bindingData.id as string
const database = fetchCosmosDatabase()
const postsContainer = fetchCosmosContainer(database, 'Posts')
const postItem = postsContainer.item(id, id)
```

5.3.2 Kaszkádolt törlés

Az alkalmazás főbb entitásai között tartalmazási kapcsolat áll fenn. Kommentek csak a tartalmazó posztjukkal együtt léteznek, posztok csak a tartalmazó csatornájukkal együtt léteznek. Mivel az adatbázissémát úgy alakítottam ki, hogy ezek lépcsőzetesen épülnek egymásra, így a törlésük is kaszkádolt kell legyen. Azaz, ha törlésre kerül egy csatorna, vele együtt a posztoknak és a posztok kommentjeinek eltávolítását is meg kell oldani.

Megoldásként Cosmos DB trigger függvényeket alkalmaztam. Abban az esetben, ha a csatorna törlésére specializált HTTP végpont meghívásra kerülne, a HTTP trigger függvényben csupán „soft delete”, azaz *puha törlés* megy végbe. Ez azt jelenti, hogy az objektum megjelölésre kerül egy `isDeleted` mező igazra állításával és az egyed adatbázisba is így kerül mentésre. Ezután, mivel változás történt a táblában, elsütésre kerül a Cosmos DB trigger függvények egyike, amit említettem:

```
await Promise.all(
  documents.filter((ch) => !!ch.isDeleted).map(async (channel) => {
    // Soft delete posts of channel:
    const { resources: posts } = /* query posts of channel ... */
    await Promise.all(posts.map(async (post) =>
      postsContainer.item(post.id, post.id).replace({
          ...post,
          isDeleted: true
      })
    ))
    const { resources: channelJoins } = /* query joins of channel ... */
    await Promise.all(
      channelJoins.map(async (join) => {
```

```
            // Create notif about channel deletion for joined people ...
            // Hard delete join ...
          })
        )
        // Hard delete channel
        return channelsContainer.item(channel.id, channel.id).delete()
      })
    )
```

Ahogy a kódban is látszik, a kódkommentek jelölik, milyen lépésekre van szükség: soft delete műveletet végzek a tartalmazott posztokon, végleg törlöm a követéseket és erről a követőknek értesítést küldök, majd végleg törlöm magát a csatornát (*hard delete*).

HTTP-n keresztül kért poszt törlésénél hasonlóképp soft delete-elem a kommenteket, végleg törlöm a szavazatokat, hozzáadott képet és magát a posztot. Kommentnél pedig a Cosmos DB trigger már csak szavazatokat és a kommentet magát törli véglegesen.

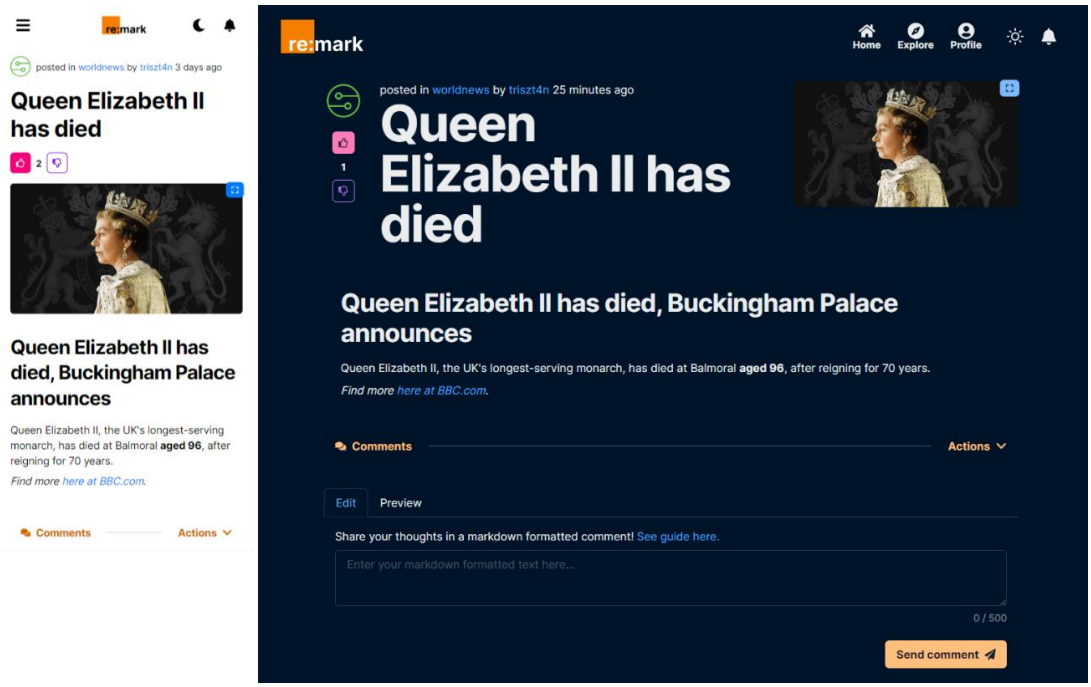
5.4 További kliens oldali megoldások

5.4.1 Éjszakai és világos mód

A React alkalmazásban kihasználásra került a Chakra UI több képessége is:

- éjszakai és világos módra is felkészítettem az alkalmazást,
- reszponzív lett a felület, azaz többféle képernyőméretre is optimalizáltam,
- egyénileg választott 4 színnel és Inter betűtípussal saját témát adtam az oldalnak.

A 16. ábra is ezeket demonstrálja egy poszt bemutatásában nappali módú mobilnézetben és egy éjjeli módú asztali nézetben.



16. ábra. Poszt kétféle nézetének összehasonlítása különböző eszközbeállításokkal

5.4.2 Űrlapok

Űrlapok egységes és a könnyű validációs logika kialakítására a React Hook Form NPM csomagot importáltam a projektbe. Mivel az alkalmazásban sokszor kell szöveges beviteli mezőket beilleszteni űrlapokba, így készítettem egy újrafelhasználható `TextField` komponenst.

```
export const TextField = ({
  title, name, helper, defaultValue, validationOptions: {
    max, min, required, pattern, setValueAs
  } = {}
}: Props) => {
  const { register, formState: { errors } } = useFormContext()

  return (
    <FormControl isRequired={required} isInvalid={!!errors[name]}>
      {title && <FormLabel htmlFor={name}>{title}</FormLabel>}
      <Input id={name} type="text" defaultValue={defaultValue}
        {...register(name, {
          required: required ? 'Required field' : false,
          maxLength: max ? { value: max, message: `...` } : undefined,
          minLength: min ? { value: min, message: `...` } : undefined,
          pattern,
          setValueAs
        })}
      />
      {errors?.[name] ?
        (<FormErrorMessage>{errors[name]?.message}</FormErrorMessage>) :
        (helper && <FormHelperText>{helper}</FormHelperText>)}
    </FormControl>
  )
}
```

Ennek megadható propokban a mező címe (`title`), mező alatti magyarázat szövege (`helper`), alapérték (`defaultValue`) és a validációs szabályok (`validationOptions`). Ezen szabályok lehetnek a minimum (`min`) és maximum (`max`) karakterszám, kötelezően kitöltendő-e (`required`), *RegEx* illeszkedési szabály (`pattern`) és esetleg egy `setValueAs` függvény, amely transzformálja szerver oldalra való leküldés előtt a felhasználó által beírt értéket.

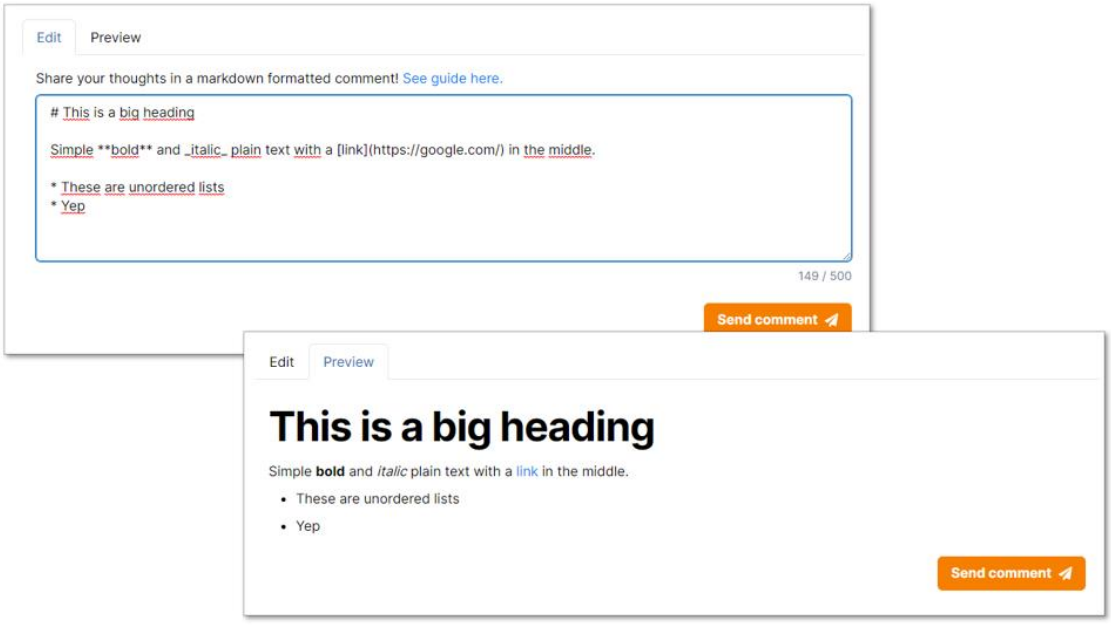
A komponens a `useFormContext` *React hook*ot használja, amely az űrlap kontextusában használható műveleteket és változókat ajánlja ki. Egy ilyen függvény a `register` függvény, amellyel az űrlap kontextusába regisztráltam be magát a beviteli mezőt tulajdonságaival. Másik pedig az űrlap `errors` állapotváltozója, amelyet beillesztettem a komponensbe, hogy lássa a felhasználó a validációs hibákat.

Fájlfeltöltési beviteli mezőre is készítettem egy újrafelhasználható komponenst `FileUpload` néven. Ezen komponensek a forrás `client/src/components/form-elements` könyvtárában találhatóak.

5.4.3 Markdown támogató szövegformázás

A React kliens alkalmazásban nyers *Markdown* nyelven lehet tartalommal lehet feltölteni kommenteket, posztok szövegét vagy csatornák leírását. A Markdown tartalmak nyers szövegként kerülnek leküldésre a szerver oldalra, és adatbázisban is ilyen formátumban kerülnek tárolásra. A felületre is nyersen kerül letöltésre a szöveg, és egy React komponens transzformálja szépen formázott HTML elemekké. Erre a funkcionalitásra a react-markdown NPM csomag `ReactMarkdown` komponensét használtam, illetve a chakra-ui-markdown-renderer NPM csomagot vetettem még be hozzá, hogy a Chakra UI témájának megfelelő CSS stílusokkal kerüljenek renderelésre a Markdownból transzformált HTML elemek.

`RemarkEditor` néven készítettem egy React komponenst, amelynek a célja olyan űrlapba ágyazható szöveges beviteli mezőt biztosítani, amellyel egy „Preview” tabon a Markdown szövegünk HTML-re formázott látványáról előnézetet kapunk (17. ábra).

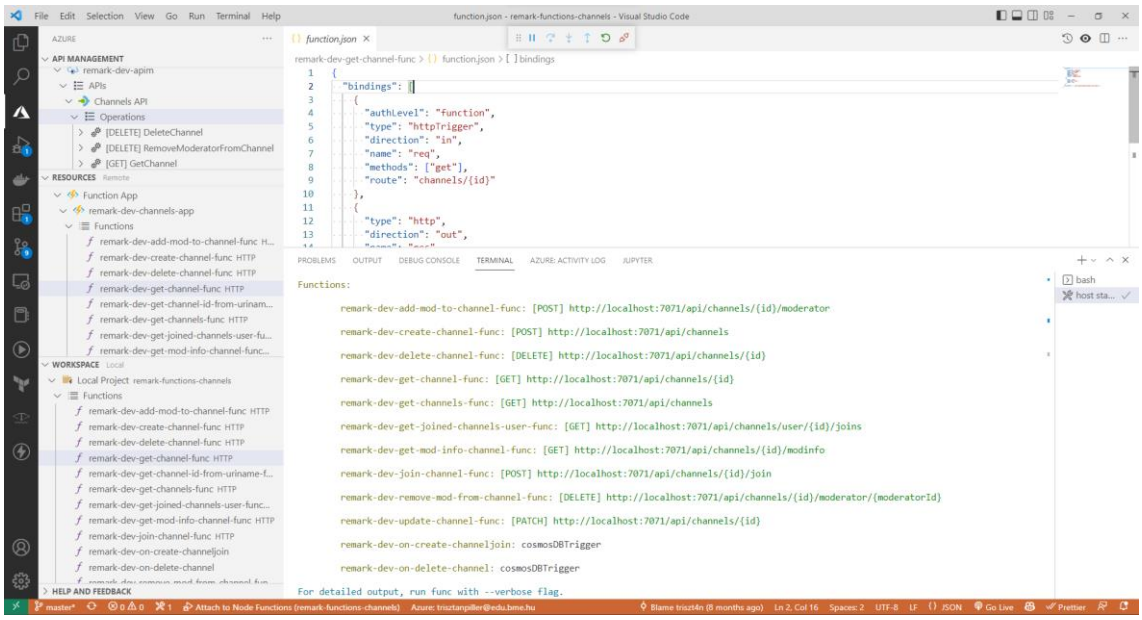


17. ábra. A RemarkEditor szerkesztési és előnézeti módban való nézetei

5.5 Fejlesztőeszközök

A legfontosabb fejlesztőeszközöm a Visual Studio Code (VSCode) volt, amely használata során kihasználtam a kódszerkesztőhöz készült bővítményeket. Egyik legfontosabb az Azure Tools nevű bővítménycsomag volt. Segítségével az Azure Function Appok projektjeit és bevezető kódját generáltam. Az elején – amíg még nem használtam Terraformot –, addig a felhőbe is ezzel az eszközzel telepítettem az alkalmazásaim. A leghasznosabb tulajdonsága a lokálisan hibakereső üzemmódban futtatható Azure Function platform volt, amellyel akár felhőbe telepítés nélkül tesztelhettem az üzleti logikát. Ennek futását ábrázolja a 18. ábra.

A hatékony kódolást segítette az ESLint statikus kódelemző és a Prettier kódformázó eszközök. Minden TypeScript projektemben használtam ezeket az eszközöket, hogy számomra megszokott és a legjobb gyakorlatoknak megfelelő („*best practice*”) és konzisztens kódolási stílust tudjak alkalmazni minden projektben. VSCode-ban a Prettier bővítmény segítségével be tudtam azt is állítani, hogy a kódolás során mentés esetén a szerkesztett kód automatikus formázásra kerüljön az előírt szabályoknak megfelelően.



18. ábra. Function App futtatása lokálisan VSCode bővítménnyel

6 Üzemeltetés

Felhő alapú környezetekben az üzemeltetés többféleképp megoldható. Felhőszolgáltatóknak általában van böngészőből is elérhető szolgáltatás-menedzsment portáljuk. Ehhez tartozhat igényes dokumentáció, sok sablon és alapbeállítás szolgáltatások élesítésére. Az Azure portál is ilyennek számít.

Viszont ez a fajta manuális konfigurálás több gondot is okozhat: megfeledkezünk erőforrások törléséről, két azonos szolgáltatás felkonfigurálása során keletkezhetnek eltérések, illetve egy alrendszer új alapokra migrálása akár órákat felölelő leállást okozhat. Ezek mind sok hibát, kiesést és nem várt költséget is tudnak okozni.

Az IaC arra hivatott, hogy a felsorolt problémákat elhárítsa és lehetőséget ad vállalati szinten bevált gyakorlatok kialakítására, konzisztens, gyors és megbízható telepítési folyamatok végrehajtására.

6.1 Terraform

Az implementált infrastuktúra a Git repository *infrastructure* könyvtárában található. Abban pedig a *modules* mappában találhatóak az újrafelhasználható Terraform moduljaim.

6.1.1 Implementációk

Törekedtem a legfrissebb Terraform nyelvverziót és a legfrissebb Azure platformhoz szükséges hivatalos *Terraform providert* (`hashicorp/azurerm`) használni a fejlesztés során. A `main.tf` fájlokban találhatóak a verzióbeállítások:

```
terraform {
  required_providers {
    azurerm = {
      source  = "hashicorp/azurerm"
      version = "~> 3.32.0"
    }
  }
  required_version = ">= 1.1.0"
}
```

A Static Web App infrastruktúraleírását szinte teljesen a Terraform dokumentáció példakódja alapján megvalósítottam meg. A *modules/static-web-app* alatt található a modulja, amelyben magát az `azurerm_static_site` erőforrást, és hozzárendelt

`azurerm_static_site_custom_domain`, azaz az egyedi domain beállítását írtam meg. A modul a `client.tf` fájlban kerül használatra.

Az API Management (`apim.tf`) esetében is példakódot követtem, viszont a leírásában fontos a `sku_name = "Consumption_0"` sor, amellyel serverless üzemmódban kerül telepítésre a szolgáltatás. A SignalR szolgáltatásnál (`signalr.tf`) a `service_mode = "Serverless"` beállítás biztosította ugyanezt a célt.

A Function Appokat indokolt volt modulba szervezni, mert működésük alapja megegyezik. A `modules/function-app-with-api` könyvtárban érhető el a modul, és ahogy már a mappa neve is sejteti, a példányosítandó Function App számára API Managementes API-t, a Function App függvényei számára API operációkat is deklaráltam. Az `api.tf` és `operations.tf` fájl rejti ezeket. Az API operációk és a függvények beállításait az api_ops_config bemeneti változóból oldottam meg, amely egy mapben adja át az API operációinak tulajdonságait. A Function App diagnosztizálására *Application Insights* példányt is deklaráltam.

A Cosmos DB számára fiókot, adatbázist és konténereket kellett deklarálnom. Ezek is modulba kerültek. A konténerek beállításait map típusú bemeneti változóként fogadja az általam írt modul. A kódduplikáció elkerülése érdekében a for_each metaargumentumot használtam, amellyel többszöröződik az erőforrás és a map értékeivel tölti ki a konfigurációt:

```
resource "azurerm_cosmosdb_sql_container" "sql-containers" {
  for_each = var.containers

  name                = each.key
  resource_group_name = var.resource_group_name
  account_name        = azurerm_cosmosdb_account.db-account.name
  database_name       = azurerm_cosmosdb_sql_database.db.name
  partition_key_path  = each.value.partition_key_path

  indexing_policy {
    indexing_mode = "consistent"

    included_path {
      path = "/*"
    }
  }
}
```

Az összes API beállítását a projektben `locals` blokk használatával lokális változóba tettem a `locals-apiconfig.tf` fájlban. Az adatbázis konténerek változóit is `locals` blokkban írtam meg, a `locals-cosmos.tf` fájlban találhatóak.

Az infrastruktúra felhőben való élesítését jelenleg lokális Terraform klienssel, manuálisan tudjuk elvégezni, hogy könnyedén megbizonyosodhassunk az élesítési terv helyességéről. Szükséges lokális Azure Identity beállítása a terminálunk megfelelő környezeti változóinak definiálásával, hogy a platformra lokális számítógépről is telepíteni tudjunk. Ehhez a Hashicorp fejlesztői oktatóanyagát [22] követtem. Majd pedig `terraform apply -var-file=dev.tfvars` parancs kiadásával lehetséges elindítani az erőforrások élesítését a Terraform kliense által.

6.1.2 Terraform state

A Terraform által menedzselt felhőerőforrásokról állapotfájlban, Terraform state fájlban tárolja a rendszer a konfigurációs részleteket. Az ebben található erőforrásokat tudja a Terraform kliens befolyásolni, illetve ezek változásait képes követni is.

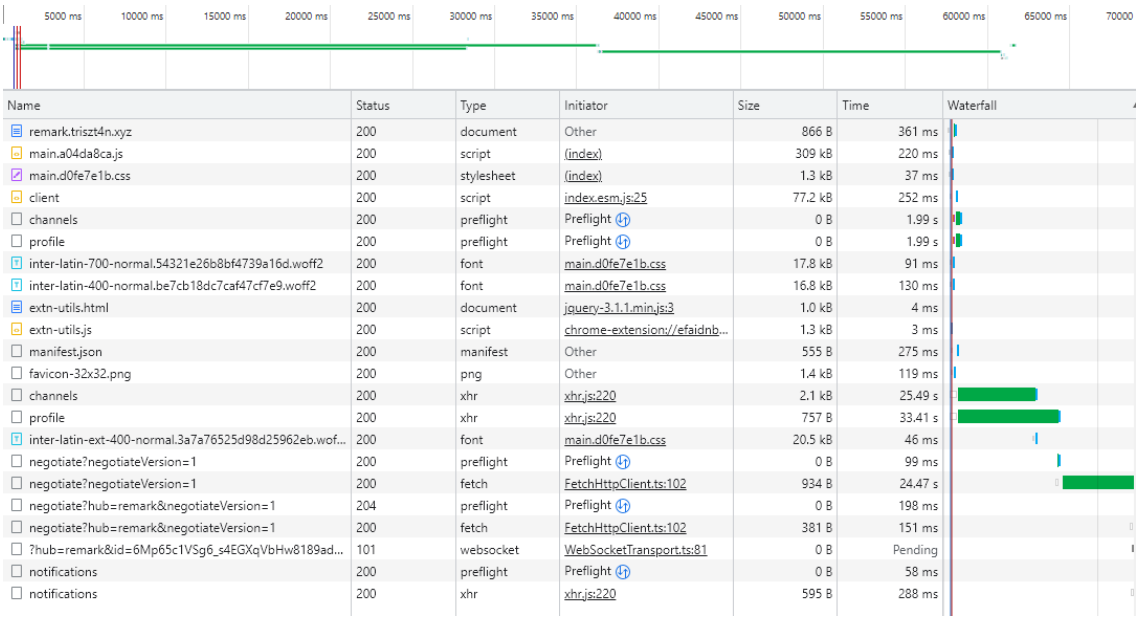
A Terraform state tárolható lokálisan a futtató eszközön, vagy megadható egy *Terraform backend*, amely egy felhőbeli tárhely a state fájl számára. Nagyobb biztonságot és elosztott állapotkezelést tesz lehetővé.

```
terraform {
  backend "azurerm" {
    resource_group_name  = "remark-dev-rg"
    storage_account_name = "remarkdevtfstate"
    container_name       = "remarkdevtfstate"
    key                  = "terraform.tfstate"
  }
}
```

A projektem state fájlját is Blob Storage konténerbe helyeztem `backend` blokk (lásd kód) és egy Storage fiók és konténer erőforrás blokk deklarálásával.

6.1.3 Értékelés

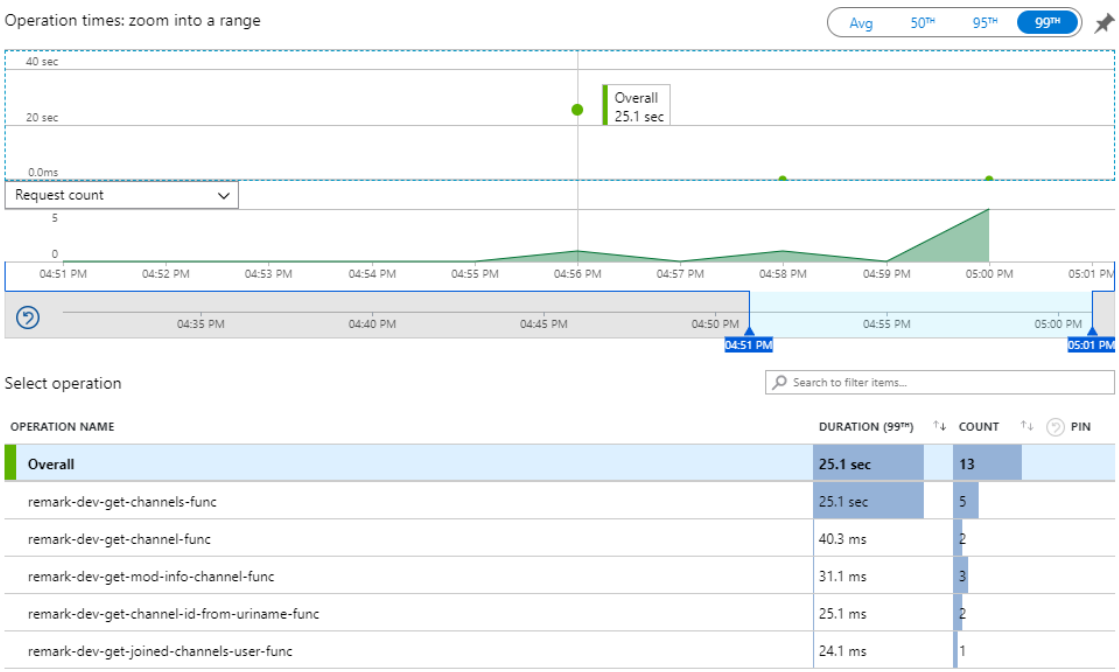
A felépített infrastruktúra minden eleme serverless üzemmódban működő szolgáltatás lett. Ennek köszönhetően az alkalmazás tesztelése során többször futottam bele a *cold start* jelenségébe.



Name	Status	Type	Initiator	Size	Time	Waterfall
remark.triszt4n.xyz	200	document	Other	866 B	361 ms	
main.a04da8ca.js	200	script	(index)	309 kB	220 ms	
main.d0fe7e1b.css	200	stylesheet	(index)	1.3 kB	37 ms	
client	200	script	index.esm.js:25	77.2 kB	252 ms	
channels	200	preflight	Preflight	0 B	1.99 s	
profile	200	preflight	Preflight	0 B	1.99 s	
inter-latin-700-normal.54321e26b8bf4739a16d.woff2	200	font	main.d0fe7e1b.css	17.8 kB	91 ms	
inter-latin-400-normal.be7cb18dc7caf47cf7e9.woff2	200	font	main.d0fe7e1b.css	16.8 kB	130 ms	
extn-utils.html	200	document	jquery-3.1.1.min.js:3	1.0 kB	4 ms	
extn-utils.js	200	script	chrome-extension://efaidnb...	1.3 kB	3 ms	
manifest.json	200	manifest	Other	555 B	275 ms	
favicon-32x32.png	200	png	Other	1.4 kB	119 ms	
channels	200	xhr	xhr.js:220	2.1 kB	25.49 s	
profile	200	xhr	xhr.js:220	757 B	33.41 s	
inter-latin-ext-400-normal.3a7a76525d98d25962eb.wof...	200	font	main.d0fe7e1b.css	20.5 kB	46 ms	
negotiate?negotiateVersion=1	200	preflight	Preflight	0 B	99 ms	
negotiate?negotiateVersion=1	200	fetch	FetchHttpClient.ts:102	934 B	24.47 s	
negotiate?hub=remark&negotiateVersion=1	204	preflight	Preflight	0 B	198 ms	
negotiate?hub=remark&negotiateVersion=1	200	fetch	FetchHttpClient.ts:102	381 B	151 ms	
?hub=remark&id=6Mp65c1VSg6_s4EGXqVbHw8189ad...	101	websocket	WebSocketTransport.ts:81	0 B	Pending	
notifications	200	preflight	Preflight	0 B	58 ms	
notifications	200	xhr	xhr.js:220	595 B	288 ms	

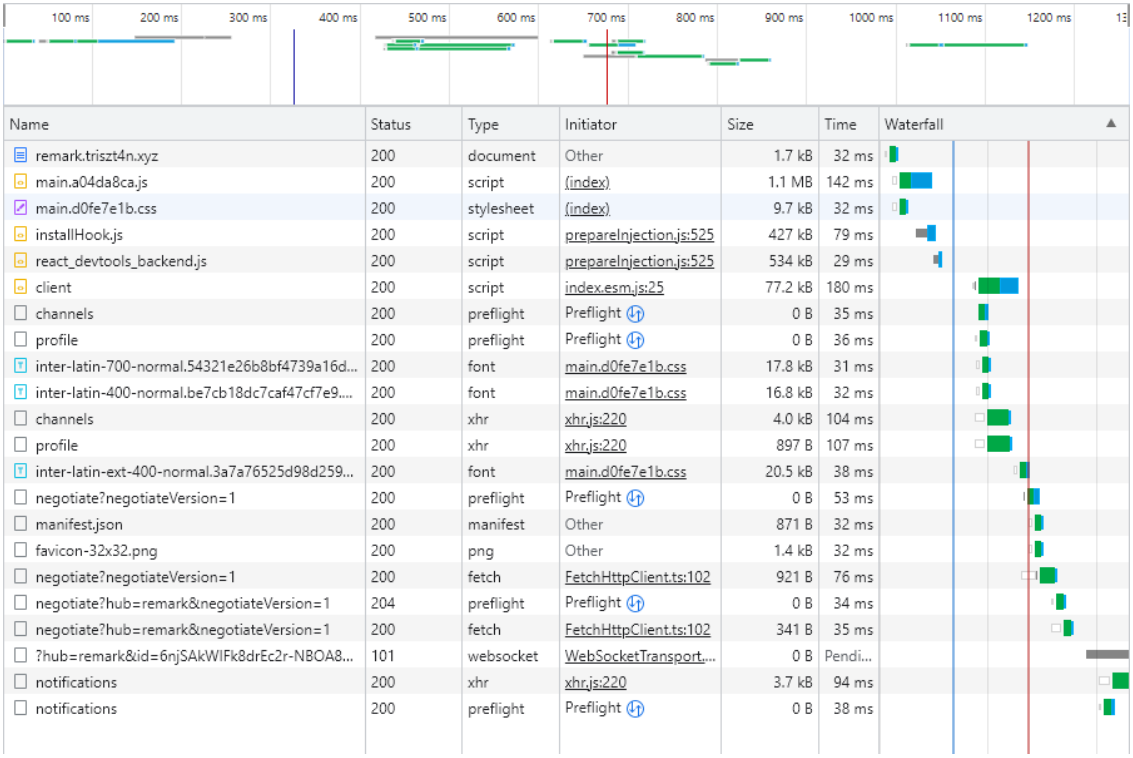
19. ábra. Böngésző hálózatdiagnosztikai nézete cold start esetén a küldött kérésekről

Egész napos kihagyás után megnyitottam a Re:mark weboldalát, a 19. ábra a Chrome böngésző „waterfall chart”-ját mutatja. A főoldalon található csatornák letöltése két kérés idejével jellemezhető. Az egyik az 5. sorban található preflight kérés, azaz a végpont elérhetőségét ellenőrző kérés 1,99 másodpercet vett igénybe. Úgy vélem, ennek a kérésnek a hosszú válaszideje abból eredeztethető, hogy az API Management példány is serverless üzemmódú, így annak a használatakor is szükséges kivárni az első rendszerfelállási időt. A 13. sorbéli kérés pedig 25,49 másodpercet vett igénybe.



20. ábra. Application Insights szolgáltatás teljesítménymérése a csatornák Function Appjáról

Minden további kérés, amely a csatornákkal kapcsolatos volt (pl. újratöltés, csatorna részleteinek lekérése) már sokkal hamarabb töltött be ezután. A 20. ábra is a tapasztaltakat szemlélteti a csatornákkal kapcsolatos műveleteket összefogó Function Appra kötött Application Insight szolgáltatásban. A napló mélyére ásva még arról is találtam információt, hogy az első kérésnél valóban cold start történt. Minden további kérés függvénybeli feldolgozása már alig lépte meg a 25-30 ezredmásodpercet, illetve maguk a HTTP kérések (preflighttal együtt) kb. 150-200 ezredmásodpercet tettek ki (lásd 21. ábra), azok esetében is valószínűleg a válaszidő nagy részét az hálózaton való szállítás tehette ki.



21. ábra. Böngésző hálózatdiagnosztikai nézete cold start után

Cold start folytonos használat mellett nem érzékelhető. Viszont roppant kellemetlen lehet azon felhasználók számára az élmény, amikor ők az első kérést intézők egy hosszabb kihagyás után. Ennek megoldására a jövőben megoldás lehet a Function Appok újraszervezése, és több HTTP végpont egy függvényre koncentrálása, és a jó kódszervezéshez pedig egy Node.js keretrendszer alkalmazása. Szóba kerülhetnek kódoptimalizálási módszerek, például a Function Appok TypeScript kódját modulcsomagoló eszközökkel (pl. Webpack) képesek volnánk optimalizálni. [23]

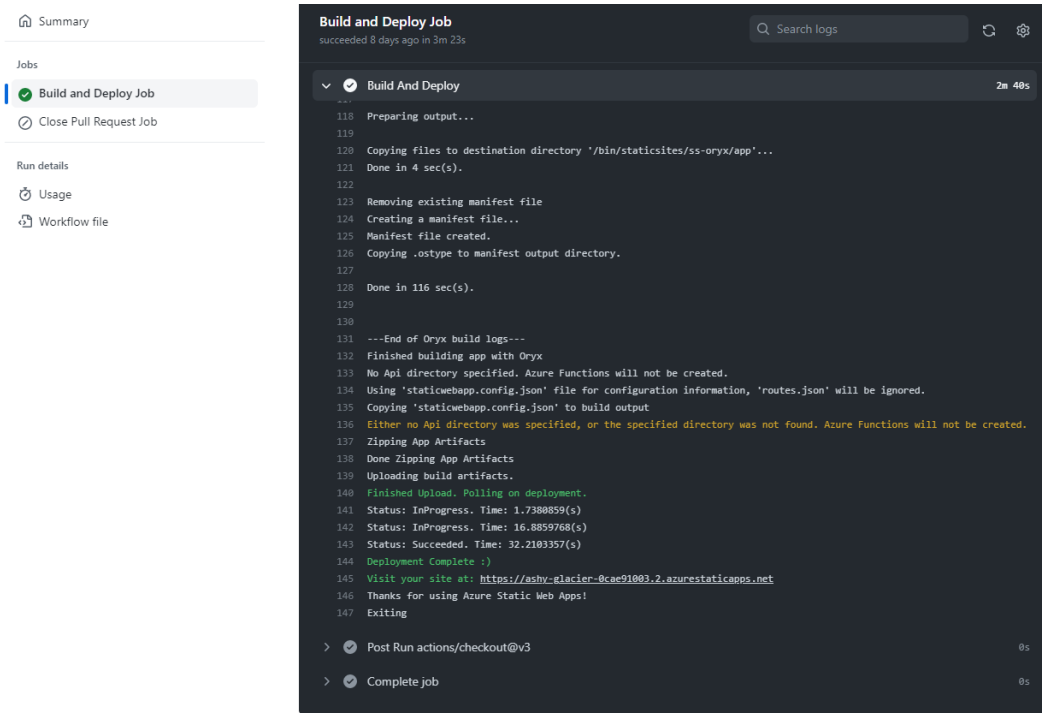
6.2 GitHub Actions

A teljes infrastruktúrában vannak szolgáltatások, amelyeknek futtatható kódra van szükségük a működésükhöz. Az Function Appok és a Static Web App kódjának feltöltését, valamint az NPM csomagjaim publikálását egy *Contiuous Integration / Continuous Delivery* (CI/CD) eszközzel, a *GitHub Actions* segítségével végeztettem el.

Az eszköz képes Git eseményekre munkafolyamatokat lefuttatni. Egy GitHub Action működését *YAML* (YAML Ain't Markup Language) nyelven lehet írni, és a repository `.github` mappájában kell elhelyezni. A környezet automatikusan futtatja a kódban definiált eseményekre a megírt munkafolyamatokat.

```yaml
- name: "Run Azure Functions Action"
  uses: Azure/functions-action@v1
  with:
    app-name: ${{ env.AZURE_FUNCTIONAPP_NAME }}
    package: ${{ env.AZURE_FUNCTIONAPP_PACKAGE_PATH }}
    publish-profile: ${{ secrets.USERS_FUNCTION_PUBLISH_PROFILE }}
```

A kódrészlet egy „*job*”-ot (munkát) ír le, a felhőbe telepítést egy már az Azure által publikált GitHub Action, az `Azure/functions-action` használatával.



22. ábra. Egy sikeresen lefutott GitHub Action folyamat a Static Web App új kódjának élesítésére

A Cosmos és Service Bus triggerfüggvényeket nem írtam Terraform infrastruktúrába, ugyanis a Function Appok kódját telepítő GitHub Action által

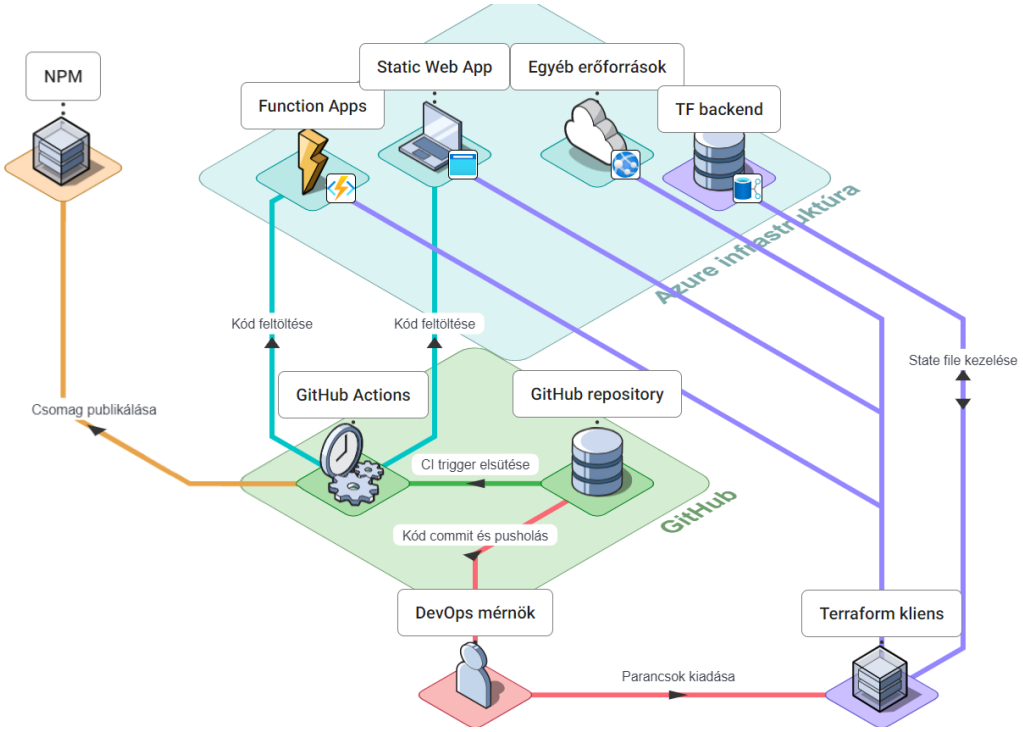
feltelepítésre kerülnek. Maguk a Function Appokat és a HTTP triggereket viszont Terraformmal telepítettem több indokból is.

A függvények API-kra kapcsolódnak, és a kapcsolatokat fel kell konfigurálni, nem elég a kódot telepíteni az Azure platformra. Maguk az appok környezeti változókat tartalmaznak, amelyeket jobb kódból, egy helyről irányítani. Ezek a változók tulajdonképpen a szolgáltatásokkal való kapcsolatépítéshez szükséges kulcsok. A kulcsokat Terraform segítségével dinamikusan tudtam rákötni az erőforrások kimeneti változóiból a környezeti változókra:

```
app_settings = {
  COSMOS_DB_ENDPOINT            = module.cosmos-db.db-endpoint
  COSMOS_DB_DATABASE_ID         = module.cosmos-db.db-name
  COSMOS_DB_KEY                 = module.cosmos-db.db-key
  remarkcosmosdb_DOCUMENTDB     = module.cosmos-db.connection-string
  JWT_PRIVATE_KEY               = var.jwt_private_key
  SERVICE_BUS_CONNECTION_STRING =
    module.service-bus.service-bus-connection-string
  SERVICE_BUS_QUEUE_NAME       = module.service-bus.service-bus-queue-name
  SIGNALR_CONNECTION_STRING    = module.signalr.connection-string
  SIGNALR_HUB_NAME             = "remark"
  STORAGE_ACCOUNT_KEY = azurerm_storage_account.images.primary_access_key
  STORAGE_ACCOUNT_NAME         = azurerm_storage_account.images.name
}
```

6.3 Áttekintés

A tárgyalt CI/CD folyamatokat az 23. ábra ábra összesíti.



23. ábra. CI/CD folyamatok

7 Összefoglalás

A Re:mark egy teljesen serverless felhő alapú szoftver. A végeredmény elérhető a https://remark.triszt4n.xyz webcímen. A projekt jól demonstrálja ilyen rendszerek készítésének kihívásait mind tervezési, implementációs és üzemeltetési szempontból.

Serverless mikroszolgáltatások előnyei, amiket tapasztaltam: a skálázhatóság, a dinamikus szolgáltatásbekötés, könnyen cserélhető komponensek lehetősége és a felelősségek hatékony eloszlása komponensek között. Megtapasztalt hátrányok közé sorolom a *cold startot* és a rendszer felépítéséből származó *kognitív komplexitást*. Ide tartozik még a *vendor lock-in*, azaz a terjesztőtől (itt: felhőszolgáltatótól) való függés. A konfiguráció és a szolgáltatások összekötését az Azure platformjára implementáltam, más felhőben egészen máshogy látom, hogy lehetne megvalósítani az architekturális tervet. Jó ötlet lehet a probléma áthidalására a szolgáltatások konténerizálása *Docker* bevetésével. Azonban ezzel elveszítenénk a serverless adta előnyöket.

Felhő alapú szoftverekkel foglalkozó mérnök informatikusok számára a piac számtalan lehetőséget kínál, nagy a kereslet a munkapiacon. [24] A webfejlesztésen kívül van lehetőség felhő alkalmazására az IoT (Internet of Things), adatbányászat és gépi tanulás területén is.

7.1 További fejlesztési lehetőségek

7.1.1 Teljes szöveges keresés

A weboldalon jelenleg nincs keresési funkcionalitás bevezetve sehol. Egy közösségi oldalon gyakran használt funkció a teljes szöveges keresés, amellyel akármilyen webes tartalomban (pl. csatornák címében, posztok szövegében, felhasználók adataiban) kereshetünk.

7.1.2 Ajánlórendszer

Egy nagy léptékű fejlesztés lehet ajánlórendszer kifejlesztése. Relevanciapontok származhatnak látogatottságból, illetve a szavazatokból. Így kialakítható egy oldal a Re:markon, amely ajánlott csatornákat és posztokat listázna felhasználóra igényeire szabva.

8 Irodalomjegyzék

[1] „Reddit | Wikipedia,” [Online]. Available: https://en.wikipedia.org/wiki/Reddit.

[2] „What is serverless?,” Red Hat, [Online]. Available: https://www.redhat.com/en/topics/cloud-native-apps/what-is-serverless.

[3] L. McCoy, „Microsoft Azure Explained: What It Is and Why It Matters,” CCB Technology, [Online]. Available: https://ccbtechnology.com/what-microsoft-azure-is-and-why-it-matters/.

[4] D. Harrington, „What is Terraform: Everything You Need to Know,” Varonis, 2022. [Online]. Available: https://www.varonis.com/blog/what-is-terraform.

[5] „What is Infrastructure as Code (IaC)?,” Red Hat, [Online]. Available: https://www.redhat.com/en/topics/automation/what-is-infrastructure-as-code-iac.

[6] W. Xu, „Benchmark Comparison of JavaScript Frameworks,” 2021.

[7] „Stack Overflow Trends in JS frameworks,” [Online]. Available: https://insights.stackoverflow.com/trends?tags=reactjs%2Cvue.js%2Cangular%2Cangularjs.

[8] „Chakra UI,” [Online]. Available: https://chakra-ui.com/.

[9] A. Goel, „10 Best Web Development Frameworks,” 2022. [Online]. Available: https://hackr.io/blog/web-development-frameworks.

[10] „Node.js Docs,” [Online]. Available: https://nodejs.org/en/docs/.

[11] „TypeScript for the New Programmer,” [Online]. Available: https://www.typescriptlang.org/docs/handbook/typescript-from-scratch.html.

[12] „What is npm?,” 2011. [Online]. Available: https://nodejs.org/en/knowledge/getting-started/npm/what-is-npm/.

[13] R. T. Fielding, „CHAPTER 5: Representational State Transfer (REST),” 2000.

[14] M. Chand, „Why and When to use Azure Functions,” 2021. [Online]. Available: https://www.c-sharpcorner.com/article/why-and-when-to-use-azure-functions/.

[15] P. S. R. Matli, „Overview of Azure Cosmos DB,” [Online]. Available: https://www.red-gate.com/simple-talk/cloud/azure/overview-of-azure-cosmos-db/.

[16] C. de la Torre, B. Wagner és M. Rousos, „.NET Microservices: Architecture for Containerized .NET Applications,” 2021.

[17] „What is Azure Service Bus?,” Microsoft, 2022. [Online]. Available: https://learn.microsoft.com/en-us/azure/service-bus-messaging/service-bus-messaging-overview.

[18] M. Jones, J. Bradley, N. Sakimura, Microsoft, P. Identity és NRI, „JSON Web Token (JWT),” IETF, 2015.

[19] „React OAuth2 Google NPM package,” [Online]. Available: https://www.npmjs.com/package/@react-oauth/google.

[20] „jsonwebtoken NPM package,” Auth0, [Online]. Available: https://github.com/auth0/node-jsonwebtoken.

[21] „SignalR client NPM package,” [Online]. Available: https://www.npmjs.com/package/@microsoft/signalr.

[22] „Build Infrastructure - Terraform Azure Example,” Hashicorp, [Online]. Available: https://developer.hashicorp.com/terraform/tutorials/azure-get-started/azure-build.

[23] E. Rokah, „Optimizing your Node.js lambdas with Webpack and Tree shaking,” 2019. [Online]. Available: https://medium.com/@erezro/optimizing-your-node-js-lambdas-with-webpack-and-tree-shaking-899c153403a9.

[24] N. Eide, „Supply, demand and talent constraints in the cloud workforce,” CIO Dive, 2020. [Online]. Available: https://www.ciodive.com/news/supply-demand-and-talent-constraints-in-the-cloud-workforce/570814/.