MŰEGYETEM 1782
Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Automatizálási és Alkalmazott Informatikai Tanszék

Azure serverless szolgáltatásokra épülő mikroszolgáltatás architektúrájú rendszer megvalósítása

Re:mark - webes közösségi tér

fejlesztői dokumentáció
BSc Önálló laboratórium (VIAUAL01)

Készítette
Piller Trisztán

Konzulens
Dr. Dudás Ákos

2022. 05. 24.

Tartalomjegyzék

1. Bevezetés

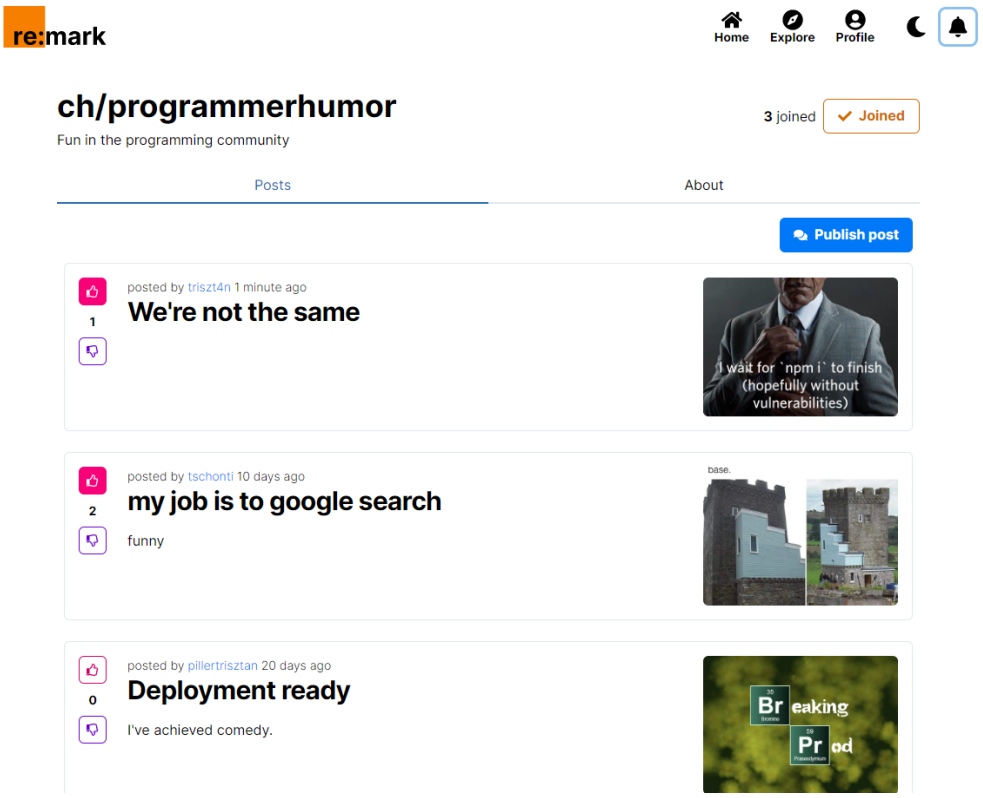
Ma már szinte mindenhonnan körbeövezik az embert a webes szolgáltatások, ezek közé tartoznak az internetes közösségi média oldalak és webes fórumok. Ezek működése legtöbbször nagyon sok komponensből áll: kezelni kell a felhasználókat (accounting), erőforrásokat (csatornák, posztok, kommentek, ezek egymásra való referenciái), valamint képeket, videókat, értesítéseket és akár csevegési felületet. Ilyen jól elkülöníthető komponensekre szedhető, nagyobb kiterjedésű alkalmazásoknak a rendszerét érdemes lehet a tervezésekor a mikroszolgáltatások architektúrába rendezni, valamint mivel a legtöbb funkcionalitása egy ilyen jellegű projektnek egyszerű, akár függvényszerű működésekre is bontható, érdemes egy felhőszolgáltató serverless szolgáltatásait igénybe venni ehhez. A projekt a fent megfogalmazott inspirációval és célokkal indult el.

A projekt egy webes közösségi tér, a Reddit ötleteiből, látszólagos struktúrájából táplálkozik. A felhasználók Google felhasználói fiókjuk segítségével tudnak bejelentkezni. Lehetőségük van felfedezni csatornákat, csatlakozhatnak ezekhez, valamint böngészhetik a csatornák alatt posztokat, ezekhez akár kommentet is fűzhetnek. A webalkalmazás használata során egyes interakciókról böngészőben megjelenő értesítéseket kap a felhasználó. Az alkalmazás igyekszik magas fokú esztétikai és felhasználói élményt nyújtani, mobile first UI tervezési szemlélettel.

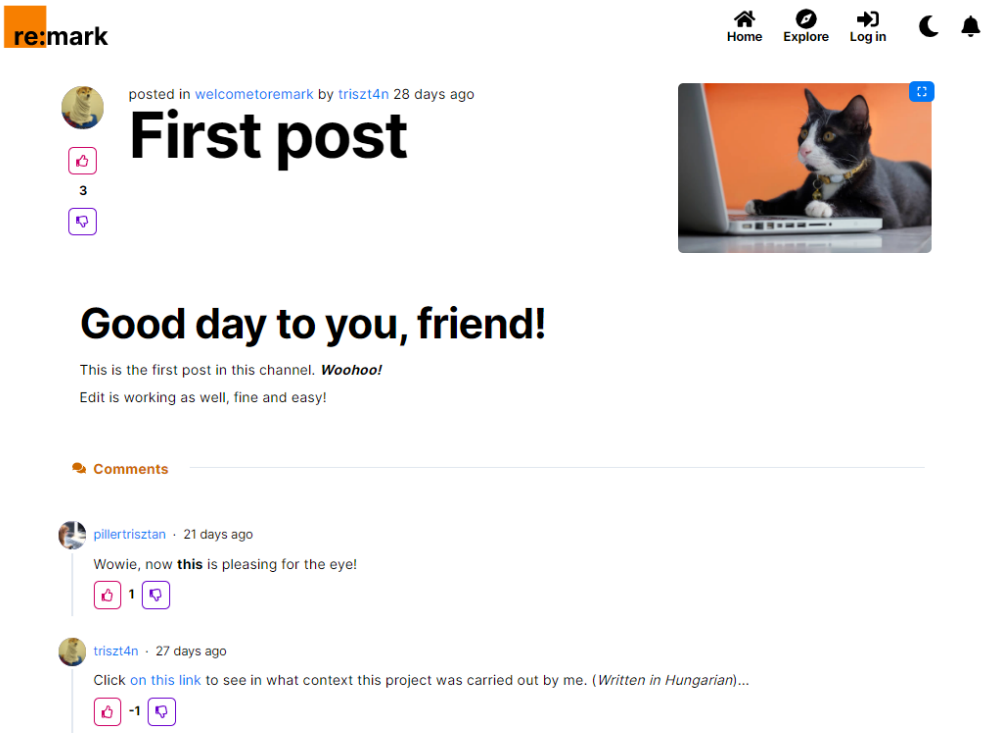
A projekt monorepója publikus itt: https://github.com/triszt4n/remark

Valamint az élő oldal tesztelhető itt: https://remark.triszt4n.xyz/

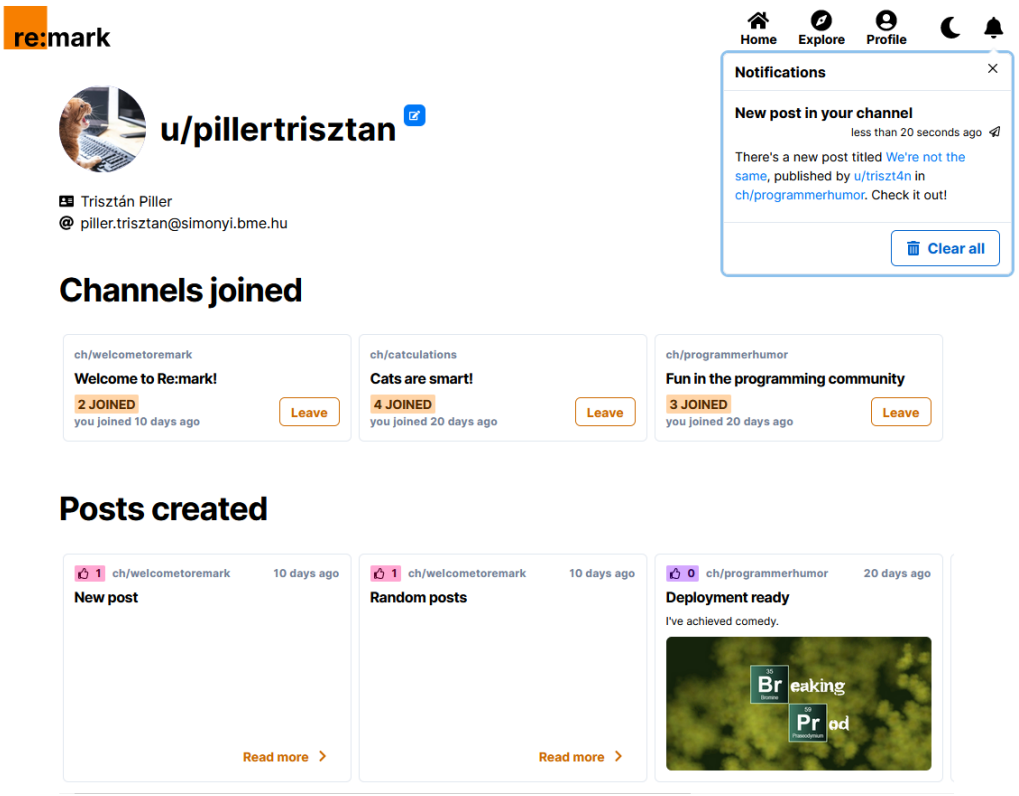
1.1. UI pillanatképek



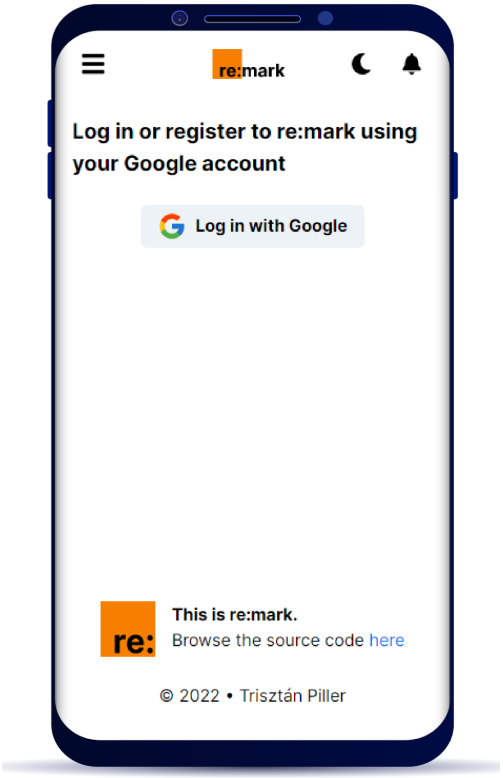
Ábra: Egy csatorna és posztjai



Ábra: Egy poszt és kommentjei



Ábra: Egy felhasználó és hozzákapcsolt adatok, valamint az értesítés popup



Ábra: Bejelentkezési oldal mobil nézetben

1.2. Mikroszolgáltatások

A monolit többrétegű rendszerek hátránya, hogy minden réteg - legyen az egy átlagos esetben a prezentációs, üzleti logikai és adatelérési réteg - funkcionalitása egy közös kódbázisban kerül kialakításra. Ez a centralizáltság jó lehet kisebb alkalmazásoknál, viszont kiterjedt rendszereknél bevett módszer, hogy az architektúrát úgy konstruálják, hogy már jól elhatárolt alrendszerekre osztják a teljes funkcióhalmazt.

Ezt a fajta architektúrát mikroszolgáltatásoknak hívjuk. A mikroszolgáltatások szoftver architektúra 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.

A mikroszolgáltatások világa szorosan kapcsolódik a felhőszolgáltatások világához. Több jobban ismert felhőszolgáltató (Microsoft Azure, Amazon Web Services, IBM és Google Cloud Provider) tud ajánlani számunkra olyan lehetőségeket a felhőjükben, amellyel saját mikroszolgáltatás alapú rendszerünket tudjuk a szolgáltatónál futtatni, telepíteni, felügyelni.

1.3. Serverless

A mikroszolgáltatás architektúrájú alkalmazásokat lehetőségünk van serverless alapon is futtatni a felhőszolgáltatónál, ha az ad lehetőséget erre. Ez nem jelenti szó szerint, hogy nincsenek a háttérben szerverek futtatva, csupán nekünk nem kell állapottal rendelkező szerverként tekinteni a saját alrendszereinkre, több figyelem marad a fejlesztésre. Serverless PaaS szolgáltatások használatával gyakorlatilag üzemeltetési feladatok nélkül tudunk egy elosztott, hibatűrő, automatikusan skálázódó, nagy rendelkezésre állású rendszer költséghatékonyan üzemeltetni.

A projektnek megfelelően a Microsoft Azure felhőszolgáltatónál is fellelhető ilyenfajta PaaS (Platform-as-a-Service) megoldások közé tartoznak: Cosmos DB, Static Web Apps, Azure Functions, Service Bus, API Management, illetve a Blob Storage. Ezek közül kiemelendő az Azure Functions, amely a FaaS (Function-as-a-Service) megoldások egyike, és lehetőséget ad üzleti logika függvényszerű megírására és annak ilyenfajta futtatására is: inputok feldolgozása, outputok kiadása.

Az ilyen felhőszolgáltatások alkalmazásának hátránya lehet a vendor lock-in, amelynek a lényege, hogy amikor kialakítjuk a rendszerünk, alapjaiban építünk egy felhőszolgáltató PaaS megoldásaira, és ha esetleg áttérnénk másik szolgáltatóhoz, akkor az nagyon sok overheaddel jár. Másik gondot a cold start tudja okozni.

2. Követelmények

A **felhasználó** be tud lépni Google accounttal, ha még nincs fiókja a re:markon, akkor létrejön a felhasználó az adatbázisban. A felhasználó módosításkor tud profiljához képet rendelni. A weboldalon egy erre kitalált oldalon **értesítéseket** kap a felhasználó

mindenféle eseménnyel kapcsolatosan, emailben nem. A felhasználó képes saját **csatornát** alapítani. Minden új posztról értesítést kap a készítő. A felhasználó képes csatlakozni meglévő csatornákhoz. Ezáltal ha új poszt jön a csatornába, akkor arról kap értesítést. A csatornán lehet **posztokat** közzétenni abban az esetben, ha a felhasználó tagja a látogatott csatornának. A készítő értesítést kap minden hozzászólásról. A felhasználó adhat képet a posztnak. Posztokon hagyhatóak **hozzászólások**, ehhez viszont nem kell csatornatagnak lenni. A vastagon szedett szavak a lényegi entitások, amelyekre adottak a CRUD funkciók.

3. Technológiák

3.1. Kliens oldali programozási technológiák

- **React.js**
  - A kliensalkalmazás Create-React-App segítségével lett generálva
  - React Hook Forms: form összeállításra
  - React Query: belső állapot menedzsmentjére
  - Chakra UI: React UI komponens library
  - Axios: HTTP kérések összeállítására
  - React Router: belső routingra
- **Yarn** package manager (Berry verzió)
- **TypeScript**
- **ESLint + Prettier**
- *Ajánlott text editor*: **VSCode**

3.2. Szerver oldali programozási technológiák

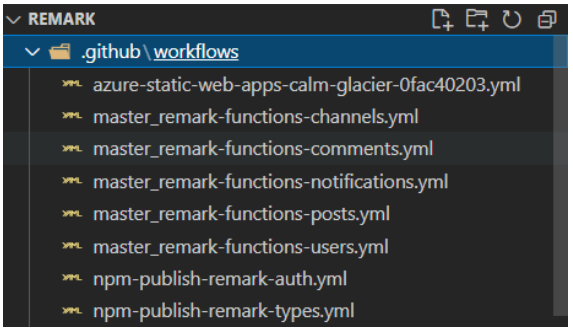
- **Node.js**
  - Service Bus Node.js SDK
  - Blob Storage Node.js SDK
  - Cosmos DB Node.js SDK
  - Axios
- **TypeScript**
- **NPM**
  - @triszt4n/remark-auth: autentikáció megvalósítása szerver oldalon
  - @triszt4n/remark-types: projekteken átnyúló típusok
- *Ajánlott text editor:* **VSCode** (Azure integráció)

3.3. Kommunikációhoz használt technológiák

- A szerver oldali HTTP funkciók alapvetően **REST**-en keresztül kommunikálnak.

- A timestampek **unix timestampek**ként kerülnek tárolásra, kerülnek passzolásra frontendre, a frontend felelőssége ebből olvasható outputot készíteni.
- A felhasználókezelés során **saját JWT generálás**i flow került kifejlesztésre.
- SignalR esetén HTTP-n (Negotiate nevű Azure Function-ön) keresztül történik a kapcsolatépítés, majd ugyanezen PaaS szolgáltatásból **websocket**en jönnek az értesítések.

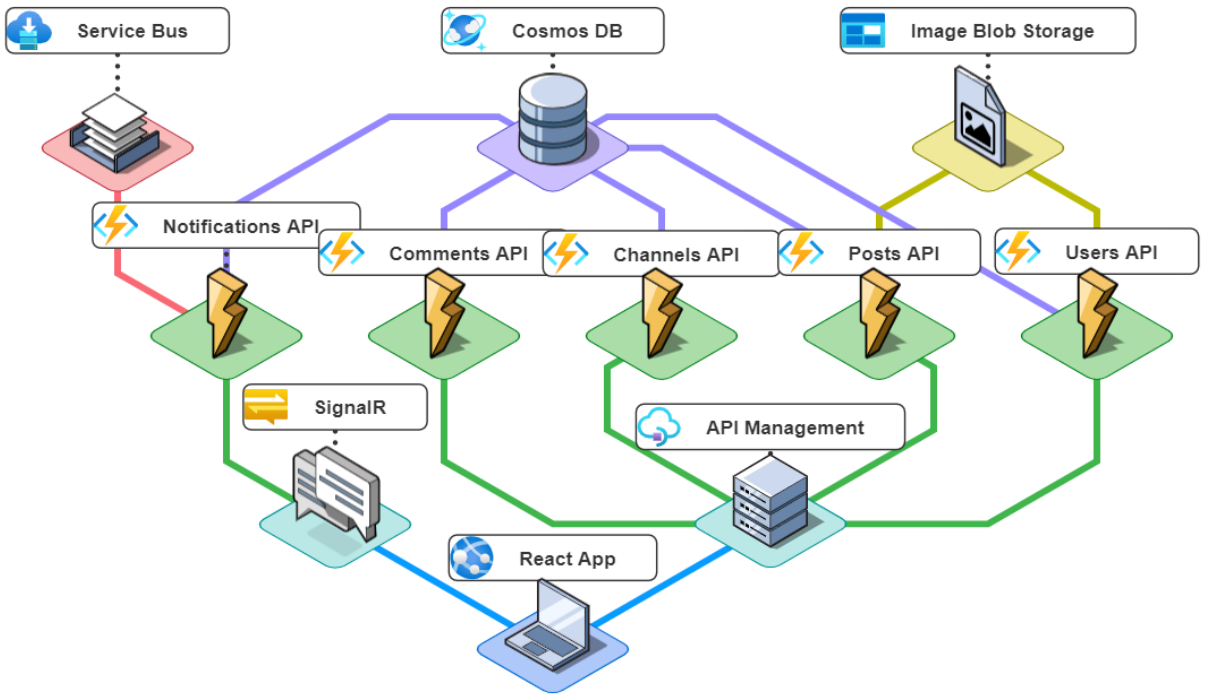
3.4. Telepítés technológiája



- **GitHub Actions**: az Azure Function App-ok, a Static Web App és az NPM csomagok deploymentje egy-egy Github Action workflow file által kerül kódban megvalósításra, a GitHub CI-ja által kerülnek fel az Azure szervereire a megfelelő forráskódok a futtatáshoz.
- Minden további PaaS megoldás az Azure Portalon kerül konfigurálásra, telepítésre.

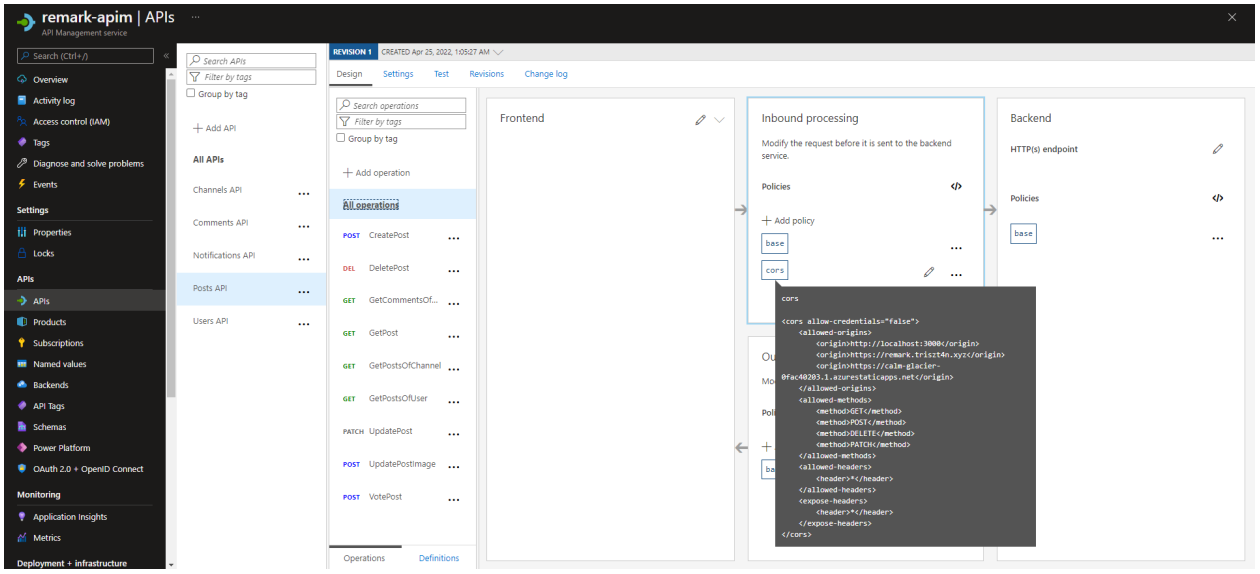
4. Architektúra

4.1. Technikai architektúra



A felhasznált Azure PaaS megoldások

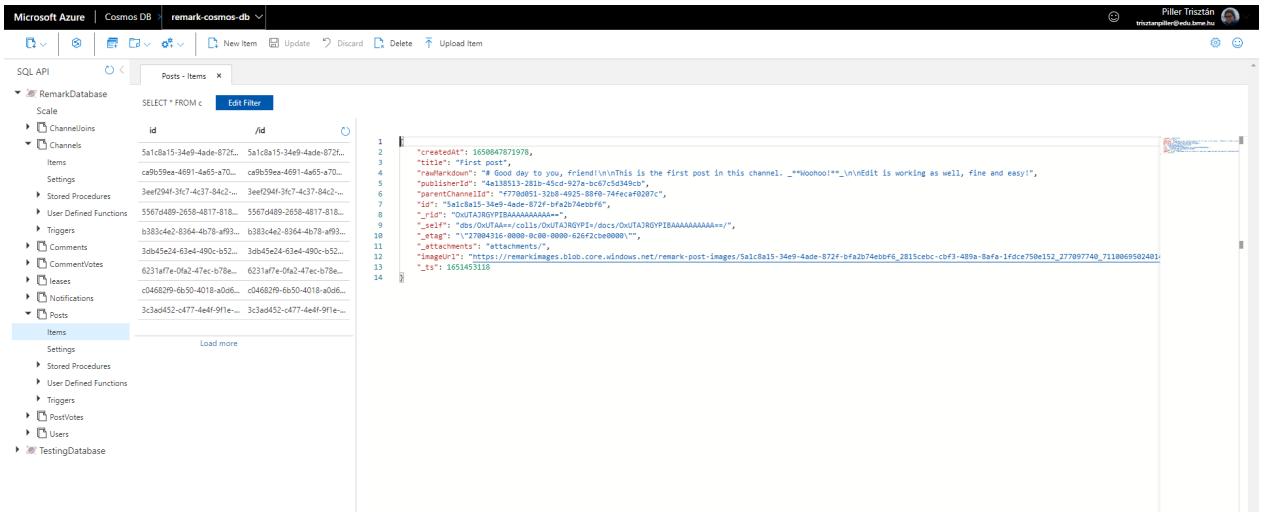
A projekt minden Azure-ból felhasznált szolgáltatása serverless alapú, az alap CRUD kérések mind az **API Management** egységen keresztül mennek, ez a gateway a kérések között, megvalósítja a **CORS** beállításokat és a megfelelő endpoint elérési útvonalakat.



Beállított CORS szabályok az API Management egységben

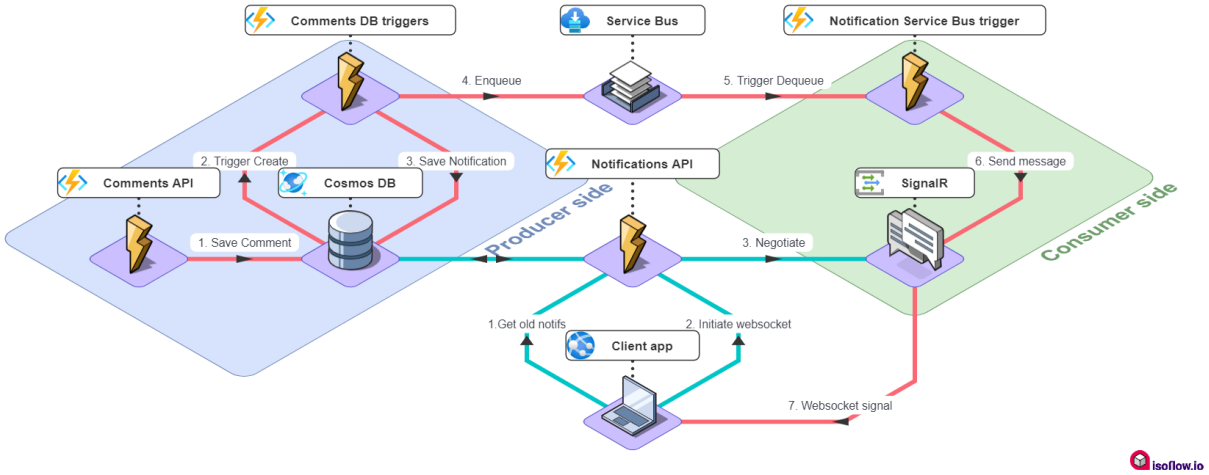
Az egyes Channels API, Posts API stb. nevezetű **Azure Function App**ok mögött több operáló függvény (Azure Function) is található, ezek valósítják meg az egyes adatbázis entitásokkal és képfájlokkal való CRUD funkciókat. A Function Appok API kulccsal védettek, amelyet csupán az API Management ismerhet (az API Management nem igényel ehhez extra konfigurációt, egy Azure resource-on belül alapból beköti az API kulcsokat a platform).

Az adatbázist érdemes Azure Portalról menedzselni, majd a https://cosmos.azure.com/ oldalon betekinteni az entitásokba. Az adatbázis **Cosmos DB**-n belül Mongo alapú (dokumentum alapú adatbázis), azonban minden támogatott backend nyelvre van SDK, amely tulajdonképpen az ODM-et biztosítja az adatmanipulációhoz és -lekéréshez.



Cosmos portálon való adatfeltárás

4.2. Értesítéskezelés



Flow a létrejött komment és a generált értesítés kiküldésére

Producer-Consumer mintát használ, ahol a producer (termelő) oldal készíti a üzeneteket, helyezi be ezeket az üzenetsorba, a consumer (fogyasztó) oldal pedig ezeket fogadja aszinkron módon, és átalakítja a felhasználó számára fogadható formátumba.

Producer oldalon ha egy egyszerű HTTP trigger Function (pl. Comments API-ban a CreateComment HTTP trigger) lement egy entitást az adatbázisba, az újonnan létrejövő entitás kifejt egy változást a Comments adatbázis konténer Change feedjében, amelyre köthetünk Cosmos Triggert: ez itt most a Comments DB Trigger. Ez a trigger Function fog besorolni egy üzenetet a Service Bus által menedzselt üzenetsorba (erre kerül használatba a Service Bus SDK).

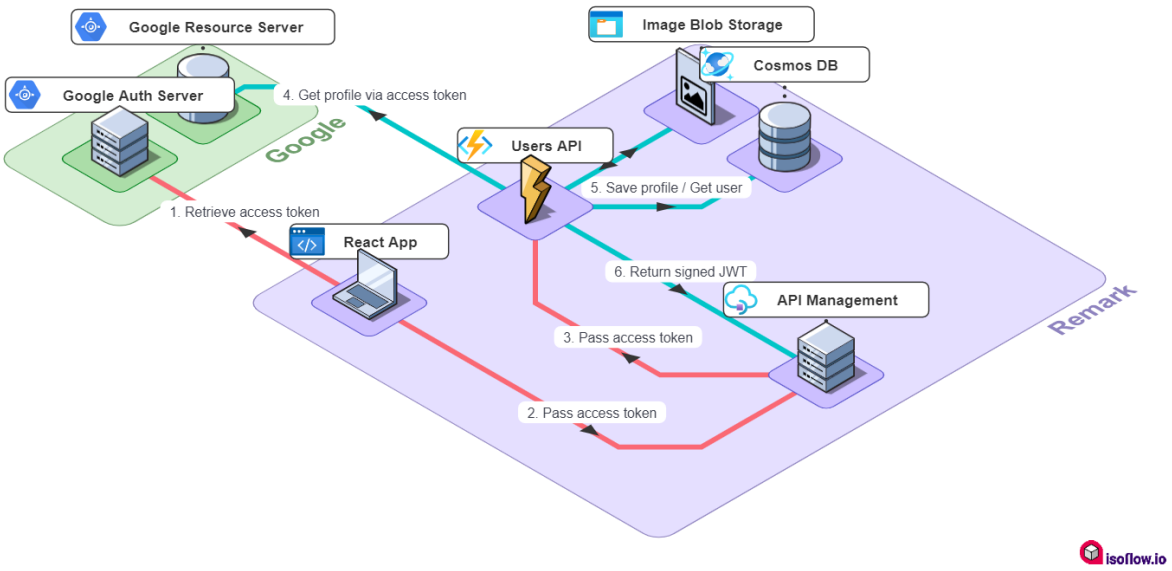
A **Service Bus** teljesen autonóm kezeli az üzenetsort, jól definiált API-val rendelkezik, és roppant sok másik PaaS szolgáltatást lehet hozzákötni (Event Grid, Event Hub, Web PubSub). A legjobb megoldás itt a consumer oldalról egy Service Bus trigger típusú Azure Function bekötése volt, amely a Cosmos Triggerhez hasonlóan működik, azonban ez az üzenetsorba való illesztésnél éled fel, példányosodik, és az üzenetsorból veszi az üzeneteket, alakítja megfelelő alakúra a SignalR Service számára.

Mivel az Azure Function-ökre köthető input és output stream, a leghatékonyabb megoldás rögtön a **SignalR Service**-t teljes egészében az előző Function output bindingjára kötni.

A SignalR kódját mi magunk nem tudjuk befolyásolni, csak konfigurálni, de az alapbeállításokkal is tökéletesen működik a szolgáltatás, kapja az input streamből az előző Function outputját, és az integráció így biztosított. A SignalR pedig mihelyst a frontend alkalmazás a Negotiate operáción keresztül megszerezte magának a websocket kapcsolatot, azon keresztül küldi le a klienseknek a megfelelő üzeneteket - újraformázva értesítésként.

Az értesítés fogadása a React alkalmazásban **@microsoft/signalr** Javascript package segítségével történik, innen jön websocketen keresztül az összes értesítés a felhasználó számára. A korábbi értesítéseket egyszer betöltéskor szerzi meg a kliens.

4.3. Bejelentkeztetés



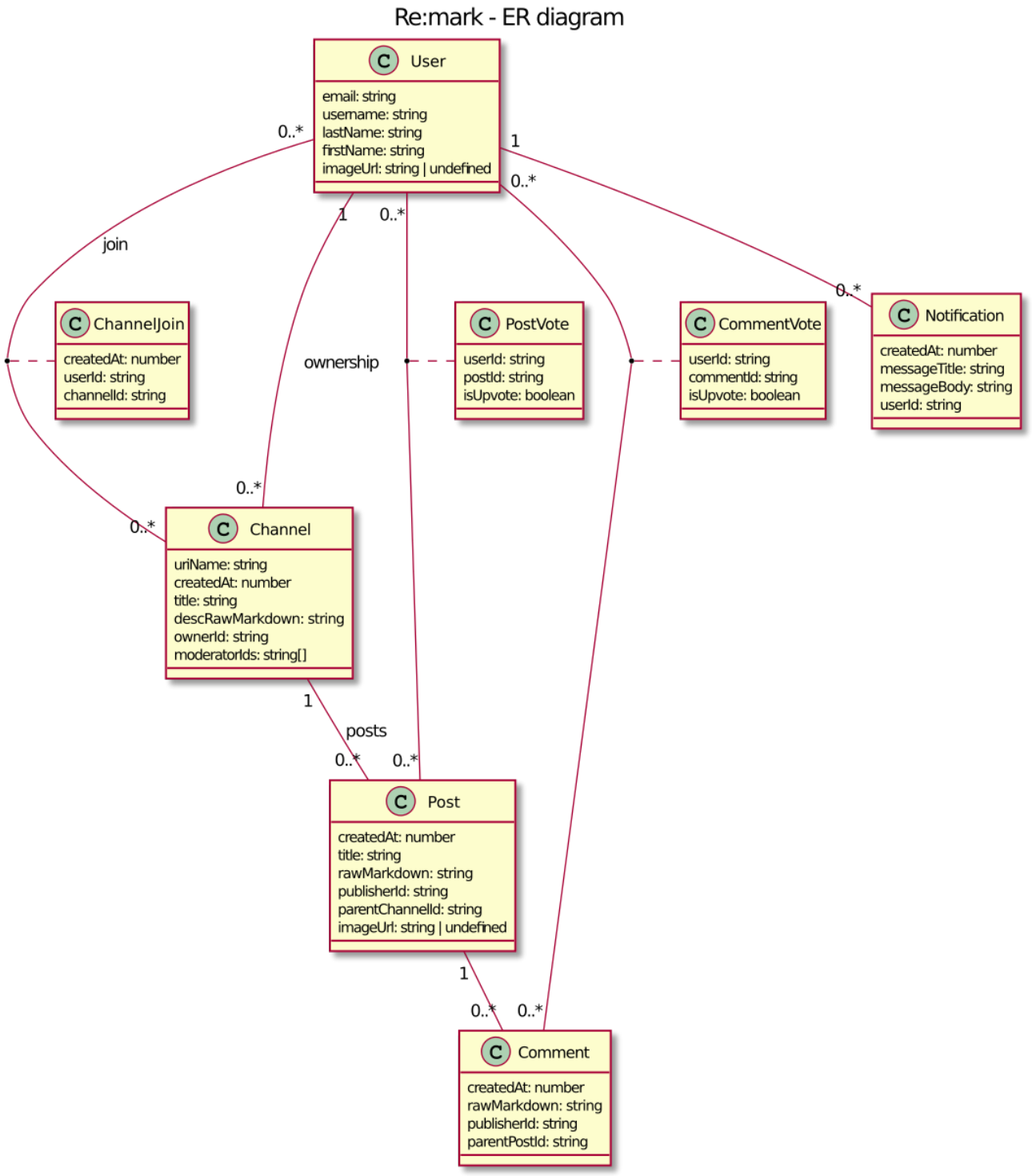
Login flow bemutatása Google autorizációval egybefűzve

Mivel mostanában az átlag felhasználónak van legalább egy **Google fiók**ja, érdemes lehet saját autentikációt kikerülni, ezzel felgyorsítva a bejelentkezés folyamatát, könnyítve a felhasználó feladatát, könnyebb profil adatok beszerzését számunkra.

A profiladatok megszerzése a React alkalmazásban a react-google-login package által biztosított, a profiladatokhoz való hozzáférési token megszerzésével a frontend ezt leküldi a mi **LoginUser Azure Function** operációnknak, ott egy újabb Google felé intézett kéréssel a Resource szerverről megszerezhetőek a profiladatai a hozzáférési token (access token) által autorizált felhasználónak.

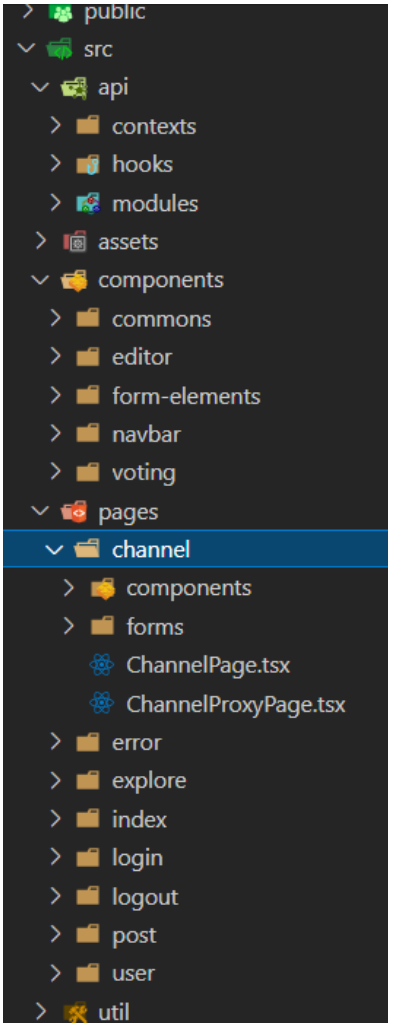
Ezeket (keresztnév, vezetéknév, e-mail cím) tulajdonságokat felhasználói entitásként tároljuk adatbázisunkban, majd még **JWT tokenbe csomagolva** kerül a felhasználó indentitása, így lehet a szerver oldalon autorizálni a megfelelő operációkra, amint az össze további kliens kérés **“Authorization” header**jébe bekerül a letárolt JWT token, és mi tudjuk feloldani csak azt a privát kulcsunkkal. A JWT kliens oldalon cookie-ban tárolódik. Kijelentkezéskor csupán törlődik a cookie-k közül a JWT token.

5. ER diagram


Re:mark - ER diagram
User
email: string
username: string
lastName: string
firstName: string
imageUrl: string | undefined
0..*
1
0..*
1
0..*
join
ChannelJoin
createdAt: number
userId: string
channelId: string
ownership
PostVote
userId: string
postId: string
isUpvote: boolean
CommentVote
userId: string
commentId: string
isUpvote: boolean
Notification
createdAt: number
messageTitle: string
messageBody: string
userId: string
0..*
0..*
Channel
uriName: string
createdAt: number
title: string
descRawMarkdown: string
ownerId: string
moderatorIds: string[]
1
posts
0..*
0..*
Post
createdAt: number
title: string
rawMarkdown: string
publisherId: string
parentChannelId: string
imageUrl: string | undefined
1
0..*
0..*
Comment
createdAt: number
rawMarkdown: string
publisherId: string
parentPostId: string

6. Kódolási részletek

6.1. Kliens kódbázis



A kódbázis alapvető React alkalmazás bázisát követi az **src** mappa alatt. Az API-val való kommunikáció kódja az **api/modules** mappa alatt található, itt található még minden kontextust igénylő komponens is (bejelentkezési és cookie-elfogadási React provider). Az **assets** mappába szorítandó minden képi, hang és szöveges erőforrás, ami beépül az oldalba (kivéve a statikusak, azok mehetnek a public mappába).

A **components** mappában találhatóak a konkrét UI komponensek (React Functional Components). Vannak közös UI komponensek, ezek a commons mappában találhatóak.

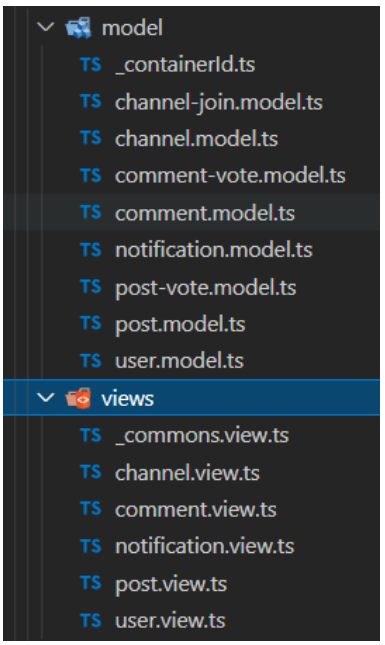
A **pages** alatt találhatóak az egyes oldalak komponensei, ezek pedig még tovább vannak csoportosítva **forms** és **components** (saját kis alkomponensek) mappák alá.

Az **util** mappa tartalmaz minden más Typescript alapú segédfüggvényt, inicializálást és konfigurációt.

Az egyes konkrét useQuery és useMutation (lásd React Query) hook-ok magában a formokat kezelő komponensekben kerülnek használatra.

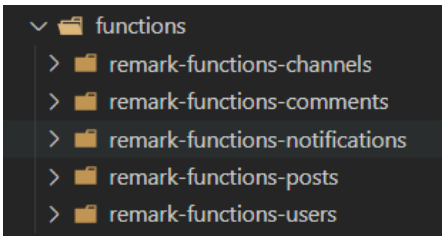
A client mappában **yarn install** paranccsal lehet telepíteni a függőségeket, **yarn start** parancs kiadásával indítható az alkalmazás fejlesztési üzemmódban (http://localhost:3000 címen látogatható lesz). Buildelni **yarn build** paranccsal lehet.

6.2. Remark-types



Fontos, hogy a projektben megkülönböztetendő a View és Model használata: Modelekben kerülnek letárolásra az adatbázis entitások. View formában kapja a backendtől a frontend a feldolgozott entitásokat. Erre főként azért van szükség, mert nincs projekteken átívelő ODM keretrendszer, és a Cosmos DB dokumentum alapú adatbázis, alapból nincs neki sémája, így a backendet kell kényszerítsük séma alkalmazására.

6.3. Szerver oldali kódbázis



Az egyes Function App-ok kódjai a git monorepó functions mappája alatt találhatóak. Az alap felépítése ezen projekteknek az, hogy a model deklaráció, a DB lekérés formalizációja, és külső PaaS (DB, Storage, Service Bus) felé való kapcsolatépítés kódja a lib mappában található. Minden más mappa az egyes Function-ök kódját és felkonfigurálását (JSON) tartalmazza.

6.4. Kaszkádosított törlés

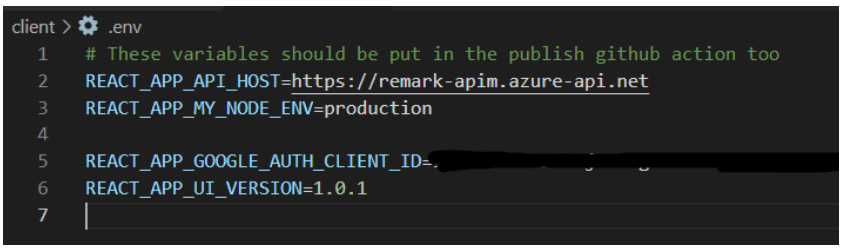
A kaszkádosított törlés Azure Functionök által kerül megvalósításra. Mivel kompozíció alapú kapcsolat van csatorna és posztjai/csatlakozásai, valamint poszt és kommentjei/szavazatai, végül pedig komment és szavazatai között, így az a megoldás került megvalósításra, hogy az egyes erőforrások törléskor magukat megjelölik egy “isDeleted” flaggel, majd pedig erőforrásonként (DeletePostCosmosTrigger, DeleteCommentCosmosTrigger stb.) fut le egy Cosmos DB trigger Azure Function formájában, amely törli magát az entitást és ugyanígy flageli a gyerekeit, amely újabb triggereket lő el.

6.5. Endpointok

Minden endpoint az API Management alatt található. A szolgáltatás nagyszerű jellemzője, hogy az endpointjaink ezen szolgáltatás alá való kötése tisztázza az endpointok használatát, így öndokumentálóvá válik.

6.6. Konfiguráció / Környezeti változók

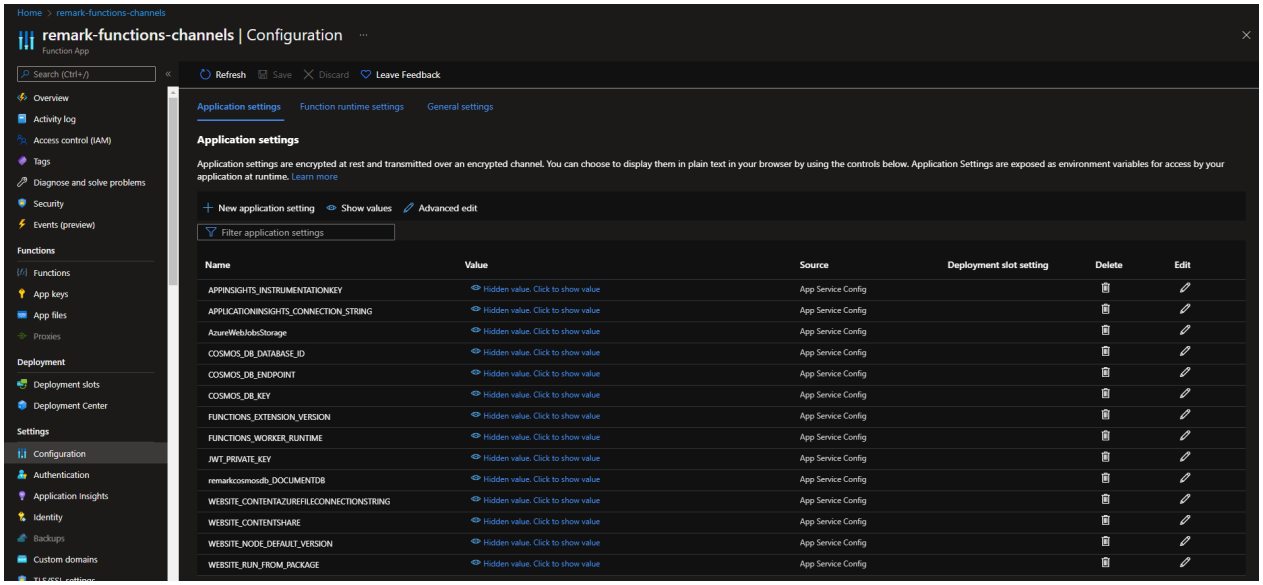
6.6.1. Kliens oldali konfiguráció



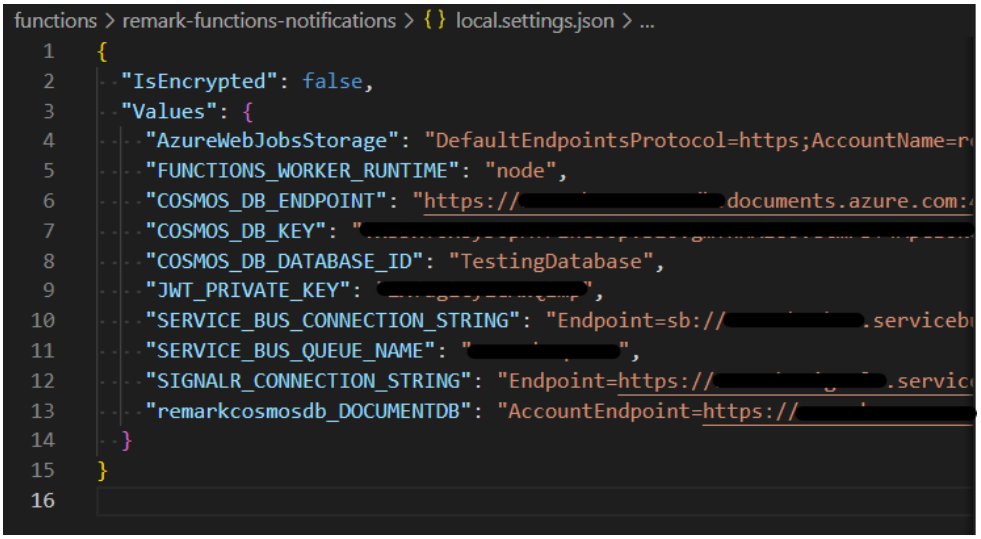
Környezeti változók lokális fejlesztésre való beállítása

Fenti képen is látható egy kitöltése a kliens oldali környezeti változóknak. Szükség van egy Google autentikációs kliens azonosítóra, amelyet a Google OAuth2.0 API dokumentáció meghatároz, hogy szerezhető be. A backend (API Management) URL-jét szükséges még beállítani. További értékek opcionálisak. Azonban ezeket az értékeket a Static Web App Github Action workflow-jában is közölni kell az ‘env’ sornál, ezeket az értékeket érdemes Github Action secretként beállítani a repository beállításai alatt.

6.6.2. Azure Function Appok konfigurációja



Környezeti változók beállítása Channel API Azure Function Appban Azure Portalon



Környezeti változók lokális beállításai Notifications API Azure Function Appban

A környezeti változók tartalmazzák az appok más szolgáltatásokhoz való hozzáféréséhez szükséges URI-kat és API kulcsokat, privát kulcsokat, konténer neveket stb. Ezeket élő használathoz az Azure Portalon kell beállítani. Lokális futtatási beállítások a local.setting.json fájlban módosíthatóak minden Function App kódjának gyökerénél. Nem mindegyik Function App használja az összes változót, azonban összesítve ezek:

- **COSMOS_DB_ENDPOINT**: Cosmos cluster URI-ja query paraméterek nélkül
- **COSMOS_DB_KEY**: az adatbázis eléréséhez szükséges egyedi kulcs
- **COSMOS_DB_DATABASE_ID**: a megfelelő adatbázis kiválasztása
- **JWT_PRIVATE_KEY**: a JWT-k aláírásához használt privát kulcs
- **remarkcosmosdb_DOCUMENTDB**: Cosmos Triggerek által használt összetétele a legfelső három beállításnak

- **SERVICE_BUS_CONNECTION_STRING**: a Service Bushoz való hozzáférés teljes kapcsolódási stringje
- **SERVICE_BUS_QUEUE_NAME**: az app által használt Service Busbeli üzenetsor neve
- **SIGNALR_CONNECTION_STRING**: a SignalR-hez való hozzáférés teljes kapcsolódási stringje
- **STORAGE_ACCOUNT_NAME**: A képeket tároló Blog Storage account neve
- **STORAGE_ACCOUNT_KEY**: A képeket tároló Blob Storage API kulcsa

6.7. Értesítések

6.7.1. Események, amikre már van notification készítés:

- Moderátorként lettél hozzáadva egy csatornához.
  üzenetsorba teszi: **AddModeratorToChannel** (HTTP trigger)
- Csatlakoztak a csatornádhoz
  üzenetsorba teszi: **CreateChannelJoinCosmosTrigger** (Cosmos trigger)
- Elvették a moderátorjogod a csatornából
  üzenetsorba teszi: **RemoveModeratorFromChannel** (HTTP trigger)
- Kommenteltek a posztod alá
  üzenetsorba teszi: **CreateCommentCosmosTrigger** (Cosmos trigger)
- Szavazatot kapott a kommented
  üzenetsorba teszi: **VoteCommentCosmosTrigger** (Cosmos trigger)
- Új poszt készült a csatornában, amihez csatlakozva/moderátor/tulajdonos vagy
  üzenetsorba teszi: **CreatePostCosmosTrigger** (Cosmos trigger)
- Szavazatot kapott a posztod
  üzenetsorba teszi: **VotePostCosmosTrigger** (Cosmos trigger)
- Törölték a csatornát, amelynek tagja voltál
  üzenetsorba teszi: **DeleteChannelCosmosTrigger** (Cosmos trigger)

6.7.2. Események, amikre még nincs notification készítés:

- Törölték a posztodat (ki törölte?)
- Törölték a kommentedet (ki törölte?)
- Vote-ok aggregálva posztokra ("A posztod elért 10 upvote-ot stb.")
- Vote-ok aggregálva kommentekre ("A kommented elért 10 upvote-ot stb.")

6.8. Képfeltöltés

A feltöltés a Blob Storage SDK-ján keresztül történik. Mivel a képet több darabban kapja meg a megfelelő Function, így Node.js alatt parse-multipart NPM csomag Parse függvénye fogja kezelni a bodyban Bufferként kapott bájtfolyamot.

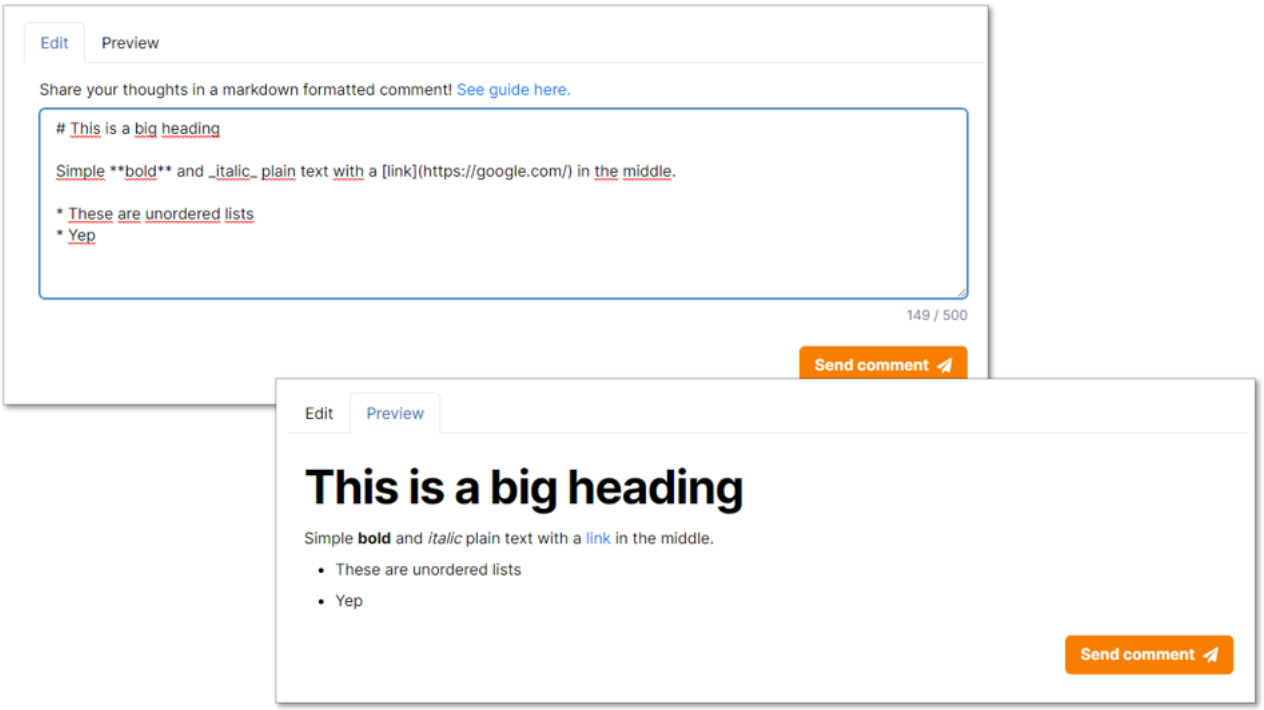
Böngésző kliens oldalról alap Javascript FormData interfész alatt kell kerüljön felküldésre a kép az inputból.

Segítség ezeken a linkeken:

- https://docs.microsoft.com/en-us/azure/developer/javascript/how-to/with-web-app/azure-function-file-upload
- https://github.com/Azure-Samples/js-e2e-browser-file-upload-storage-blob/blob/57eb885fdd29deec7629fb0d9206384dad86cb5c/src/azure-storage-blob.ts#L21

6.9. RemarkEditor

A kliens alkalmazásban biztosítva van, hogy a nyers markdown formázott kommentet, poszt vagy csatorna leírást előrenderelje a weboldal, és lássa a felhasználó, hogy fog kinézni majd a végső állapotában a weboldalon. Ezt a RemarkEditor React UI komponens biztosítja integrált React Hook Forms támogatással és Chakra UI-ra való rendereléssel.



A RemarkEditor szerkesztési és előnézeti módra váltva

7. Kitekintés

7.1. Middlewares

Az autorizáció és autentikáció jelenleg a @triszt4n/remark-auth NPM csomag által kerül megvalósításra, azonban nem garantálja azt, hogy egységesen kezelhetővé váljon az endpointok védelme, így érdemes lehet bevezetni middleware struktúrát az egyes Function Appokban (express.js vagy fastify könyvtárak segítségével Node.js alatt), avagy az API Management JWT validációs konfigurációját használni ennek megoldására.

7.2. Tesztelés

Tesztelés egyelőre hiányzik mind frontend, mind backend oldalon. Ezeknek hiánya enyhítendő a megfelelő működés biztosítása érdekében.

7.3. Features

Az alkalmazás továbbfejleszthető UI bugok javításával, további értesítéskezelési mechanizmusok bevezetésével (email egyes interakciókról, egyes értesítéstípusok kikapcsolása felhasználónként, értesítések szűrése frontenden), valamint a képfeltöltés kiterjeszthető csatorna bannerekre stb. Bevezetésre kerülhet a felhasználók követésének lehetősége más felhasználók által.

7.4. Ajánlórendszer (epic)

További nagyobb léptékű fejlesztés lehet egy ajánlórendszer kifejlesztése a relevanciapontok alapján. Így a felhasználóknak adható egy személyre szabott feed, amely frissül óránként. Ajánlható a felhasználóra szabva egy-egy csatorna, poszt, akár felhasználó is, akinek hasonló ízlése van.