



TANSZÉKVEZETŐ

SZAKDOLGOZAT FELADAT

Fekete Sámuel

Mérnök-informatikus hallgató részére

Versenyértékelő serverless webalkalmazás fejlesztése Azure alapokon

A Magyar Tájékozódási Futó Szövetség (továbbiakban a Szövetség, vagy MTFSZ) felügyelete alatt különböző amatőr rendezői csapatok felelősek Magyarország tájfutóversenyeinek szervezéséért. A szövetség már régóta szeretne egy portált, ahol a versenyek résztvevői különböző szempontok alapján értékelhetik a versenyeket, ezzel visszajelzést nyújtva a rendezői csapatoknak. A hallgató feladata egy ilyen webalkalmazás elkészítése a Szövetség iránymutatásai alapján.

A tájfutóversenyek tipikusan hétfőként vannak, így az oldal terheltsége várhatóan vasárnap esténként lenne magasabb, míg hétközben nagyon alacsony lenne. Ahhoz, hogy alkalmazás üzemeltetése hosszútávon költséghatékony maradjon, a hallgatónak törekednie kell serverless szolgáltatások használatára. A jelölt feladata a serverless megoldások felderítése, értékelése és hatékonyságuk elemzése is.

Az alkalmazásnak a versenyek adatait egy külső rendszerből, az MTFSZ adatbázisából kell lekérnie egy privát REST API-n keresztül. A felhasználók azonosítása az MTFSZ meglévő autentikációs rendszerén keresztül kell történjen. A versenyek értékelési szempontjai nem fixek, ezeknek a változtatására létre kell hozni egy adminisztrációs felületet, ahol a Szövetség megbízottjai elkészíthetik és beállíthatják a szempontokat.

A hallgató feladatának a következőkre kell kiterjednie:

- Ismerje meg a serverless alapelveket és Azure-specifikus megvalósításait.
- Tervezze meg az alkalmazás architektúráját Azure serverless megoldások felhasználásával.
- Implementálja a megtervezett alkalmazást a fentebb ismertetett funkciókkal.
- Értékelje a felhasznált serverless szolgáltatásokat különféle mérnöki szempontok szerint, például költséghatékonyság, rugalmasság, teljesítmény – reakcióidő.

Tanszéki konzulens: Simon Gábor, ügyvivő szakértő, BME-AUT

Budapest, 2023. szeptember 24.

Dr. Charaf Hassan
egyetemi tanár
tanszékvezető





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

Fekete Sámuel

**VERSENYÉRTÉKELŐ
SERVERLESS WEBALKALMAZÁS
FEJLESZTÉSE AZURE
ALAPOKON**

KONZULENS

Simon Gábor

BUDAPEST, 2023

Tartalomjegyzék

Összefoglaló	6
Abstract.....	7
1 Bevezetés	8
1.1 Az alkalmazás és a projekt háttere.....	8
1.1.1 A tájfutásról és a tájfutó versenyekről	9
1.1.2 Specifikáció	9
1.1.3 Előzetes tudásom, tapasztalataim	10
1.1.4 Az MTFSZ létező rendszerei	11
1.2 Serverless	11
2 Felhasznált technológiák	13
2.1 TypeScript.....	13
2.2 Azure.....	14
2.2.1 Azure Functions	14
2.2.2 Azure API Management	16
2.2.3 Azure SQL	16
2.2.4 Azure Static Web Apps.....	16
2.2.5 Application Insights	17
2.3 Backend	17
2.3.1 TypeORM	17
2.3.2 Class-validator, class-transformer	18
2.3.3 Redis	19
2.4 Frontend	19
2.4.1 React	19
2.4.2 Chakra UI.....	20
2.4.3 React Hook Forms	21
2.4.4 React Query	21
2.5 Nx monorepo	21
3 Tervezés	23
4 Megvalósítás	27
4.1 Nx Monorepo	27
4.2 Backend	28

4.2.1 Kapcsolat az adatbázissal	29
4.2.2 Autentikáció	31
4.2.3 Jogosultságkezelés	34
4.2.4 Validáció	35
4.2.5 Hibakezelés	36
4.2.6 Naplózás	37
4.2.7 Függvények felépítése	37
4.2.8 Gyorsítótár.....	38
4.3 Frontend	38
4.3.1 UI/UX.....	40
4.3.2 Értékelő oldal	40
4.3.3 Betöltés gyorsítótárból	41
4.4 CI/CD	42
5 Az éles teszt	43
5.1 Előzetes teljesítménytesztek.....	43
5.2 Teljesítmény, válaszidő	45
5.3 Naplózott hibák	46
5.4 Beérkezett adatok	47
5.5 Visszajelzések	48
5.6 Költségek.....	49
6 Serverless technológiák értékelése	51
6.1 Azure Functions	51
6.1.1 Fejlesztői élmény.....	51
6.1.2 Üzemeltetés, telepítés.....	52
6.1.3 Elérhető irodalom.....	52
6.1.4 Cold-start.....	53
6.1.5 Költségek.....	53
6.2 Azure SQL.....	54
6.3 Összevetés egy virtuális szerver bérletével	56
7 A projekt jövője.....	57
8 Konklúzió	58
Irodalomjegyzék.....	59

HALLGATÓI NYILATKOZAT

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

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

Kelt: Budapest, 2023. 12. 03.

.....
Fekete Sámuel

Összefoglaló

A Magyar Tájékoztató Futó Szövetség felkért, hogy készítsek számukra egy olyan a weboldalt, ahol a tájfutók különböző, dinamikusan változó szempontok alapján értékelhetik a tájfutó versenyeket. Ennek én a Microsoft Azure serverless szolgáltatásaival és a React frontend könyvtárral álltam neki. Azért választottam serverless technológiákat, mert jelentősen ingadozó forgalmat vártam az oldaltól, illetve szerettem volna jobban megismerni őket. A backendem egy TypeScriptben írt Azure Functions projekt, melyben HTTP típusú serverless függvények alkotnak egy REST API-t, amit még pár időzített függvény egészít ki. A rendszerem kapcsolódik a Szövetség már létező API-jához és bejelentkeztető rendszeréhez is. Adatbázisnak egy szintén serverless elveken működő Azure SQL adatbázist választottam. Mivel ennek a válaszüzeje hosszú inaktivitás után rendkívül hosszú volt, egy Redis gyorsítótárat is beleépítettem a rendszerbe, ahonnan a legfontosabb adatok elérhetőek akkor is, amikor az adatbázisból nem.

A dolgozatomban először bemutatom a felhasznált technológiákat, majd hogy milyen tervezési lépéseket hajtottam végre. Ezután részletesen leírom, hogy az alkalmazásom főbb komponensei hogyan készültek, milyen nehézségek és döntési pontok jöttek fel fejlesztés közben. Az alkalmazást egy hét napos teszt keretében körülbelül 100 felhasználó próbálta ki. A dolgozat következő részében az ez alatt az idő alatt gyűjtött adatok és tapasztalataim alapján értékelem a serverless technológiákat és megvizsgálom, hogy mennyire illettek ehhez a projekthez.

Végül szót ejtek a hosszú távú terveimről a projekttel kapcsolatban, illetve mivel az elemzésben arra jutottam, hogy a serverless modellhez kevésbé illik a projekt, felhozok pár lehetséges továbbfejlesztési lehetőséget is.

Abstract

The Hungarian Orienteering Federation asked me to create a website for them, where orienteers can rate orienteering races based on different, dynamically changing criteria. I achieved this using serverless services of Microsoft Azure and the React frontend library. I chose serverless technologies because I expected significantly fluctuating traffic to the site, and I wanted to get to know them better. My backend is an Azure Functions project written in TypeScript, where HTTP serverless functions form a REST API, complemented by a few timer functions. My system is also connected to the existing API and authentication system of the Federation. For data storage, I chose a serverless Azure SQL database. Since its response time was extremely long after long periods of inactivity, I included a Redis cache in the system, from which the most important data can be served even when the database is unavailable.

In my thesis, I will first describe the technologies used and then the planning steps I took. I will then describe in detail how the main components of my application were built and what difficulties and decision points arose during the development. The application was tested by about 100 users in a seven-day trial. In the next part of the thesis, I will evaluate serverless technologies based on the data and experiences I gathered during this time and examine how well they fit this project.

Finally, I will talk about my long-term plans for the project and, since the analysis has led me to conclude that the serverless model is a poor fit, I will also list some possible further improvements.

1 Bevezetés

Szakdolgozat témámnak serverless webalkalmazás fejlesztést választottam. Már egyetemi tanulmányaim előtt is foglalkoztam webfejlesztéssel, egyetem mellett pedig a Simonyi Károly Szakkollégium webfejlesztő körében (Kir-Dev¹) tanultam nagyon sokat ebben a témában. Itt azonban mindig hagyományos alkalmazásokat fejlesztettünk, viszont én sokat hallottam a serverless technológiákról és előnyeiről, és úgy gondoltam, ez a tárgy egy remek lehetőség arra, hogy első kézből kipróbáljam, milyen is ilyen módon webalkalmazást fejleszteni.

1.1 Az alkalmazás és a projekt háttere

Körülbelül tízéves korom óta tájfutok, és habár egyetem mellett már sokkal kevesebbet edzek, még mindig időnként járok versenyekre és követem a híreket. Az első általam készített hobbi webalkalmazás témája is a tájfutáshoz kapcsolódott, egy olyan alkalmazást készítettem, ahol a felhasználók tájfutó versenyeket értékelhettek. Ezt az alkalmazást Pythonban írtam, egy Flask² nevű keretrendszerrel használva, és ma visszanezve már látom, hogy nagyon súlyos tervezési és implementációs hibákat vétettem. Ettől függetlenül az alkalmazás többnyire működött és megtetszett a Magyar Tájékozódási Futó Szövetségnek³ (továbbiakban MTFSZ vagy a Szövetség) is, akik felkértek engem, hogy fejlesszem ezt tovább az ő igényeik szerint. Az említett hibák miatt én úgy gondoltam, hogy mindenképpen nulláról kezdeném a fejlesztést, hiszen rengeteget tanultam az eredeti alkalmazás fejlesztés óta. A Szövetség, mint nonprofit szervezet, rendelkezik évi 3500 dolláros Azure kredittel⁴, amit felajánlottak nekem a fejlesztéshez. Így minden összeállt ahhoz, hogy nekiálljak a projektnek e tárgy keretében: van egy adott alkalmazás-ötlet, ami egészen összetett, egy vágy bennem, hogy megismerjek egy új technológiát és még keretem is, amivel Azure-ban garázdálkodhatok.

¹ <https://kir-dev.hu/>

² <https://pythonbasics.org/what-is-flask-python/>

³ <https://www.tajfutas.hu/mtfsz>

⁴ <https://www.microsoft.com/en-us/nonprofits/azure>

1.1.1 A tájfutásról és a tájfutó versenyekről

A tájfutás (hivatalos nevén tájékozódási futás) egy különleges és Magyarországon egészen kevéssé ismert sport. Eredeti verziója erdőben zajlik, ahol a versenyzőket egyenként indítják. Minden versenyző kap egy nagyon pontos térképet az erdőről, amin ellenőrző pontok vannak bejelölve, ezeket kell érinteni minél gyorsabban a megfelelő sorrendben. Az, hogy ki milyen útvonalat választ két pont között, teljesen a versenyzőre van bízva. Így ebben a sportban a fizikális kihívás mellett a tájékozódás és útvonalak választása komoly szellemi kihívást is jelent. A sportot ma már nem csak erdőben, hanem városokban is űzik, illetve több különböző versenyforma vált elterjedté. Magyarország legnagyobb tájfutó versenyein általában 1000-1500 induló szokott lenni. A versenyzők életkora nagyon különböző, 8-9 évesektől kezdve rengeteg gyerek űzi sportot, de a 80-85 éves indulók sem ritkák. Természetesen ők nem mind egymás ellen versenyeznek, életkortól és a pálya nehézségétől függően vannak különböző kategóriák.

Magyarországon ezeket a versenyeket többnyire amatőr csapatok rendezik, akiknél természetesen cél az, hogy ne legyen veszteséges a verseny, de elsősorban azért csinálják, mert szeretik a sportot és a közösséget. Ezért gondolta a Szövetség, hogy szükség lenne egy olyan alkalmazásra, ahol minden versenyző értékelheti a versenyeket, ezzel visszajelzést küldve a rendezői csapatoknak, hogy mely területen emelkedtek ki országosan, illetve mely területeken kell még fejlődniük.

1.1.2 Specifikáció

Az alkalmazás elsődleges célja, hogy magyar tájfutók visszajelzést adhassanak a versenyek minőségéről, mégpedig bizonyos szempontok szerint. Egy átlagos felhasználó látja az éppen értékelhető versenyeket, és bejelentkezés után ezek értékelését meg is kezdheti. Az értékelés megkezdésekor beállíthatja, hogy milyen szerepkörben szeretné végezni az értékelést. Négy szerepkör létezik: versenyző, rendező, edző és zsűri. Ezek közül az utolsó kettő csak bizonyos felhasználóknak érhető el. A felhasználó azt is kiválaszthatja, hogy a verseny mely futamait szeretné értékelni. Értékelés közben először a teljes versenyre vonatkozó szempontokat kell kitöltenie, majd a futamokra vonatkozókat, minden futamhoz, amit kiválasztott az értékelés elején. Akkor tudja véglegesíteni az értékelést, ha már minden szempontra adott le értékelést, ezután szerkeszteni már nem lehet. Minden felhasználó látja még a profilján az általa értékelt versenyek listáját és ezek értékelési státuszát.

Az alkalmazás adminisztrátorai számára elérhető egy külön kezelőfelület. Itt lehet például szempontokat létrehozni. Egy szempontról megadható, hogy milyen skálán értékelhető, illetve hogy mely értékelési szerepkörök számára elérhető. Itt állítható be az is, hogy verseny- vagy futamspecifikus-e a szempont, valamint hogy kötelező-e megválaszolni. Az egyes szempontokat kategóriákba is tudják rendezni az adminisztrátorok. Értékelés közben egyszerre egy kategória szempontjai jelennek meg egy oldalon. A kategóriákat pedig szezonokba kell szervezni, ennek van kezdő és befejező dátuma. Szezonok időtartama között nem lehet átfedés, így minden verseny pontosan egy szezonhoz tartozik. Minden versenyt azon kategóriák alapján kell értékelni, melyek a verseny szezonjához tartoznak. Ennek köszönhetően egy adott időtartamon belül minden verseny ugyanazon szempontok alapján lesz értékelve, azonban időnként van lehetőség a szempontok változtatására, de a régi szempontok sem fognak elveszni. Az adminisztrátorok tudnak még felhasználókat kinevezni különböző szerepkörökbe. Itt lehet például zsűri vagy edzői jogot adni egy felhasználónak, aki ezután minden versenyt ebben a szerepkörben is tud értékelni. Illetve itt tudnak adminisztrátor jogot is adni más felhasználóknak.

Mindezt a Szövetség úgy szerette volna, hogy jelenlegi infrastrukturális költségek és üzemeltetési feladatok ne emelkedjenek, hiszen az IT rendszereken dolgozók többsége a szabadidejében üzemelteti azokat. A jelenlegi tárhelye a Szövetségnek PHP programok futtatására alkalmas, ezen kívül jár még nekik, mint nonprofit szervezetnek évi \$3500 kredit a Microsoft Azure felhő szolgáltatónál.

1.1.3 Előzetes tudásom, tapasztalataim

Mint azt már említettem, a projekt megkezdése előtt is sok tapasztalatom volt már hagyományos webfejlesztésben. Magamtól először Python nyelven írtam webalkalmazásokat, majd a szakkollégiumban a TypeScript nyelvet és a Node.js futtatókörnyezetet ismertem meg jobban. React-tel is sokat foglalkoztam, nem csak szakkollégiumi projekteken, hanem egy négyhónapos gyakornoki program keretében az iparban is. Ezek a tapasztalataim természetesen sokat segítettek, amikor ehhez a projekthez választottam technológiákat, próbáltam mindenből a legkényelmesebbet választani, bár a serverless téma helyenként megkötötte a kezem.

Felhő alapú, illetve serverless fejlesztésben viszont gyakorlatilag semmilyen tapasztalatom nem volt. Ötödik félévemben elvégeztem a Felhő alapú szoftverfejlesztés nevű szabadon választható tárgyat az egyetemen. Ezt egy nagyon hasznos tárgynak

véltem, amin keresztül megismertem az Azure számomra releváns szolgáltatásait, azonban gyakorlati tapasztalatot itt is nagyon keveset szereztem.

1.1.4 Az MTFSZ létező rendszerei

A Szövetségnek létezett már több működő informatikai rendszere, melyeket használnom kellett az alkalmazásomban. Ilyen például a Szövetség REST API-ja, ami lényegében egy interfészt ad a Szövetség teljes adatbázisához. Elérhetők ezen keresztül a Szövetségen keresztül rendezett versenyek és azoknak adatai, valamint a nyilvántartásban szereplő egyesületek és versenyzők is. Habár ezen adatok nagyrésze publikus a szövetség más oldalain, ezt az API-t nem használhatja bárki. Az OAuth 2.0 szabvány *client credentials* folyamata szerint csak a megengedett alkalmazások fognak érvényes választ kapni a szervertől. [14] Erről bővebben írok az 4.2.2-es fejezetben.

Ehhez szorosan kapcsolódik a Szövetség Single Sign-On (SSO, Egyszeri bejelentkezés) rendszere, mely szintén az OAuth 2.0-s szabványra épül. A felhasználók szempontjából ez egy bejelentkező felület, ahol az MTFSZ fiókjukba tudnak belépni. Én, a Szövetség fejlesztője tudom ezt a rendszert használni autentikációnak, így nekem ezt nem kell implementálnom, valamint biztos lehetek benne, hogy a bejelentkezett felhasználó tájfutó. Ezen kívül még adatokat is tudok lekérni a felhasználókról a korábban említett REST API-n, így például lesz lehetőségem ellenőrizni, hogy az éppen bejelentkezett felhasználó elindult-e egy bizonyos versenyen.

1.2 Serverless

Sokan félreértik a serverless technológiákat és nem hibáztatom érte őket, hiszen kicsit félrevezető a neve. Nem azt jelenti, hogy az alkalmazás mögött nincs egy szerver, csak azt, hogy annak karbantartása és üzemeltetése nem a fejlesztő feladata. A fejlesztő ilyenkor tipikusan valamelyik nagy felhő szolgáltatótól bérel számítási kapacitást, és a szolgáltató lesz felelős a szerveren futó operációs rendszerért, a szerver biztonságáért és egyéb üzemeltetési feladatokért. Így a fejlesztő teljes mértékben a kód írására fókuszálhat. A kód nem is feltétlen mindig ugyanazon a szerveren fog futni, de az ebből származó nehézségeket is a felhőszolgáltató fogja megoldani.

Másik előnye a serverless technológiáknak az automatikus skálázódás. Ez működhet egyrészt lefelé. Ha van például egy egyórás időszak, amikor egyetlen kérés sem futott be egy alkalmazás szervere felé, akkor a szolgáltató leállít minden futó példányt az alkalmazásból. Természetesen ez pont így működhet felfele is, azaz nagy

terhelés esetén a szolgáltató automatikusan több példányt is telepít a programból, így sokkal több felhasználót tud kiszolgálni. A terhelésfüggő automatikus skálázódás közben természetesen mindig csak annyit fizetünk, amennyi erőforrást éppen használunk, ami bizonyos esetekben nulla is lehet.

Ezzel kapcsolatos hátrány a *cold-start*. Ez azt jelenti, hogy ha hosszabb ideje (tipikusan 30 perc) nem volt használva az alkalmazás, akkor a szolgáltató a szervert, amin az alkalmazás futott, másra használja fel. Így amikor újra jön egy kérés az alkalmazás felé, azt először újra telepíteni és indítani kell, lehet, hogy egy teljesen másik szerveren. Ennek az időtartama sok mindentől függ, de egy 10-20 másodperces válaszidő nem ritka egy *cold-start* esetén. Ez azonban csak az első kérésnél lesz így, az ezt követőek már a megszokott gyorsasággal megtörténnek, ameddig van valamekkora forgalma az oldalnak. [17]

Az egyik legnagyobb ok, amiért serverless technológiákkal kezdtem meg az alkalmazás fejlesztését az egyszerűen az volt, hogy ki akartam próbálni, kíváncsi voltam hogyan is zajlik egy ilyen típusú fejlesztés. Érdekelt, hogy hogyan is néz ki a valóságban az, amikor a fejlesztőnek tényleg csak a kódírássra kell fókuszálnia, és nem az üzemeltetésre.

2 Felhasznált technológiák

Ebben a fejezetben röviden ismertetem a projektben használt szolgáltatásokat, keretrendszereket és könyvtárakat, valamint írok arról is, hogy miért választottam őket.

2.1 TypeScript

A TypeScript (röviden TS) a Microsoft által karbantartott nyílt forráskódú nyelv, mely a jól ismert JavaScript (JS) továbbfejlesztése. A JavaScript egy dinamikusan, de gyengén típusos nyelv, ami azt jeleneti, hogy egy változó típusa futás során megváltozhat, akár úgy is, hogy az a kódban nem volt explicit leírva. Ez nagyszabású projekteknél sok nehézséget hoz. Webes projekteknél gyakran dolgozunk nagyobb objektumokkal, mint például egy HTTP kérés törzse. JavaScriptben minden ilyen objektumnak a típusa egyszerűen *object*, így semmilyen információnk nincs arról, hogy pontosan milyen mezők léteznek az objektumon. Ha feltételezünk egy olyan mezőt, ami nem is létezik, akkor ez a hiba csak futásidőben fog kiderülni és az alkalmazás nem fog megfelelően működni. [12]

A TypeScript ezeket a problémákat hivatott megoldani. A könnyű átállás érdekében érvényes JavaScript kód TypeScriptben is érvényes lesz, azonban sok nyelvi újdonság van a TypeScriptben. Lehet például típust adni változóknak, amik futás közben már nem válhatnak típust. Lehet saját típusokat is definiálni, ezzel tudunk tetszőlegesen nagy objektumokat is leírni.

TypeScriptet magában nem lehet futtatni, hanem JavaScripté kell fordítani. Ebben a lépésben előjönnek azok a hibák, amik a JS esetében csak futás közben, például ha egy objektumnak nem létező mezőjét használtuk. Valamint a modern kódszerkesztő programok a kód írása közben tudnak segítséget nyújtani, hiszen például ismert, hogy egyes függvények milyen típusú paramétereket várnak, vagy hogy milyen mezői vannak egy típusnak.

A TypeScript egyszerre funkcionális és objektum-orientált nyelv is. Az én projektben osztályokat ritkán használtam, de bizonyos keretrendszerekben nagyon gyakoriak. A funkcionalitásnak köszönhetően különösen a tömbökkel kapcsolatos műveletek egyszerűsödnek le, hiszen beépített és névtelen függvényekkel komplex logikát is le lehet írni pár sorban.

Az utóbbi pár évben nagyon sokat használtam a TypeScriptet és én nagyon megszerettem. Mivel a frontend kódnak végül JavaScriptnek kell lennie, hogy egy böngészőben futni tudjon, itt nem volt kérdés, hogy TS-t fogok használni, hiszen egy ekkora projekten már komoly fájdalmakat okozna a sima JavaScript. Backendre lett volna több lehetőségem, azonban annyira hozzászóktam a TypeScripthez, hogy végül itt is ezt választottam.

2.2 Azure

A serverless alkalmazások esetében elengedhetetlen, hogy a fejlesztés legelején kiválasszuk azt a felhőszolgáltatót, ahol a kódunk futni fog. Egy serverless infrastruktúrát nem lenne értelme csak egy alkalmazás kedvéért kiépíteni, ez pont attól lesz olcsó és hatékony, hogy egy nagy szolgáltató alkalmazások ezreit futtatja rajta. Manapság a három legnagyobb felhőszolgáltató az Amazon Web Services (AWS), a Microsoft Azure és a Google Cloud Platform (GCP). Mind a három hasonló szolgáltatásokat és árakat kínál, azonban a részletekben komoly különbségek vannak, így például egy AWS-en elkészített alkalmazás átköltöztetése Azure-be egyáltalán nem triviális, kódot is biztosan változtatni kéne. Tehát a tervezési fázisban el kell dönteni, hogy melyiket válasszuk, és onnantól nem nagyon van visszaút. Ezt is felvehetjük a felhőalapú fejlesztés hátrányai közé, hiszen ha megépítettél egy ilyen alkalmazást, ki vagy szolgáltatva a felhőszolgáltatónak, ugyanis máshol nem tudod futtatni ugyanazt az alkalmazást.

Nálam ez a döntés nem a szokásos módon történt, de mégis egyszerű volt. Egyrészt, ahogy már említettem, elvégeztem a BME-n a Felhő alapú szoftverfejlesztés tárgyat, ami kizárólag Azure szolgáltatásokkal foglalkozott, másrészt pedig a Szövetség Azure kreditekkel tudott engem támogatni. Mivel én csak az Azure-höz értettem, illetve konzulensem is ennek volt szakértője, valamint itt még felhasználható kreditem is volt, nem volt kérdés, hogy ezt választom.

Ezt megelőzően, különösen a tantárgy végzése során, megbizonyosodtam róla, hogy az Azure biztosít olyan szolgáltatásokat, amikre nekem szükségem lesz egy ilyen alkalmazás fejlesztéséhez. A következőkben ezeket fogom röviden bemutatni.

2.2.1 Azure Functions

Az Azure Functions az Azure azon szolgáltatása, mellyel serverless alkalmazásokat lehet készíteni. Egy *Function App* egymástól független függvények halmazát definiálja. Minden függvényhez egy triggerrel kell kötni, aminek bekövetkezésére

lefut az adott függvény. Ez lehet egy HTTP végpont meghívása, egy időpont, vagy egy Azure-ön belüli esemény. A serverless architektúra miatt előfordulhat, hogy két egymás után futó függvény nem ugyanazon a szerveren fut, és természetesen lehetővé teszi ugyanannak a függvénynek a párhuzamos futtatását is.

Az Azure Functions több programozási nyelvet is támogat. Leggyakrabban talán a C# nyelvvel használják, azonban én az előző fejezetben említett okok miatt itt is a TypeScript nyelvet és ezzel együtt a Node.js futtatókörnyezetet választottam. Ezen a nyelven jóval kevesebb forrás volt elérhető Azure Functions témában, de a Microsoft Learn oldalai minden támogatott nyelvre meg lettek írva, így mindig találtam segítséget, amikor elakadtam.

A projektem kezdetén jelent meg a Node.js alapú Azure Functions támogatásának negyedik verziója. Az ilyen támogatás - ami valójában egy JavaScript könyvtár - szabja meg, hogy a projektben hogyan kell a kódot rendezni, illetve hogy hogyan lehet definiálni serverless függvényeket. A legnagyobb előnye az új verziónak, hogy sokkal szabadabb kezdet kap a fejlesztő a projektjének a mappaszerkezetében. Korábban minden függvénynek saját mappájának kellett lennie, ahol a mappa neve volt a függvény neve. A mappában kötelezően lennie kellett egy *index.ts* fájlnek, és ebben a fájlban kellett magának a függvénynek lennie. Kellott még ezen kívül egy *function.json* fájl, amiben a függvény opcióit lehetett leírni, például hogy milyen típusú a függvény, vagy hogy egy HTTP típusú függvény milyen végponthoz tartozik. Ebből az következett, hogy projekt legfontosabb fájlainak mind *index.ts* vagy *function.json* volt a neve, csak a szülőmappa neve volt a különbség közöttük, ami a fejlesztést nehezebbé tette. Ezzel szemben az új, negyedik verzióban nincs semmilyen megkötés a mappaszerkezetre vonatkozóan. A függvények regisztrálása dinamikusan, kódból történik, ezért teljesen a fejlesztőre van bízva, hogy hova teszi függvényeit.

Ezek az újítások nagyon tetszettek nekem, és mivel a projektem még nagyon kezdetleges állapotban volt az új verzió megjelenésekor, átálltam rá. Mivel ez akkor még egy előnézetben lévő verzió volt, voltak kis nehézségek az átálláskor. Az interneten számomra releváns források száma tovább csökkent, hiszen nagyon friss volt akkor ez a verzió. Azonban a Microsoft Learn dokumentációk itt is segítettek. Szeptember óta már nincs előnézetben ez a verzió, sőt, ez lett az alapértelmezett is Node.js alapú Azure Functions projektekhez. [6]

2.2.2 Azure API Management

Ez az Azure szolgáltatás API-ok kezelésére lett kitalálva. Már létező API-ok elé lehet új interfészt tenni, azaz a backend működését elrejtteni az API-ok felhasználói elől. Különböző helyen futó rendszerek elé lehet közös interfészt tenni, akár úgy is, hogy az egyik rendszer Azure-ben fut, míg a másik nem. Az API-t felhasználó alkalmazás ebből semmit sem fog látni, mert ő az API Management által kínált végpontokat fogja hívni, az API Management lesz azért felelős, hogy a háttérben ezeket a kéréseket a megfelelő helyre irányítsa. Ezen kívül képes egy külső, autentikációt megkövetelő API esetén az autentikációs logika megvalósítására, ami így nem az API felhasználó dolga. [19]

2.2.3 Azure SQL

Az Azure SQL a Microsoft SQL Server felhőbeli változata. Ugyanazt az adatbázis motort kapjuk, azonban az ahhoz tartozó üzemeltetési feladatok, mint például frissítés, biztonsági mentések készítése, áthárulnak a felhőszolgáltatóra. Van lehetőség hagyományos adatbázist vásárolni, mely folyamatosan futni fog és bármikor gyorsan elérhető. Azonban van serverless változat is, mely terheléstől függően skálázódik. Így ha hosszabb ideig nem érkezik be kérés, ez is le tud kapcsolni, és innentől az adatbázis futtatásáért nem kell fizetnünk. Azonban az itt tárolt adatok természetesen nem veszhetnek el, ezeknek a tárolásáért folyamatosan fizetni kell, de ez a rész jóval olcsóbb. A következő kérés beérkezésekor aztán az adatbázisszerver újraindul, így a *cold-start* jelenség itt is megjelenik, az ilyen kérések kiszolgálása jóval hosszabb lehet az átlagosnál. [4]

2023 őszétől az Azure minden előfizetésben ingyen adja az első Azure SQL adatbázist. Persze nem teljesen ingyen, a havi első 100 000 másodperc futása az adatbázismotornak és az első 32 GB adat tárolása ingyenes. Ez elsőre soknak hangzik, azonban ez kevesebb, mint 28 óra folyamatos futást jelent, amit még serverless üzemmódban is messze túllép egy ténylegesen használt alkalmazás. Ettől természetesen még valamennyit ér, különösen a tárhely, azt biztosan nem fogom átlépni. [18]

2.2.4 Azure Static Web Apps

Az Azure ezen szolgáltatásával statikus weboldalakat lehet publikálni az interneten. A statikus weboldalak olyan oldalak, amik csak HTML, CSS és JavaScript fájlokat tartalmaznak és teljes egészében a felhasználó böngészőjében futnak, nincs szerveroldali kód. Ilyen szolgáltatás nem csak a nagy felhőszolgáltatóknál elérhető,

hanem például Vercelen vagy Netlify-on is, azonban célszerűbb volt nekem mindent Azure-ön tartani, ezért ezt a szolgáltatást választottam az alkalmazásom frontendjének publikálására. Ezen szolgáltatások mindegyike ingyenes, így itt az ár sem volt kérdés. Az Azure Static Web Apps több módszert is kínál az alkalmazás telepítésére, ilyen például a GitHub integráció, ami egy GitHub repository-ban minden pushra feltelepíti a legújabb változatot Azure-be. [20]

2.2.5 Application Insights

Az Azure Application Insights szolgáltatása futó alkalmazásokról gyűjt és összegez adatokat. Adatokat több forrásból is tud gyűjteni, így egy alkalmazásnak több Azure-ön futó komponensét és be lehet kötni ugyanahhoz az Application Insights példányhoz. Gyűjt egyrészt adatokat a teljesítményről, például hogy mennyi idő alatt válaszolt a kérésekre a szerver. Gyűjt terheltségi adatokat is, azaz hogy mennyire voltak kihasználva az alkalmazásunkat futtató szerverek. Továbbá az alkalmazás által írt naplóbejegyzéseket is ide lehet irányítani, valamint egy böngészőben futó alkalmazás esetén a felhasználók oldallátogatási szokásairól is tud adatokat gyűjteni. Ezekből az adatokból aztán egy saját lekérdező nyelvvel lehet mindenféle kimutatásokat készíteni, nem csak táblázatos, de grafikonos formában is. Összeségében egy nagyon hasznos szolgáltatás, ha meg akarunk győződni arról, hogy az alkalmazásunk gond nélkül fut. [2]

2.3 Backend

Az előző alfejezetben beszéltem azokról az Azure szolgáltatásokról, amelyek az architektúráim alapját adják. Most kitérnék még a backenden, azaz az Azure Functions projektemben használt legfontosabb könyvtárakra, illetve egy másodlagos adatbázisra.

2.3.1 TypeORM

A TypeORM egy Node.js alapú ORM (Object Relational Mapping, Objektum-relációs leképezés) könyvtár. Az ORM könyvtárak lényege, hogy egy interfészt biztosítanak az adatbázisról a szerveroldali kód felé. A backend kódból minden adatbázis felé irányuló kérés ennek a könyvtárnak valamelyik függvényén keresztül fog az adatbázishoz jutni. Így ez a könyvtár felelős az adatbázis kapcsolat nyitásáért és zárásért, a lekérdezések végrehajtásáért, de a TypeORM esetében még a séma módosításáért is. A TypeORM támogatást ad arra, hogy az adatbázis sémáját kódban, TypeScript osztályokkal írjuk le. Ha ebben bármi változás történik, úgynevezett migrációt kell

végrehajtani. Ez ennyit jelent, hogy a könyvtár összehasonlítja az adatbázis jelenlegi sémáját a kódban leírt sémával, és egy olyan SQL szkriptet generál, ami a jelenlegi állapotból a kód által leírt állapotba viszi az adatbázis sémáját. A szkript lefutása után az adatbázis és a kód szinkronban van. Ez akkor is tud működni, ha az adatbázisban már vannak adatok, így lehetővé teszi a folyamatos fejlesztést az alkalmazás megjelenése után is. Mivel a TypeORM ismeri az adatbázis sémáját, azok a függvények, melyekkel lekérdezéseket lehet végrehajtani a táblákon, típusbiztosak tudnak lenni. Ez annyit jelent, hogy ahelyett, hogy egy stringbe kellene SQL parancsot írni, ahol nagy a veszélye egy mezőnév félregépelésének, a TypeORM függvénynek egy TypeScript objektumot kell adni paraméterül, ami leírja mely táblák milyen adatait szeretnénk lekérdezni. Ha ebben a tábla- vagy mezőnevek nem helyesek, azt a TypeORM még fordítási időben jelezni fogja.

Habár talán a TypeORM a legelterjedtebb ORM könyvtár a Node.js világában, nekem nem ez lett volna az első választásom, korábban nem is használtam még. Annál többet használtam a Prisma nevű ORM-et, ami körülbelül ugyanezt tudja, csak véleményem szerint sokkal egyszerűbb szintaxissal. Azonban az Azure Functions projektben nem sikerült megoldanom a Prisma működését, leginkább azért, mert bizonyos fejlesztői parancsok végrehajtásához a Prisma rövid időkre egy másodlagos, úgynevezett árnyék adatbázist is létrehoz a valódi mellett, ami miatt a költségeim is megnőttek volna. [1]

2.3.2 Class-validator, class-transformer

Ez a két Node.js könyvtár a backendre érkező adatok ellenőrzésénél nyújtott óriási segítséget. Mivel a backendem gyakorlatilag egy REST API, a függvények nagyrésze kap a felhasználó által megadott, a HTTP kérésben utazó paramétereket. Mielőtt ezeket felhasználnám, ellenőrizni kell őket, hogy megfelelő formátumúak-e. Hiába teszek meg mindent azért, hogy a frontend alkalmazásból csak jól formált adatokat küldjek a backend felé, ez egy publikus API az interneten, amit bárki más is használhat, nem csak az én kliens alkalmazásom. Ezért szerveroldalon minden adatot ellenőrizni kell, mielőtt azok az adatbázisba kerülnek. Ezt a folyamatot egyszerűsíti nagyon ez a két könyvtár. A bejövő adatok struktúrájára DTO (Data Transfer Object) osztályokat kell létrehozni. Ezek egyszerű TypeScript osztályok, melyek csak adatot tárolnak, metódusaik nincsenek. A class-validator számtalan olyan TypeScript dekorátort biztosít, amelyekkel bizonyos szabályokat lehet megkövetelni a bejövő adat egy mezőjéről. Ez lehet annyira egyszerű

is, hogy kötelező mező, vagy string típusú mező, de olyan is, hogy ennek a mezőnek egy tömbnek kell lennie, melynek minden eleme egy egész szám és minden elem különbözik a többitől. Ilyen logikát minden egyes mezőre kézzel fáradságos lenne írni, ezt a könyvtárt használva ez azonban csak a megfelelő dekorátor alkalmazása a megfelelő mezőn. Egy ilyen osztály kódja TypeScript-ben csak pár mező és pár dekorátor, JavaScript-é fordításkor kerül bele az ellenőrző logika. A HTTP kérésben beérkező adat azonban még nem ilyen osztályban van, hanem csak egy sima TypeScript objektumban, ezért az átalakításért a class-transformer könyvtár felelős.

2.3.3 Redis

A Redis egy olyan nyílt forráskódú adatbázis, amely az adatokat nem merevlemezen, hanem memóriában tárolja. Ennek köszönhetően mind az írási, mind az olvasási műveleteket nagyon gyorsan hajtja végre, azonban nagy mennyiségű adatot nem költséghatékony itt tárolni, ezért a legtöbb rendszerben nem egyedüli adatbázisként, hanem egy másik, hagyományos adatbázist kiegészítő gyorsítótárként használják. Nincsenek táblák benne, minden rekord egyetlen gyűjteményben van, és egy kulcsból és egy értékből áll. [11]

Én is gyorsasága miatt használom a projektben az Azure SQL adatbázis mellé. Kevés forgalomnál ez a cold-start miatt lassan válaszol az első kérésre, ilyenkor jön jól, hogy bizonyos adatokat a Redis adatbázis is ki tud szolgálni. A Redis példányomat szintén Azure-ben telepítem, az Azure Cache for Redis szolgáltatással. A backendem pedig a Node.js-hez írt Redis könyvtárral kapcsolódik ehhez az adatbázishoz és tárolja el az alkalmazás legfontosabb adatait.

2.4 Frontend

A következőkben az alkalmazás webes kliensének fejlesztéséhez használt legfontosabb könyvtárakat fogom bemutatni.

2.4.1 React

A React egy JavaScript könyvtár, amivel könnyen használható és gyors felhasználói felületeket lehet építeni. Saját vagy mások által fejlesztett komponensekből tetszőlegesen bonyolult felületeket lehet összeállítani. A Reacttel fejlesztett oldalakat *Single Page Application*-öknek (SPA) is szokták hívni, ugyanis a végtermék klasszikus értelemben egy oldalból áll. Amikor felkeressük a webhelyet, egy olyan HTML

dokumentum töltődik le, melynek a törzse üres, azonban a fejlécben egy nagy JavaScript fájl be van hivatkozva. Ennek a letöltése után ez építi fel a weboldal struktúráját, majd az oldal minden funkcionalitását is ez végzi. Azaz a szerver felé induló kéréseket - legyenek azok adatkérések vagy adatmódosítások - is JavaScript indítja, valamint a navigációt is ő végzi. Amikor rákattintunk egy linkre, ami az alkalmazás másik oldalára vezet, akkor nincs új HTTP kérés, csak a háttérben futó JavaScript átalakítja az oldal HTML struktúráját. Az ilyen alkalmazások első töltése valamivel tovább tart, mint egy klasszikus weboldalé, de utána sokkal kényelmesebb őket használni. [15]

A React mellett manapság még az Angular és a Vue.js az a két keretrendszer, ami hasonló megoldást kínál frontend fejlesztésre. Nekem messze Reacttel van a legtöbb tapasztalatom, és habár nem tökéletes, már elég jó rutinom van vele. Mivel a projekt lényege a serverless technológiák alkalmazása, minimális időt szerettem volna a frontend fejlesztéssel foglalkozni, ezért mindenképpen egy olyan technológiát akartam választani, amit jól ismerek. Mindemellett a projekt végtermékétől azt várom, hogy valódi környezetben is használva legyen, ezért természetesen a frontendet sem szabad elhanyagolni, azonban a felhasználói élmény javítása főként nem ennek a tárgynak keretében fog megvalósulni.

Egyik nagy előnye a Reactnek más hasonló keretrendszerekkel szemben az az ökoszisztéma körülötte. Számtalan kis könyvtár létezik olyan problémák megoldására, amik szinte minden weboldal készítésénél előjönnek. A következő három könyvtár is ilyen, de ezeken kívül is használok még másokat kisebb feladatokra.

2.4.2 Chakra UI

A Chakra UI egy Reacthez készült komponens könyvtár, azaz előre elkészített komponensek halmaza. Megtalálható benne nagyjából mindenféle komponens, ami egy átlagos weboldal elkészítéséhez szükséges: gombok, ikonok, felugró ablakok, beviteli mezők és még sok minden más. Ezek teljesen testre szabhatóak, de alapértelmezetten egymáshoz illenek. Így ha ilyen elemekből építed fel az oldaladat, akkor garantáltan egy egységes kinézetet kapsz, de van bőven lehetőség személyre szabni. Különösen kényelmes benne, hogy ezek a komponensek TypeScript paraméterként kaphatnak olyan értékeket, amik a CSS-üket fogja felülrni, így CSS írása nélkül lehet egyedi stílust adni az oldalnak. [13]

2.4.3 React Hook Forms

Reactben egy beviteli mezőhöz mindig szükség lesz egy állapot változóra, amiben a jelenlegi érték tárolódik. Egy ilyen változó definiálása nem egyszerű, maga a változó mellett egy függvényt is definiálni kell, amivel frissíteni lehet a változó értékét. Majd ezt a változót és függvényét hozzá kell kötni a beviteli mezőhöz. Ez elég sok lépés, amit minden egyes mezőhöz meg kell lépni, és komplex webalkalmazásokban gyakran sok mező is lehet egy-egy űrlapon. Az ilyen űrlapok építésében segít a React Hook Forms, amivel egy helyen lehet definiálni az űrlapot és annak összes mezőjét. A háttérben ő létrehozza mindegyikhez az állapot változókat, de ezzel nem a fejlesztőnek kell foglalkoznia. Az űrlaphoz validációs szabályokat is lehet definiálni, majd könnyedén lekérni a teljes űrlap, valamint egyes mezők állapotát. [7]

2.4.4 React Query

Ez a React könyvtár a hálózati kérések kezelését egyszerűsíti Reactben. Tipikusan egy SPA szinte minden navigáció után az adott oldal betöltéséhez egy szervertől kéri le az adatokat. Így minden oldal nagyon hasonlóan nézne ki Reactben: a komponens betöltésekor elindul egy hálózati kérés, közben egy állapotváltozóban tároljuk, hogy megjött-e a válasz. Amíg nem jött meg, valamilyen töltőképernyőt mutatunk a felhasználónak. Ha megjött, és az sikeres volt, egy másik állapotváltozóba kerülnek az adatok, a töltés figyelő változót hamisra állítjuk, így megjelennek az adatok a felhasználó számára is. Ha azonban hibás válasz érkezett, egy harmadik változóba kerül a hiba, amit máshogy jelenítünk meg. A React Query ezt a sok változót definiálja helyettünk. Kétféle kérést definiálhatunk benne. Az egyik a *query*, ez egy olyan kérés, mely a komponens betöltésekor azonnal elindul. A másik pedig a *mutation*, amelyet mi kódból tudunk indítani. Mindkét féle kéréshez a könyvtár ad különböző lehetőségeket, hogy lekérdezzük a kérés jelenlegi állapotát, nem kell azt nekünk kezelni. [5]

2.5 Nx monorepo

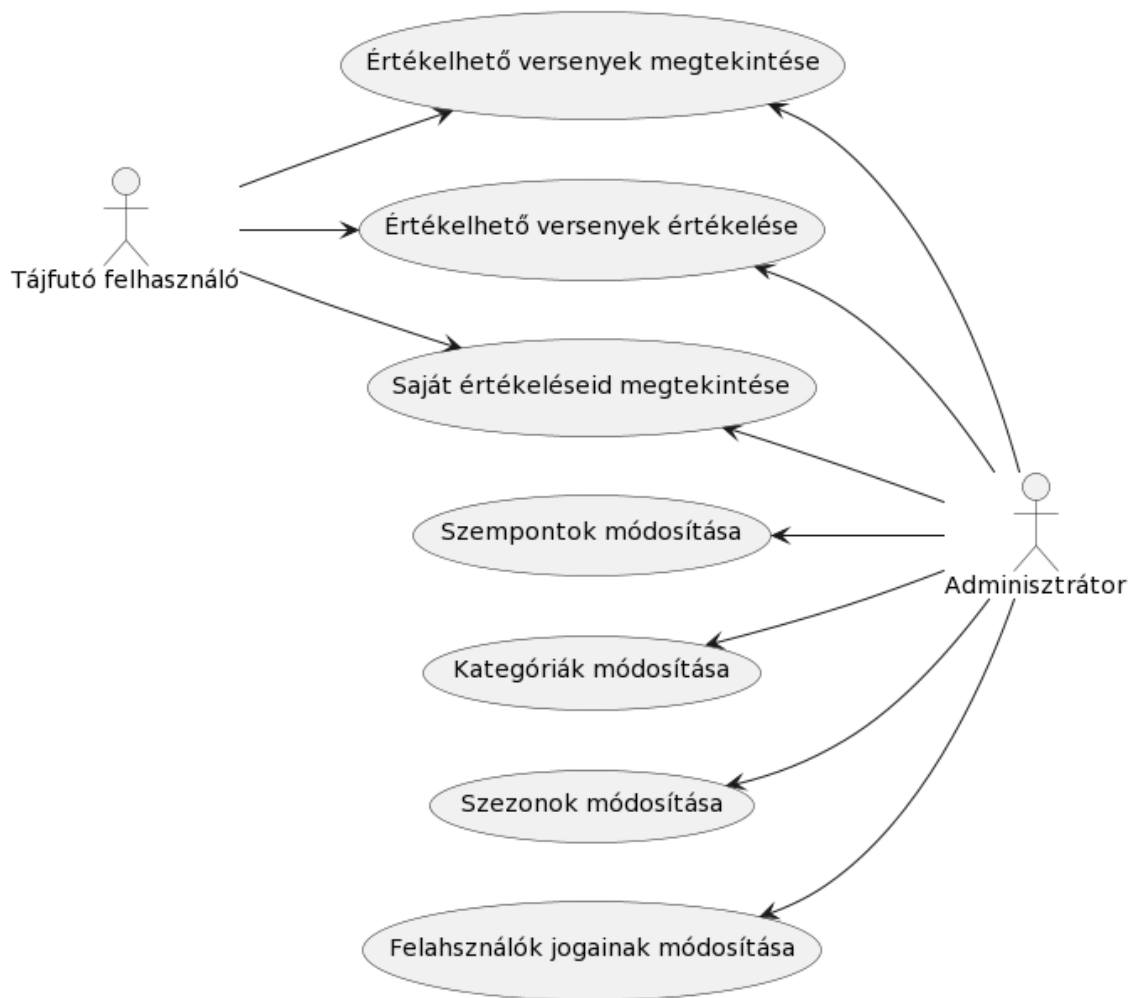
Az Nx egy telepítést segítő rendszer, melyet elsősorban azért fejlesztenek, hogy megkönnyítsék *monorepo*-k karbantartását. Egy *monorepo* azt jelenti, hogy egy projektben több, egymástól valamilyen szinten független alkalmazás van. Ilyen például egy full-stack webalkalmazás, ahol a frontend és a backend magában is működő alkalmazások és sok esetben még különböző nyelven is íródtak. Nagyon sok mindenre lehet használni, az én alkalmazásomban leginkább csak arra használom, hogy azokat a

kódrészleteket, amiket mind a frontend, mind a backend használ, mindkét projektbe belefordítsa.

3 Tervezés

A tervezés első lépése az volt, hogy eldöntsem, milyen környezetben fusson az alkalmazásom. A Szövetség nem tudta volna átvállalni a felmerülő üzemeltetési költségeket, azonban az 1.1.4-es fejezetben említett létező rendszereket rendelkezésemre bocsátották. PHP-hoz én nem értettem és nem is szerettem volna megtanulni, ezért mindenképpen más szerver kellett nekem. Az én alkalmazásomat talán egy virtuális szerveren lett volna a legegyszerűbb futtatni, egy ilyen bérlésének azonban van ára, valamint az üzemeltetése is ránk hárult volna. Mivel az üzemeltetési részhez én is kevésbé értek és a Szövetségben sem volt senkinek kapacitása erre, ez az opció se volt járható. Így döntöttem végül amellett, hogy Azure-ön fog futni az alkalmazásom, de itt még mindig sokféle módon lehet webalkalmazásokat futtatni. A serverless architektúra mellett részben azért döntöttem, mert érdekelt és szerettem volna kipróbálni. Emellett azért volt technikai indok is a választás mögött. Mivel a magyar tájfutó közösség nem nagy, a felfelé skálázódást nem vártam, hogy kihasználja az alkalmazás, azonban lefelé igen. A tájfutó versenyek tipikusan hétfévente vannak, így a felhasználók nagyrésze várhatóan szombattól hétfőig értékelné a versenyt, a hét többi részében minimális forgalmat vártam az oldalra. Ebben az esetben nagyon költséghatékony, ha az alkalmazásom teljesen leáll és semmit nem fizetek érte. Persze csak akkor, ha az első indítás nem tart olyan sokáig, hogy azt a felhasználó ne várná meg.

A tervezés következő fázisában a specifikáció alapján elkészítettem az alkalmazás use-case diagramját. Alapvetően kétféle felhasználó lesz az alkalmazásomban, az átlagos tájfutó, illetve az adminisztrátor. A tájfutók kerülhetnek különböző szerepkörökbe attól függően, hogy milyen módon vettek részt egyes versenyeken, azonban ettől még az alkalmazás pontosan ugyanazon részét látják, csak kicsit más szempontok alapján értékelik a versenyeket.



3.1 ábra: Az alkalmazás use-case diagramja

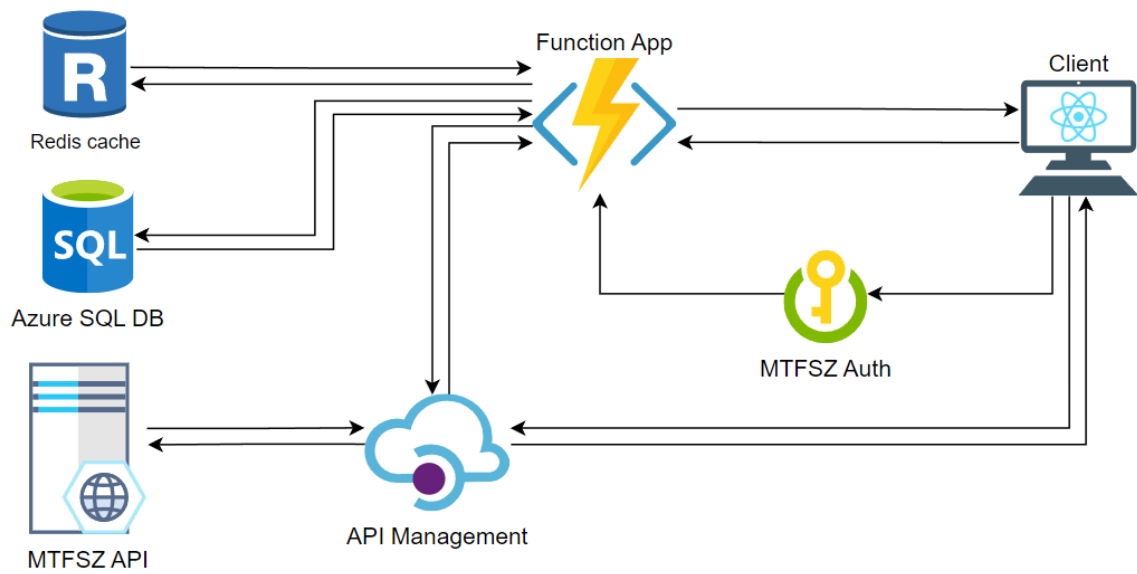
Ezek után el kellett döntenem, hogy milyen adatbázist szeretnék használni, ugyanis Azure-ön több lehetőségem is volt az Azure SQL mellett. Az egyik ilyen az Azure Cosmos Database, ami szintén elérhető serverless módban, azonban ez nem egy relációs adatbázis, hanem NoSQL, azon belül is dokumentum alapú. Viszonylag keveset dolgoztam korábban dokumentum alapú adatbázisokkal, de utánaolvasás alapján arra jutottam, hogy egy ilyen alkalmazás esetén nem éri meg annak a választása. Már a tervezési fázisban tudtam, hogy az adatbázisom sémája egészen komplex lesz, azonban relációs adatbázisokból szerzett tapasztalatom alapján voltak jó ötleteim, hogy ezt hogyan lehetne optimálisan megvalósítani. Ugyanezek a tapasztalatok nem álltak rendelkezésemre egy dokumentum alapú adatbázisnál, valamint ennek az előnyei leginkább akkor jönnek elő, amikor az alkalmazásod terhelése nagy, hiszen ezek az adatbázisok jobban skálázhatóak. Mivel én nem számítottam nagy forgalomra, inkább az általam jobban ismert utat, a relációs adatbázis választottam. [16]

Relációs adatbázisok között is még két lehetőségem volt: PostgreSQL és Azure SQL, mindkettő elérhető Azure-ön. PostgreSQL-t korábban már sokszor használtam, igaz nem Azure-ön, hanem a szakkollégium szerverén futtatva. Soha nem volt problémám vele, ezért szívesen választottam volna ehhez az alkalmazáshoz is, azonban ez Azure-ön nem érhető el serverless módban. Így ezt csak úgy tudtam volna használni, ha a futásáért folyamatosan fizetek. Ezt én az alkalmazásom esetében pazarlónak véltem, valamint a témám is serverless technológiák kipróbálását hangsúlyozta, ezért választottam végül az Azure SQL adatbázist serverless módban.



3.2 ábra: Az adatbázis sémája

Végül megterveztem az alkalmazás alapvető architektúráját, azaz hogy milyen szolgáltatásokat fogok használni, és ezek hogyan kapcsolódnak egymáshoz. Az alkalmazás két legfontosabb része az Azure Functions app (backend) és a React kliens alkalmazás (frontend). Ezek HTTP kérésekkel folyamatosan kommunikálnak. A frontend mind az Azure SQL adatbázist, mind a Redis gyorsítótárat a backenden keresztül éri el. A backend és a frontend is kér le adatokat az MTFSZ API-tól, de ezt mindig egy API Management példányon keresztül teszik. Bejelentkezéskor a frontend alkalmazás irányít át az MTFSZ bejelentkeztető rendszeréhez, majd az a backendre irányít vissza. Ezek közül a Redis gyorsítótár nem szerepelt még az eredeti terveken, erre az első teljesítményteszt eredményei miatt lett szükség.



3.3 Ábra: Az alkalmazás komponenseinek kapcsolatai

4 Megvalósítás

Most, hogy bemutattam, milyen technológiákat, szolgáltatásokat, keretrendszereket és könyvtárakat használtam az alkalmazásom megépítéséhez, illetve hogy milyen tervezési lépéseket végeztem a megvalósítás előtt, szeretném konkrétan elmagyarázni, hogyan is készültek az alkalmazás egyes részei, milyen kihívások elé állítottak ezek a technológiák, és hogy milyen tervezési döntéseket hoztam ezeknek a kihívásoknak a leküzdése érdekében.

4.1 Nx Monorepo

Kezdetben az alkalmazásom gyökérkönyvtárában volt egy *functions* és egy *client* mappa, az előbbiben készült az alkalmazás backendjét adó Azure Functions alkalmazás, utóbbiban pedig a React kliens alkalmazás. Mivel a kettő alkalmazás sokat kommunikál egymással, voltak közös kódrészletek. Például amikor a kliens alkalmazás egy űrlap eredményét elküldi a backendnek, akkor az ahhoz tartozó típus mindkét projektben definiálva volt. Ez természetesen nem könnyítette meg a fejlesztést, hiszen ha az egyik helyen változtattam valamit ezen a típuson, akkor mindig figyelniem kellett, hogy a másik helyen is elvégezzem ugyanazt a módosítást. Azonban mivel a két alkalmazás egymástól függetlenül lett lefordítva, nem volt egyszerűen megvalósítható, hogy ugyanarra a típusra hivatkozzak mindkét helyen. Azonban ahogy az alkalmazás nőtt, már nem volt fenntartható a sok duplikált kód, ezért megoldás után néztem. Az Nx monorepo rendszere jelentett megoldást a problémámra, ezért bevezettem, ami az akkor már egészen nagy projekt teljes átstrukturálását jelentette.

Ehhez a gyökérkönyvtárban létrehoztam egy Nx workspace-t, ami egy *nx.json* konfigurációs fájl jelenti. Ezután a *packages* mappába jöttek az alkalmazásom részei. A kliens alkalmazással könnyű dolgom volt, hiszen az Nx-nek alából van React támogatása, így a már létező React alkalmazást könnyen átalakítottam erre. Azure Functions-höz nem létezett ilyen hivatalos támogatás, azonban az Nx szabadon bővíthető, így már létezett egy, az Nx-től függetlenül fejlesztett Azure Functions projektet támogató plugin. Azon kívül, hogy ez intézte a projekt fordítását, futtatható szkripteket is adott, ezek közül a *deploy* volt a leghasznosabb, amivel az alkalmazás jelenlegi verzióját publikálhattam Azure-be.

Majd létrehoztam egy harmadik projektet *common* néven, és itt definiáltam az összes olyan típust és függvényt, amit mindkét alkalmazás használt. Ez egy egyszerű Node.js projekt, ami magában is fordítható, azonban belépési pontja nincs, ezért futtatni magában nem érdemes. Az igazi kihívás az volt, hogy a másik két projektnek függősége legyen ez a projekt, és így mind a kettő használhassa az itt definiált típusokat és függvényeket. Ehhez a két futtatható projekt *tsconfig* fájljában kellett hivatkozni a common projektre. A tsconfig fájlokban lehet a TypeScript fordításához szükséges konfigurációkat beállítani. Ennek a megértése egészen sok időbe telt, de abszolút megérte, hiszen a refaktorálás befejezése után már nem volt duplikált kód a projektben, ami nagyban könnyítette a fejlesztést.

4.2 Backend

Az alkalmazásom backendje egy, a 2.2.1-es fejezetben említett Azure Functions projekt TypeScript nyelven. A projekt egymástól független serverless függvényekből (továbbiakban SL függvény) épül fel.

Az SL függvényeim nagyrésze HTTP triggerrel működik, azaz akkor hívódnak meg, amikor egy adott végpontra beérkezik egy HTTP kérés. Ilyen SL függvényekkel könnyedén ki lehet építeni egy REST API-t, ami az alkalmazásom legtöbb funkcióját le is fedi. Az Azure Functions Node.js támogatásának negyedik verziójában egy HTTP típusú SL függvény definiálásához az alábbi kódrészletet kell tetszőleges helyre beilleszteni:

```
app.http('events-getOne', {
  methods: ['GET'],
  route: 'events/getOne/{eventId}',
  handler: getOneEvent,
})
```

Az app objektumot a könyvtár adja, ezen a *http* metódus hívásával jelezzük, hogy HTTP triggert szeretnénk az SL függvényhez. Az első paraméter az SL függvény neve lesz, ez igazából csak azonosításhoz szükséges, az Azure Portálon ilyen névvel fog megjelenni. Ezután egy objektumban tudunk további konfigurációt beállítani, például hogy milyen HTTP igékre, illetve milyen útvonalra érkező HTTP kérésekre fusson le az SL függvény. Majd a *handler* opcióhoz meg kell adni egy adott típusú TypeScript függvényt, ami le fog futni a végpont meghívásakor. Ez a függvény paraméterként meg fogja kapni a HTTP kérést reprezentáló objektumot, amin elérheti a paramétereket, illetve egy context objektumot, amin például naplózást tud végrehajtani. Visszatérési értéknek

pedig egy olyan TypeScript objektumot kell adnia, amiből a keretrendszer felépíti a HTTP választ. Meg lehet adni a státuszkódot, fejléceket, illetve az adatot, amit a törzsben küldeni szeretnénk.

Mivel a Node.js támogatás új verziója nagyon nagy szabadságot adott az SL függvények definiálásához, én mappákba rendeztem azokat aszerint, hogy mely entitással foglalkoznak. Így van például egy *events* mappa, ebbe kerültek azok az SL függvények, melyek a versenyek lekéréséhez vagy importáláshoz kapcsolódnak. Minden SL függvény külön fájlt kapott, a fájlnevük megegyezik az SL függvény nevével. Ezekben a fenti kódrészlettel definiálom az SL függvényt, valamint a hozzá tartozó TypeScript függvényt.

A HTTP típusú SL függvényeken kívül van még három Timer trigger típusú SL függvényem is. Ezek nem egy HTTP végpont meghívására futnak le, hanem időzítetten. Az időzítést úgynevezett *NCRONTAB* kifejezésekkel kell leírni. Az ilyen SL függvények definíciója így néz ki:

```
app.timer('events-import', {
  schedule: '0 0 12 * * *',
  handler: importEvents,
  runOnStartup: false,
})
```

Ehhez az app objektumon a *timer* metódust kell meghívni, a *schedule* mezőbe kell a kifejezést írni, a példában szereplő kifejezés azt jelenti, hogy a függvény minden nap 12:00:00-kor fog lefutni. Ilyen típusú függvényt használok többek között a versenyek importálásához és az értékelések lezáráshoz is.

4.2.1 Kapcsolat az adatbázissal

A backend a 2.3.1-es fejezetben bemutatott TypeORM könyvtáron keresztül tartja a kapcsolatot az adatbázissal. Így az adatbázis sémája is kódban van leírva. Ehhez minden táblához egy TypeScript osztályt kell létrehozni, majd erre a TypeORM *@Entity()* dekorát alkalmazni. Az osztály minden olyan tagváltozója, melyre alkalmazzuk a *@Column()* dekorátort, egy mező lesz az adatbázisban. A táblák vagy mezők extra konfigurációs beállításait paraméterként adhatjuk ezeknek a dekorátoroknak. Táblák közötti kapcsolatokhoz is hasonló dekorátorokat kell használni. Alább látható a versenyeket tároló táblát reprezentáló osztály kódja:

```

@Entity()
class Event implements DbEvent {
  @PrimaryColumn()
  id: number
  @Column()
  name: string
  @Column()
  type: string
  @Column()
  startDate: string
  @Column({ nullable: true })
  endDate?: string
  @OneToMany(() => Stage, (s) => s.event, { eager: false, cascade: true })
  stages: Stage[]
  @OneToMany(() => EventRating, (er) => er.event, { eager: false })
  ratings: EventRating[]
  @Column({ default: true })
  rateable: boolean
  @Column()
  @Check("highestRank in('REGIONALIS', 'ORSZAGOS', 'KIEMELT')")
  highestRank: Rank
  @ManyToMany(() => Club, (c) => c.events, { cascade: true })
  @JoinTable()
  organisers: Club[]
  @ManyToOne(() => Season, (s) => s.events, { eager: false, onDelete:
'CASCADE' })
  season: Season
  @Column()
  seasonId: number
}

```

Látható, hogy sima mezők beállításai között meg lehet adni alapértelmezett értéket vagy engedélyezni lehet null értékeket. `@Check()` dekorátorral tetszőleges constraintet lehet definiálni egy mezőre. Kapcsolatoknál be lehet állítani mohó lekérdezést, ami azt jelenti, hogy bármikor lekérdezzünk kódból egy rekordot, annak a kapcsolt rekordjai is lekérdezésre kerülnek és a TypeScript objektumon már elérhetőek lesznek. Ezt globálisan nem érdemes beállítani, hiszen sokszor nincs szükség erre és elég költséges is lehet. Természetesen a TypeORM minden lekérdezésnél lehetőséget ad arra, hogy eldöntsük, mely kapcsolatokat szeretnénk még lekérdezni a rekordok mellé, így mindig pontosan annyi adatot kapunk, amennyire szükségünk van.

Mivel az SL függvények egymástól független futnak, nem tudnak osztozni egy adatbázis kapcsolaton. Így nem az alkalmazás indításakor, hanem minden egyes SL függvény futásának elején kapcsolódok az adatbázishoz. Fontos még azt is megemlíteni, hogy ezt a kapcsolatot egy olyan adatbázis felhasználóval építem ki, akinek van joga rekordokat olvasni, módosítani, létrehozni és törölni, azonban táblákat módosítani vagy törölni nem tud. Így ha egy támadónak sikerülne is az alkalmazás nevében tetszőleges SQL parancsot kiadni az adatbázisnak, nem tudna igazán nagy kárt tenni.

4.2.2 Autentikáció

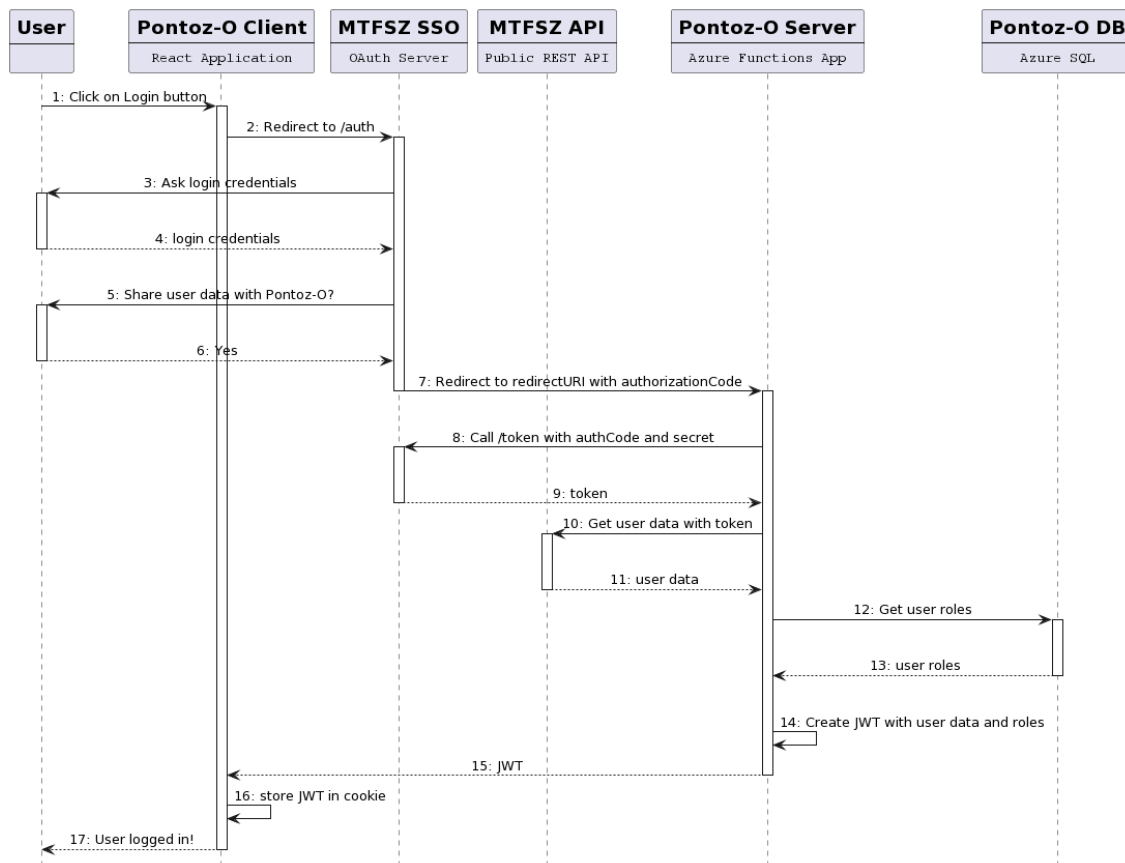
Az autentikáció az a folyamat, amikor a szerver meghatározza, hogy ki küldte hozzá a kérést. Az én alkalmazásomon belül két, egymástól többnyire független autentikációs folyamat is zajlik, ezeket fogom most bemutatni.

Az első fejezetben említett MTFSZ REST API-hoz ha indítunk egy sima HTTP kérést, akkor a *401 Unauthorized* választ fogjuk kapni. Azaz ezt az API-t csak bejelentkezés után lehet használni. Azonban ezen az API-n az elérhető adatok nagy része teljesen független attól, hogy melyik felhasználó kéri azokat. A nyilvántartásban szereplő versenyek, szervezetek és versenyzők adatai találhatóak itt, ezek mindenki számára ugyanazok. Ezeket a végpontokat az én alkalmazásom backendje fogja hívni, nem közvetlenül az én oldalam felhasználója, ezért ide nem is neki kell magát azonosítania, hanem az alkalmazásomnak. Az MTFSZ API-jának üzemeltetőjétől kaptam egy azonosítót és egy kulcsot, ezekkel tudja az alkalmazásom elérni az adatokat. Ezeket természetesen biztonságosan kell tárolnom, így például a GitHub tárolóba se kerültek fel. De ahhoz, hogy értelmes választ kapjak az API-tól, nem ezeket az értékeket kell küldenem a kérés mellé, hanem ezekkel még csak egy *access token* tudok kérni. Ezt kell minden kérés mellé küldeni, időnként azonban meg kell újítani. Ez a folyamat az OAuth 2.0-ás szabványban *Client Credentials* folyamat néven pontosan definiálva van, és az MTFSZ API ezt követi is. Így nem kellett nekem ezt kézzel megvalósítanom, elég volt az Azure API Management példányomban beállítani, hogy ezen végpontokon ilyen típusú autentikációt végezzen el, és a tokenek lekérését és megújítását elvégzi helyettem.

A másik autentikációs folyamat pedig az, amikor az én backend szerveremnek el kell döntenie, hogy melyik felhasználó küldte hozzá a kérést. Ehhez elsősorban az MTFSZ SSO rendszerét használtam, a felhasználóknak ezen keresztül kell bejelentkeznie. Ezt a folyamatot is az OAuth 2.0-ás szabvány írja le, annak is az *Authorization Code* folyamata. Az enyémhez hasonló webes frontend alkalmazásokban ezt a problémát leggyakrabban a *PKCE (Proof Key for Code Exchange)* folyamattal szokták megoldani, azonban az MTFSZ SSO rendszere ezt nem támogatta. Az *Authorization Flow* folyamatban egy ponton egy tokenet kell küldeni az autentikációs szervernek, de ahhoz, hogy a szerver meggyőződjön róla, hogy ezt a tokenet nem egy támadó küldte, egy titkos értéket is kell küldeni mellé. Azonban a frontend kód a felhasználó böngészőjében fut, így ha abban szerepel ez a titkos érték, akkor azt bármely felhasználó megszerezheti. A PKCE folyamattal ezt a veszélyforrást lehet megkerülni,

azonban én ezt nem használhattam. [3] Helyette úgy oldottam meg a problémát, hogy az autentikációs folyamat alatt a logikát nem a frontenden, hanem a backenden, egy SL függvényben végzem. Így a titkos értéknek nem kell megjelennie a frontend kódban, a backend kód fordított verziója pedig nem publikus.

Mivel az *Authorization Code* folyamat is szabványos, erre is léteznek kész megoldások. Azonban az Azure Functions sajátosságai miatt ezeket nem tudtam felhasználni, így ezt a folyamatot magamnak kellett implementálnom.



4.1 ábra: A bejelentkezés folyamata

1. A folyamat ott kezdődik, hogy a felhasználó a frontend alkalmazásban a bejelentkezés gombra kattint.
2. Ekkor átirányítódik az MTFSZ bejelentkeztető rendszeréhez. URL paraméterben utazik közben az alkalmazásom azonosítója, így a rendszer tudja, éppen melyik alkalmazásba jelentkezik be a felhasználó.
3. Ezen az oldalon meg kell adnia az MTFSZ fiókjához tartozó felhasználónevét és jelszavát.

5. Amennyiben először jelentkeznek be az én alkalmazásomba, ezután a lépés után el kell fogadnia, hogy az adatai meg lesznek osztva az alkalmazással.
7. Miután ezt megtette, átirányítódik az én backend rendszerembe, paraméterként most egy úgynevezett *authorizationCode* utazik, amit az SSO rendszer generált.
8. Ezt a szerverem kiolvassa és elküldi az MTFSZ bejelentkeztető rendszerének, mellékelve a titkos értéket is. Ebből tudja az autentikációs szerver, hogy tényleg egy valódi alkalmazáson keresztül jelentkeznek be a felhasználó, és nem egy lopott tokennel próbálkozik egy támadó.
9. Amennyiben ez helyes, válaszban egy access token érkezik. Ezt szintén a bejelentkeztető rendszer állította ki, hogy az MTFSZ API-n keresztül az éppen bejelentkezett felhasználóról adatokat tudjon lekérni a szerverem.
10. A kapott tokennel lekéri a szerver a felhasználó adatait, jelen esetben a felhasználó azonosítóját, nevét, születési dátumát és még pár tájfutás specifikus adatot.
12. Ezeket az adatokat sok rendszer ilyenkor elmenti a saját adatbázisába, azonban nekem nincs különösebben szükségem másra, mint az azonosítóra. Az olyan entitásaim, amik egy adott felhasználóhoz kapcsolódnak, mint például egy értékelés, ezzel az azonosítóval hivatkoznak a felhasználóra. Azonban adatbázis szinten itt nem jön létre kapcsolat, hiszen felhasználókat tároló táblám nincs is. Ez a mező csak arra használ, hogy mindig csak az a felhasználó tudja módosítani vagy lekérni ezt az entitást, akihez az tartozik. Azonban a felhasználók szerepköreit tárolom az adatbázisban, ezek közül bejelentkezéskor lekérem a jelenlegi felhasználóhoz tartozókat.
14. Így megvan a felhasználó azonosítója, alap adatai és szerepkörei. Ezt az objektumot egy JWT-vé (JSON Web Token) alakítom. Ez egy digitálisan aláírt token, azaz bármikor egy ilyet kap a szerverem, biztosan meg tudja mondani, hogy azt az én alkalmazásom állította-e ki, ezzel azt is validálva, hogy a benne szereplő adatok helyesek.
15. Ezzel a JWT tokennel a szerverem átirányítja a felhasználót a frontend alkalmazáshoz. Természetesen a felhasználó számára a jelszava beírása óta semmi értelmes tartalom nem volt, csak a böngésző töltését látta, de ez normális esetben pillanatok alatt megtörtént.
16. A frontend alkalmazás a JWT létéből tudja, hogy megtörtént a bejelentkezés. Amikor ezt megkapja, egy sütiben eltárolja.

Innentől fogva minden kéréshez, amit a backend felé küld a frontend, küldeni fogja ezt a tokent is HTTP fejlécben, a backend pedig ebből fogja tudni, melyik felhasználó küldte. Mégpedig úgy, hogy kiolvassa a fejlécből a tokent, ellenőrzi a helyességét és integritását, majd dekódolja a törzsét, ami a felhasználó alapadatait tartalmazza.

Ennek a folyamatnak a megvalósításához a backenden két dolgot kellett megcsinálnom. Először is létrehoztam egy bejelentkező SL függvényt. Amikor a felhasználó beírta a jelszavát a bejelentkeztető rendszerbe, az ennek az SL függvénynek az URL-jére fog átirányítani. Ebben történik tehát az access token, majd a felhasználó adatainak megszerzése, a szerepkörök kiolvasása, végül pedig a JWT generálása és aláírása. A többi SL függvényemmel ellentétben ez nem egy hagyományos HTTP válasszal tér vissza, hanem átirányít a frontend alkalmazásomra, paraméterként küldve a JWT-t.

A másik dolog, amit meg kellett írnom, az a bejelentkezést igénylő SL függvényeknek azon része, ahol a token ellenőrzésére sor kerül. Ez minden SL függvényben ugyanaz, ezért természetesen ezt is kiszerveztem segédfüggvénybe. Ez kiolvassa a HTTP fejléct és validálja a tokent. Ha nincs token vagy érvénytelen, akkor hibát dob, így az SL függvények nem futnak tovább. Egyébként pedig visszatér a tokenben tárolt adatokkal, ami a felhasználó alapadatai, ezeket az SL függvények fel tudják használni például a jogosultságkezelés során.

4.2.3 Jogosultságkezelés

Miután tudjuk a felhasználóról, hogy kicsoda, meg kell győződnünk arról is, hogy van joga elvégezni az adott műveletet. Erre a problémára is léteznek könyvtárak, azonban az én alkalmazásomban nem volt bonyolult ez, így kézzel oldottam meg. Alapvetően két dologra kellett figyelnem a jogosultságkezelésnél.

Az első az az adminisztrátori jogkör. Szempontokhoz, kategóriákhoz és szezonokhoz csak ők férhetnek hozzá, más felhasználók nemhogy módosítani, olvasni sem tudják ezeket. (Természetesen értékelés közben látják az adott versenyhez tartozó szempontokat, de a teljes listát sohasem látják.) Azokban az SL függvényekben, ahol ezekkel foglalkozok, még a legelején ellenőrizni kell, hogy a felhasználó szerepkörei között ott van-e az adminisztrátor. Mivel ez nagyon sok helyen előfordul, ezt is segédfüggvénybe szerveztem.

A második pedig az, hogy egy felhasználóhoz kötődő entitást csak ő maga tudjon később szerkeszteni vagy törölni. Ehhez persze ilyen entitás létrehozásakor el kell menteni, hogy kihez tartozik, majd minden későbbi ilyen műveletnél ellenőrizni, hogy a jelenlegi felhasználó ugyanaz-e, mint aki létrehozta.

4.2.4 Validáció

Szerveroldalon minden beérkező adatot ellenőrizni kell mielőtt azokat az adatbázisba mentem. Mivel relációs adatbázist használok, az előre definiált séma teljesen rossz formátumú adatokat nem fogadna el, de a legtöbb esetben ennek az eldöntéséhez nem kell az adatbázishoz nyúlni. Mint arról később írok, az adatbázisműveletek a legköltségesebbek az alkalmazásban, mind időben, mind pénzben, ezért ahol csak lehet, szeretném elkerülni őket. Éppen ezért a beérkező adatokat a lehető legtűzetesebben ellenőrzöm, és csak akkor nyitok kapcsolatot az adatbázissal, ha minden rendben van. Ehhez a 2.3.2-es fejezetben bemutatott class-validator könyvtárat használom. A legtöbb esetben itt egyszerű dekorátorok alkalmazásával megkövetelem, hogy egy mező típusa megfelelő legyen, de saját dekorátorok létrehozásával komplexebbek dolgokat is lehet ellenőrizni. A könyvtár beépített dekorátorai vagy egy mező típusát tudják ellenőrizni, vagy egy mező értékét tudják egy konstans értékhez hasonlítani. Azonban az alkalmazásban van lehetőség szezon létrehozására, aminek kezdő és befejezési dátuma is van. Egy szezon természetesen csak akkor érvényes, ha a befejezési dátum később van, mint a kezdődátum. Ehhez két mező értékét kell egymáshoz hasonlítani, erre már nem létezik beépített dekorátor. Szerencsére a class-validator könnyen kibővíthető, így tudtam létrehozni saját dekorátort. Egy olyan függvényt kell írni, ami paraméterként kapja a teljes ellenőrizendő objektumot és igazgal vagy hamissal tér vissza attól függően, hogy sikerült-e a validáció. Dekorátoroknak is lehet paramétere, így ez megkapja annak a mezőnek a nevét, amelyiknél későbbi dátumra kell mutatni, így a függvényben könnyen eldöntöm, helyesek-e a dátumok.

Természetesen lehetnek olyan feltételek is, amiket nem lehet adatbázisműveletek nélkül eldönteni. A leggyakoribb ilyen talán az egyediség ellenőrzése, azonban ilyen ebben az alkalmazásban nem kellett ellenőriznem, de például a szezonokra vonatkozóan van egy olyan feltétel is, hogy két szezon nem lehet átfedésben egymással. Ennek az ellenőrzéséhez természetesen szükséges lekérdezni a szezonokat az adatbázisból, így ezt nem is lehet class-validatorral megcsinálni.

4.2.5 Hibakezelés

Az SL függvények futása során sokféle hiba léphet fel, beleértve például az előbb említett eseteket is, amikor a validáció sikertelen. Mivel a backenden egy REST API-t építünk, ezért fontos, hogy ilyen esetekben is értelmes választ adjak vissza az API felhasználójának. Erre a megfelelő HTTP státuszkód és valamilyen hibaüzenet visszaküldése a leggyakoribb megoldás, így én is ezt választottam. Azonban vannak olyan SL függvényeim, amikben akár tíznél is több dolgot ellenőrzök, és természetesen másféle hibák is keletkezhetnek, mint például az adatbázis nem elérhető, ezért kivételt kapok. Fontosnak tartottam, hogy ezeket minden SL függvényben egységesen kezeljem, úgy, hogy a lehető legkevesebbet kelljen ismételjem magam, hiszen körülbelül 30 SL függvényből áll a backendem, nem lett volna kényelmes, ha egy minden hibát ott kézzel kezelek, ahol először megtalálom.

Ezért létrehoztam egy saját kivétel osztályt, mely példányosításakor paraméterként kap egy hibaüzenetet és egy HTTP státuszkódot. Bárhol, egy SL függvény futása során olyan hibát találok, ami után nem szeretném, ha az SL függvény futása folytatódna, egy ilyen kivételt dobok, hibaüzenetnek és státuszkódnak megadva azt, amit a felhasználónak szeretnék visszaküldeni. Ebben az az igazán kényelmes, hogy ilyet nem csak az SL függvényhez tartozó függvényben lehet dobni, hanem az az által meghívott segédfüggvényekben is. Így az olyan ellenőrzéseket, mint például a class-validator szabályok ellenőrzése, amit nagyon sok SL függvényben ugyanúgy kell ellenőrizni, ki lehet szervezni egy segédfüggvénybe, és ha hiba lép fel, ott dobom a kivételt. Végül minden SL függvény legkülső eleme egy nagy try-catch blokk, ahol az SL függvény lényegi része a try blokkon belül fut. Ebben bármilyen kivétel keletkezik, az egy szintén kiszervezett segédfüggvényben lesz lekezelve, az összes SL függvényben pontosan ugyanúgy. Ha a kivétel az én általam definiált típusú, akkor a kivételnek megadott hibaüzenetből és státuszkódból elkészítem a HTTP választ és ezzel tér vissza az SL függvény. Ha viszont nem ilyen típusú, akkor naplóbejegyzést készítek a hibáról, hogy később lássam, milyen gyakran fordul ez elő, majd egy 500-as státuszkóddal és „Ismeretlen hiba” üzenettel térek vissza. A pontos kivételt nem küldöm el a felhasználónak, ez neki valószínűleg nem lenne érthető, azonban a naplóbejegyzésbe bekerül, így én később ki tudom találni, mi történhetett.

4.2.6 Naplózás

Ahhoz, hogy az alkalmazás valódi működése közben meg tudjak győződni arról, hogy jól működik az alkalmazás, elengedhetetlen a naplózás. Nem csak a váratlan kivételekről készítek naplóbejegyzést, hanem az adatmódosítással járó adatbázis-műveletekről is. Az alkalmazásban szereplő szempontokról, kategóriákról nem tárolom az adatbázisban, hogy melyik felhasználó hozta létre, hiszen erre nincs szükség az alkalmazás szempontjából. Azonban biztonsági szempontból hasznos információ lehet, különösen mondjuk egy törlésnél, ha utólag vissza tudjuk követni, milyen műveleteket hajtottak végre. Ezért minden olyan adatbázis-műveletnél, ahol új adat kerül az adatbázisba, adatmódosítás vagy törlés történik, naplózom a műveletet és hogy melyik felhasználó hajtotta végre. Ugyanígy naplózom azt is, ha egy felhasználó olyan adatokat akar lekérni, amihez csak adminisztrátoroknak van joga. Így ha ez a művelet sikeres lenne, utólag ezt is visszakövethetném.

Azure-ön alapértelmezettként be van kapcsolva, hogy az Azure Functions alkalmazások által készített naplóbejegyzések egy Application Insights projektbe mentésre kerüljenek. Így ezek a bejegyzések bármikor lekérdezhetőek és tárolásuk ilyen mennyiségben is még gyakorlatilag ingyen van. [10]

4.2.7 Függvények felépítése

Az előző fejezetekben olyan problémákról írtam, amelyekkel szinte minden SL függvényemben foglalkoznom kell. Így a leírt megoldások szinte mindenhol azonosak, éppen ezért az SL függvényeim is egységesek. Röviden összegzem, hogy néz ez ki olyan SL függvényeknél, ahol történik adatmódosítás. Az adatlekérő SL függvényeknél is nagyon hasonló a folyamat, csak pár lépést ki lehet hagyni.

Mindegyik egy nagy try-catch blokkal kezdődik. Ha a try blokkon belül bármilyen kivétel keletkezik, az egységes hibakezelő segédfüggvény naplóbejegyzést és hibaüzenetet készít. A try blokk elején megtörténnek az olyan validációs lépések, melyekhez nincs szükség adatbázisra. Lefut a class-validator validáció, ellenőrzöm a HTTP útvonal paramétereit is. Ha szükséges, ellenőrzöm hogy be van-e jelentkezve a felhasználó, illetve adminoknak engedélyezett útvonalakon azt is, hogy admin-e. Ezután az adatbázisból lekérem azokat az adatokat, amelyek szükségesek a további validációhoz vagy a jogosultságok ellenőrzéséhez. Ha itt is minden rendben volt, akkor történhet meg az adatmódosító adatbázisművelet, amiről naplóbejegyzés is készül. Az adatbázis

válaszát még sokszor valamilyen szinten átalakítom, majd ezt küldöm vissza HTTP válaszként.

4.2.8 Gyorsítótár

Ahogy arról később írni fogok, cold-start esetén az adatbázisom első indítása nagyon hosszú időt vesz igénybe, amit egy átlagos felhasználó nem várna ki. Ezért szerettem volna az alkalmazás olyan adatait, amik ritkán változnak, egy gyorsítótárban is tárolni. Így a felhasználó az alkalmazás egy bizonyos részét akkor is tudja használni, ha az igazi adatbázis még nem indult el. Az alkalmazásom főoldalán az értékelhető versenyek vannak felsorolva, ezek napi egyszer változnak, amikor megtörténik az importálás és az értékelések lezárása. Így ha a versenyeket eltárolom a gyorsítótárban, ezt az oldalt a gyorsítótár is ki tudja szolgálni. Habár a felhasználó az értékelést nem fogja tudni megkezdeni, talán addig nem is jutna el, amíg betölt az adatbázis.

Gyorsítótárnak a 2.3.3-as fejezetben bemutatott Redis-t használtam, ebbe kulcs-érték párokat lehet tárolni. Versenyek importálásakor amellet, hogy elmentem őket az adatbázisba, a gyorsítótárba is elmentem őket. A kulcs a verseny azonosítója lesz, az érték pedig a verseny minden adata stringként, JSON-nel szerializálva. Redisben meg lehet adni a rekordoknak egy lejáratási időtartamot, így be tudom állítani, hogy a rekord pont akkor törlődjön a gyorsítótárból, amikor lezárul a verseny értékelése.

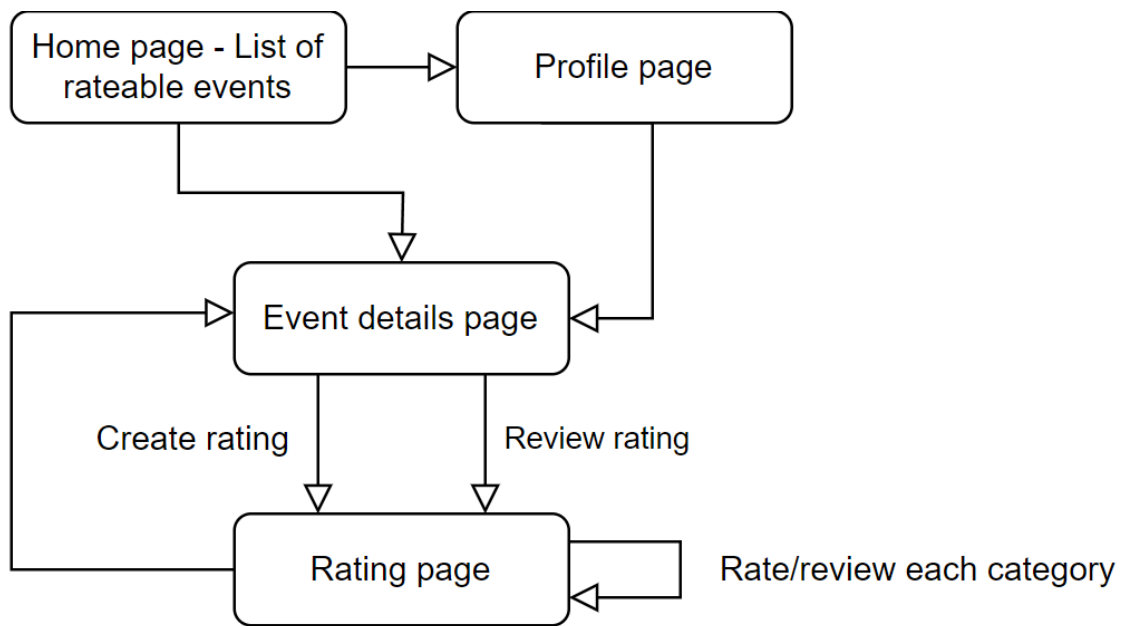
Amikor egy felhasználó az alkalmazásom főoldalára navigál, egy olyan SL függvény hívódik meg, ami kiolvassa az adatbázisból az összes értékelhető versenyt. Ennek mintájára készítettem egy olyan SL függvényt is, ami ezeket az adatokat a gyorsítótárból olvassa ki. Mivel ez az adatbázis sosem kapcsol ki, ez mindig közel ugyanannyi idő alatt fogja elkészíteni a választ. A frontend pedig megoldja azt, hogy egészen addig, amíg a valódi adatbázistól nem jött válasz, a gyorsítótártól érkezett válasz adatai lesznek megjelenítve. Így a felhasználó látja az értékelhető versenyeket, amíg a háttérben a valódi adatbázis újraindul.

4.3 Frontend

Mivel a témám a serverless technológiák megismerése volt, egészen keveset foglalkoztam az alkalmazás frontendjével. Hosszú távon természetesen fontos, hogy az alkalmazás jól nézzen ki és könnyen használható legyen, hiszen egyébként a felhasználók nem fognak visszajárni. Ezekkel tervezek még többet foglalkozni a jövőben, a

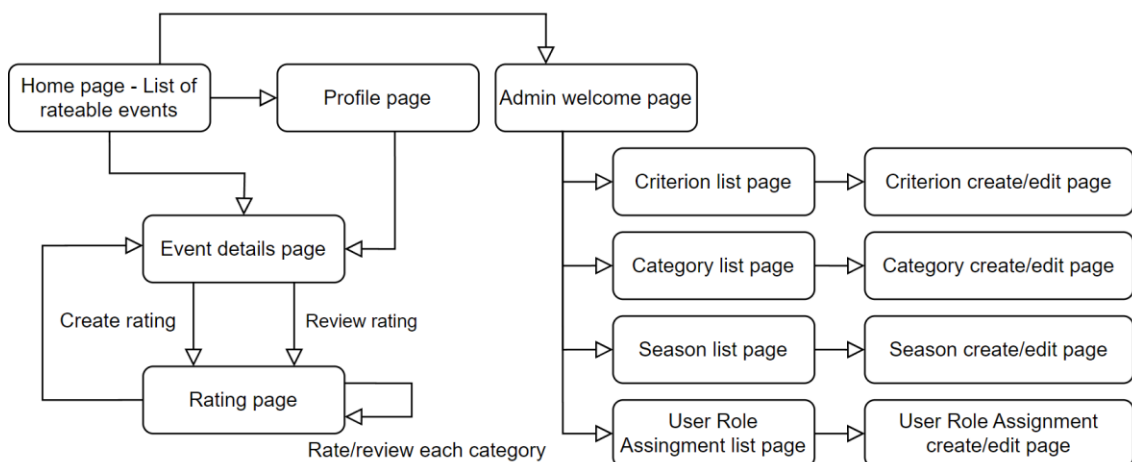
szakdolgozat alatt azonban annyi volt a célom, hogy egy olyan frontendet összerakjak, ami használható és a backend minden funkciójához ad felhasználói felületet.

Az alkalmazás oldalait két nagy csoportba lehet rendezni aszerint, hogy kik számára elérhetőek. A be nem jelentkezett felhasználók csak a főoldalt látják, ezt nem vettem külön csoportnak. Az első csoport azokat az oldalakat tartalmazza, amiket minden bejelentkezett felhasználó lát, ezeken az oldalakon lehet a versenyek értékelését végrehajtani.



4.2 ábra: Minden felhasználó által látható oldalak navigációs diagramja

Az alkalmazás adminisztrátorainak van lehetősége átlépni egy külön felületre, ahol az értékelési szempontok, kategóriák és szezonok módosíthatóak.



4.3 ábra: Adminisztrátorok által látható oldalak navigációs diagramja

4.3.1 UI/UX

Az alkalmazás megjelenéséhez a Szövetség honlapját vettem alapul. Az implementációhoz nagyban támaszkodtam a 2.4.2-es fejezetben bemutatott Chakra UI-ra, amivel nagyon gyorsan lehet igényesen és konzisztensen kinéző weboldalakat készíteni. A navigációs sáv zöldjét választottam az oldal *brand* színének, ebből generáltam egy színskálát. Így például a gombok is ugyanezt a színt használják, de amikor a felhasználó a kurzorral a gombra mutat, a színe egy kicsit sötétül.



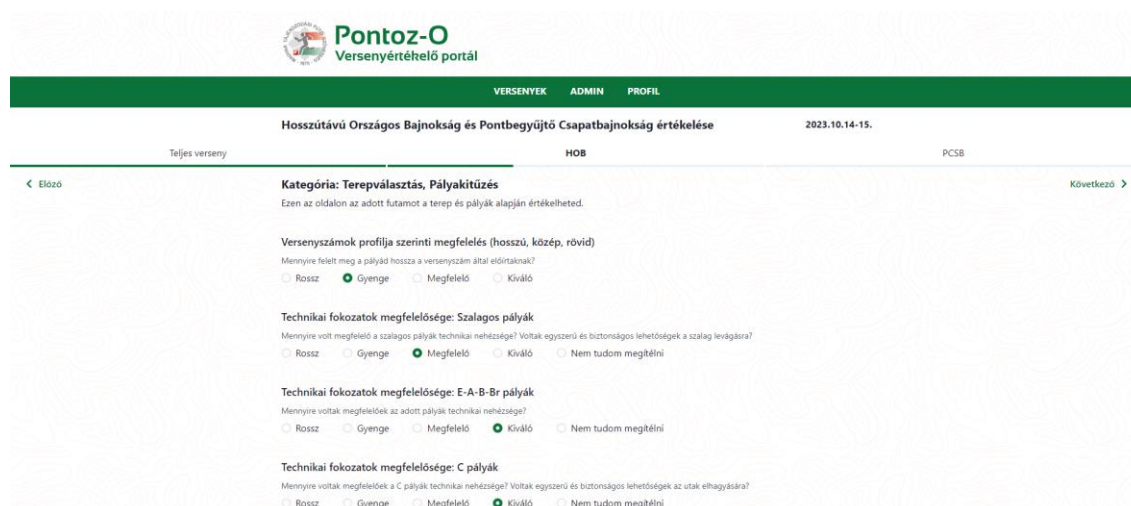
4.4 ábra: Az alkalmazásom főoldala

4.3.2 Értékelő oldal

A weblap legfontosabb oldala egyértelműen az értékelő oldal, ahol a felhasználó egy kategóriába tartozó szempontokat látja és ezek alapján értékelheti a versenyeket. Szerettem volna, ha itt a felhasználók egyértelműen látják, hogy hol tartanak az értékelésben, mennyi van még hátra belőle. Ezért az oldal tetejére raktam egy folyamatjelző sávot, ami azt mutatja, hány kategóriát értékeltünk már az összes közül. Itt jelenik meg az is, hogy jelenleg a teljes versenyre vonatkozó szempontok szerint értékeltünk, vagy valamelyik futamot értékeljük.

Az éppen aktív értékelés adatait és műveleteit egy React Contextben kezelem. A Reactben alaphoz egy komponens a saját gyerekeinek tud adatot továbbadni, így ha két, egymástól a komponensek fáján messze lévő komponens akar adatot közösen használni, akkor nagyon sok komponensen keresztül át kell adni azt az adatot. Ezen probléma megoldására vezették be a Contextet, ami maga is egy komponens. Az ő gyerekei

bármikor el tudják érni azokat az adatokat és műveleteket, amiket a Context megoszt velük, de ezeket nem kell paraméterként átadni, hanem egy úgynevezett *hook-on* keresztül kapják meg. Így a *RatingContext* küldi el egy szempont értékelését a szerver felé, ő menedzseli azt, hogy éppen melyik kategória aktív, valamint ő definiálja azokat a metódusokat is, amelyekkel kategóriát lehet váltani. Az oldal alján látható gombok, a felső sáv alatti gombok és az egyes szempontok rádiógombjai is ezeket a metódusokat hívják.



Pontoz-O
Versenyértékelő portál

VERSENYEK ADMIN PROFIL

Hosszútávú Országos Bajnokság és Pontbegtűző Csapatbajnokság értékelése 2023.10.14-15.

Teljes verseny HOB PCSB

< Előző

Kategória: Terepválasztás, Pályakitűzés Következő >

Ezen az oldalon az adott futamot a terep és pályák alapján értékelheted.

Versenyszámok profilja szerinti megfelelés (hosszú, közép, rövid)
Mennyire felelt meg a pályák hossza a versenyszám által előírtaknak?
☐ Rossz ☒ Gyenge ☐ Megfelelő ☐ Kiváló

Technikai fokozatok megfelelése: Szalagos pályák
Mennyire voltak megfelelő a szalagos pályák technikai nehézségei? Voltak egyszerű és biztonságos lehetőségek a szalag levágására?
☐ Rossz ☐ Gyenge ☒ Megfelelő ☐ Kiváló ☐ Nem tudom megítélni

Technikai fokozatok megfelelése: E-A-B-Br pályák
Mennyire voltak megfelelő az adott pályák technikai nehézségei?
☐ Rossz ☐ Gyenge ☐ Megfelelő ☒ Kiváló ☐ Nem tudom megítélni

Technikai fokozatok megfelelése: C pályák
Mennyire voltak megfelelőek a C pályák technikai nehézségei? Voltak egyszerű és biztonságos lehetőségek az utak elhagyására?
☐ Rossz ☐ Gyenge ☐ Megfelelő ☒ Kiváló ☐ Nem tudom megítélni

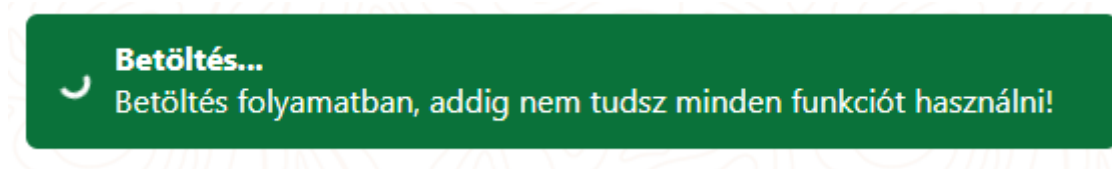
4.5 ábra: Az értékelő oldal

4.3.3 Betöltés gyorsítótárból

Ha egy ideje senki nem használta az alkalmazást, az adatbázis lekapcsol, és az első kérés nagyon sokáig fog tartani. Ehhez bevezettem egy gyorsítótárat, ahonnan a legfontosabb adatok gyorsabban elérhetőek. Azonban minden funkció ilyenkor még nem elérhető az oldalon, például egy verseny értékelésének megkezdése nem lehetséges addig, ameddig nincs kapcsolat az adatbázissal. Ezért valahogy jeleznem kellett a felhasználónak, hogy a háttérben még folyik a betöltés.

Ezt a folyamatot szintén egy Contextben kezelem. Az alkalmazás első betöltésekor egyszerre két kérés indul a backend felé: mind a kettő az értékelhető versenyek listáját kéri le, azonban az egyik az adatbázisból, a másik pedig a gyorsítótárból. A gyorsítótár várhatóan mindig gyorsabban válaszolni fog, azonban az esetek többségében az adatbázisból is egy másodpercen belül megjön a válasz, ilyenkor ezeket az adatokat szeretnénk látni. A gyorsítótár válaszána megérkezésekor elindítok egy időzítőt, és ha a következő másodpercben nem jön meg a válasz az adatbázistól, megjelenítek egy kis üzenetet a felhasználónak, egészen addig, amíg meg nem jönnek az

adatok. Ha az első válasz megjött, akkor az adatbázis felébredt, innentől már minden kérést gyorsan ki fog szolgálni, ezért innentől fogva a gyorsítótárhoz nem küldök kéréseket.



4.6 ábra: A töltést jelző üzenet

4.4 CI/CD

A Continuous Integration és Continuous Delivery (folyamatos integráció és szállítás) módszertanok az én projektben leginkább ott jelennek meg, hogy egy funkció elkészítése és annak az éles verzióban való megjelenése között nagyon kevés lépés van, így nagyon gyorsan lehet a legújabb verziókat telepíteni.

Legegyszerűbb dolgom a frontend projekttel van. Az Azure Static Web Apps alaphoz kínál egy YAML fájlt, amit a GitHub repository *workflows* könyvtárában elhelyezve minden push hatására elindul az új verzió telepítése Azure-be. Ez körülbelül 2-3 perc alatt le is fut, így amikor úgy gondolom, végeztem egy funkcióval, csak pusholom azt GitHub-ra, és pár perc múlva már kint is lesz az éles projekt. Természetesen a YAML fájlban testre lehet ezt szabni, például hogy csak bizonyos *branch*-eken történjen telepítés.

A backenden is működik ugyanez a megoldás, azonban ott jóval lassabb, időnként 9-10 perc is volt a telepítés. Azonban itt más lehetőségem is volt: az Azure Functions Core Tools parancssori eszközzel egy egyszerű parancs kiadásával megkezdődik a telepítés és körülbelül 5 perc alatt végez is. Új backend funkció fejlesztésekor sokszor adatbázis migráció futtatására is szükség van, ehhez ugyanazt az egy parancsot kell kiadnom, mint amit a lokális migrációkor kiadok, csak előtte a konfigurációban át kell állítanom, hogy a *production* adatbázissal dolgozzak.

5 Az éles teszt

Október elejére a projekt első szakaszában elvárt funkciók elkészültek, ezért szerettem volna, ha egy éles teszt keretében kipróbálnák a felhasználók az oldalt. Az összes funkció elkészültéig valódi visszajelzés gyűjtésére még nem lehet használni az oldalt, azonban a teszt keretében a tájfutó közösség is megismerhette, hogy milyen alkalmazás van készülőben, valamint én is értékes visszajelzést kaptam. Itt nem csak felhasználói visszajelzésekre gondolok, hanem arról is többet tudtam meg, hogy milyen a teljesítménye az oldalnak, illetve hogy nagyobb forgalom mellett hogyan alakulnak a költségek.

Október 14-15-én rendezték az Hosszútávú Országos Bajnokságot és Pontbegyűjtő Országos Csapatbajnokságot, körülbelül 1000 versenyző vett részt legalább az egyik futamon. A Szövetség támogatta az éles teszt ötletét, és segítettek is meghirdetni a tájfutók körében. Pár nappal a verseny előtt Facebookon és a Szövetség honlapján jelent meg hír a tesztről, amiben felhívtuk a tájfutók figyelmét erre a lehetőségre. Maga a teszt vasárnap délután 2-től kezdődött, ekkor történt meg a verseny automatikus importálása az alkalmazásba. Ekkor a versenyen hangosbemondóban is elhangzott egy felhívás, azonban a verseny helyszínén alig volt térorő, ezért a versenyzők többsége csak hazaérkezésük után tudta kipróbálni az alkalmazást. A verseny értékelését végül 82 felhasználó kezdte meg az ezt követő hét napban, ebből 69-en értékelték minden szempontot és véglegesítették az értékelésüket. Összeségében ez egészen magas szám, hiszen a tájfutó közösség nagyobb részét idősebb emberek teszik ki, akikről kevésbé elvárható, hogy kipróbálnak egy ilyen új rendszert. Mindenesetre ahhoz mindenképpen elegendő volt, hogy következtetéseket vonjak le az oldal teljesítményéről.

5.1 Előzetes teljesítménytesztek

Természetesen már az éles teszt előtt is végeztem méréseket, ahol az oldal válaszidejét mértem különböző helyzetekben. Az érdekelt igazán, hogy a cold-startnál megjelenő késleltetés mennyire jelentős. Az első próbáknál nagyot kellett csalódnom, ugyanis azt tapasztaltam, hogy amikor mind az Azure Functions projekt, mind az Azure SQL adatbázis lekapcsolt módban volt inaktivitás miatt, az első kérésre körülbelül 60 másodperc után érkezett válasz. Aztán az ezt követő kérések már mind 1 másodpercen belül, általában 500-800 ezredmásodperc alatt zajlottak le, de ez persze attól is függ,

ennyire volt komplex a kérés. Volt, hogy ugyanebben az állapotban is 10-15 másodperc alatt megérkezett a válasz, de a 60 másodpercet is gyakran elő tudtam idézni, különösen ha hosszabb ideje nem volt forgalom. Az alábbi képen azon SL függvény válaszüzei látszanak egy kétnapos intervallumon (még az éles teszt előtti időszakból), amelyik az értékelhető versenyek listáját adja vissza. Ez hívódik meg a főoldal betöltésekor, ezért a legtöbb felhasználó ennek az eredményét látja először. Adatbázis műveletek bonyolultsága szempontjából egy aránylag egyszerű művelet, egy tábla azon értékeit kérdezi le, ahol egy igaz-hamis mező értéke igaz.

Date (UTC)	Success	Result Code	Duration (ms)
2023-09-22 19:07:33.370	✔ Success	200	8165
2023-09-22 12:06:13.757	✔ Success	200	6433
2023-09-22 09:17:01.667	✔ Success	200	55798
2023-09-21 19:22:39.147	✔ Success	200	11604
2023-09-21 15:59:25.264	✔ Success	200	13240
2023-09-21 14:39:00.584	✔ Success	200	770
2023-09-21 14:24:25.110	✔ Success	200	762
2023-09-21 14:22:20.284	✔ Success	200	3066
2023-09-20 07:31:49.698	✔ Success	200	735

5.1 ábra: Az értékelhető versenyeket visszaadó SL függvény válaszüzei

A bal oszlopban az SL függvény meghívásának pontos ideje látszik, a jobb oldalon pedig a függvény futásának időtartama. Láthatunk itt olyan esetet, amikor 55 másodpercig futott a függvény, de az is látható, hogy előtte több, mint 12 óráig nem volt meghívva a függvény, így valószínűleg egyáltalán nem volt forgalma az oldalnak. Láthatunk 10-15 másodperces válaszüzeket is, ilyenkor csak pár óra telt el az előző futás óta. Illetve azt is láthatjuk, hogy rendes forgalom mellett, amikor egy óránál kevesebb idő telt el az előző hívás óta, 700-800 ezredmásodperc a függvény futása.

Még ha sok felhasználót nem is érintene, 50-60 másodperces töltési idő az alkalmazás főoldalán elfogadhatatlan, ezért valamit tennem kellett ezellen, mielőtt az éles tesztről szó lehetett. Hamar észrevettem, hogy olyan SL függvények, melyek nem kapcsolódnak az adatbázishoz, messze nem tartanak ilyen sokáig, hiába volt kikapcsolt módban a futtatókörnyezet. Tehát az adatbázis cold-startja volt az, ami az igazi problémát okozta, nem az Azure Functions projekt. Ezen probléma áthidalására vezettem be gyorsítótárat a versenyek adataihoz, ahogyan azt a 4.2.8-as fejezetben említettem.

Date (UTC)	Success	Result Code	Duration (ms)
2023-11-09 21:44:22.735	✓ Success	200	54
2023-11-09 21:44:02.277	✓ Success	200	3039
2023-11-09 13:03:58.666	✓ Success	200	2248

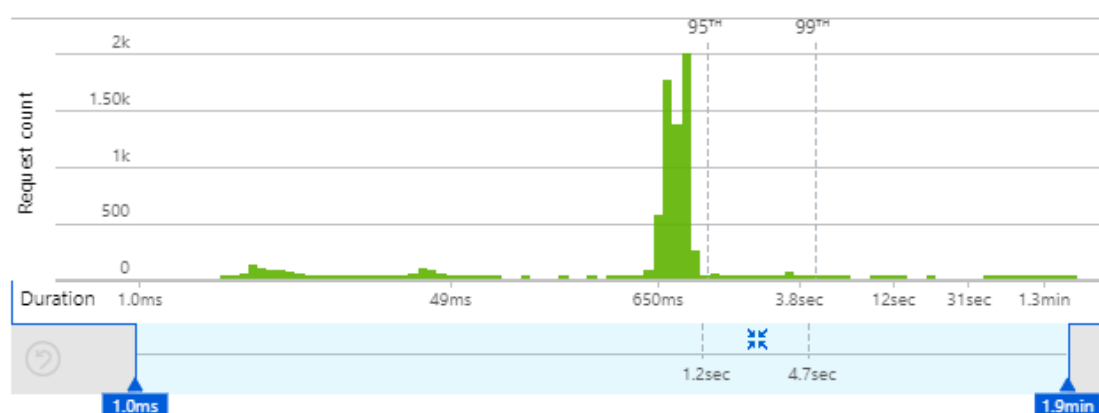
5.2 ábra: Az értékelhető versenyeket a gyorsítótárból lekérő SL függvény válaszüzei

Látható, hogy egy olyan SL függvény, ami nem az adatbázishoz, hanem a Redis gyorsítótárhoz kapcsolódik, nagy kihagyás után is 2-3 másodperc alatt lefut. A Redis gyorsítótár nem serverless elvek szerint működik, tehát bármikor gyorsan olvasható belőle adat. A két-három másodperces késleltetés nagy része tehát maga az Azure Functions cold-startjával járó késleltetés, ami már sokkal elfogadhatóbb az előbbi 55-60 másodperchez képest, különösen mert csak bizonyos felhasználókat érint, és náluk is csak az első kérés tart eddig.

Így született a már korábban is bemutatott megoldás: az oldal első betöltésekor mindkét SL függvény lefut, így az adatbázis felébresztése megindult, de közben a gyorsítótárból a legfontosabb adatokat ki tudjuk szolgáltatni a felhasználóknak: meg tudják tekinteni az értékelhető versenyek listáját és a versenyek adatait, be is tudnak jelentkezni, csak az értékelést nem tudják megkezdeni.

5.2 Teljesítmény, válaszidő

A következőkben az Application Insights szolgáltatás által nyújtott érdekesebb adatokat fogom elemezni. Az adatok minden esetben abból a hét napból származnak, amikor az éles teszt zajlott.



5.3 ábra: A kérések kiszolgálási idejének eloszlása logaritmikus skálán

Ezen a grafikonon az éles teszt hét napján beérkezett kérések kiszolgálási idejének az eloszlása látszik. Megfigyelhetjük, hogy a kérések 95 %-át 1200 ezredmásodpercen

belül szolgálta ki a szerver. Ez nem kiemelkedő teljesítmény, de véleményem szerint teljesen elfogadható, ilyen értékek miatt nem fogok felhasználókat veszteni. Az is látható, hogy 2 percre tartó műveletek is előfordultak, ezek természetesen a korábban említett cold-start esetén történtek. A logaritmikus skála miatt innen nem igazán leolvasható, de pont a kiugró értékek miatt a teljes átlagos időtartam 1370 ezredmásodperc volt.

OPERATION NAME	DURATION (AVG) ↑↓	COUNT ↑↓
Overall	1.37 sec	7.48k
events-close-rating	1.46 min	7
events-import	40.4 sec	7
seasons-init	26.0 sec	7
auth-user	3.75 sec	150
events-rateable	3.54 sec	851
auth-login	2.36 sec	113

5.4 ábra: Az egyes SL függvények átlagos futási ideje és a futások száma

Ezen az ábrán az átlagosan leghosszabb ideig futó SL függvényeim és futási számuk látható. Az első három mind timer trigger típusú, azaz nem egy felhasználói akció váltja ki őket, hanem megadott időpontban automatikusan futnak le. Így itt nem probléma, ha sokáig futnak, mert a futásuk alatt is használható az oldal, senki nem lát emiatt töltőképernyőt. Ezek éjszakára vannak beütemezve, tehát valószínűleg minden futásukkor a cold-start miatt is növekszik a futásidő. A következő három SL függvény pedig jellemzően az oldal első megnyitásakor fut le, így ezek gyakran a cold-start miatt tarthattak tovább, ezért jöttek ki ilyen magas értékek.

5.3 Naplózott hibák

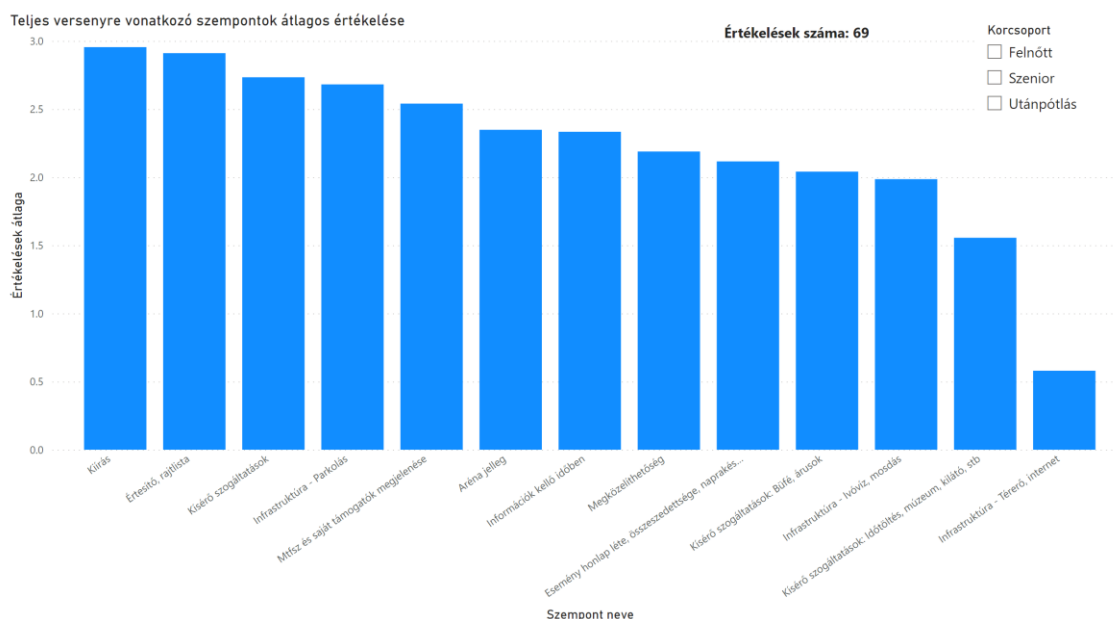
Az Azure Functions projektem automatikusan minden hibát és naplóbejegyzést elküldött az Application Insights-nak, ezért a teszt közben is tudtam figyelni, hogy léptek-e fel hibák. Sajnos volt is pár hiba, az értékelhető versenyeket lekérő SL függvény adatbázishoz csatlakozása többször is időtúllépés miatt kivételt dobott. Ez minden esetben olyankor történt, amikor lekapcsolt állapotban volt az adatbázis, és a függvény futása körülbelül 60 másodpercnél állt meg. Az adatbázis kapcsolaton be volt állítva, hogy csak 120 másodperc után dobjon időtúllépési hibát, de úgy tűnt, hogy ez nem lett figyelembe véve, mert 60 másodperc után már kivétel keletkezett. Természetesen a gyorsítótárból az előzetes adatok ki lettek szolgáltatva, de a felhasználó az értékelést nem tudta megkezdeni amíg le nem frissítette az oldalt. Mivel a kivételeket még az első nap észrevettem, manuálisan beállítottam a kódban, hogy az adatbázis csatlakozásakor

keletkező kivételkor próbálja újra a csatlakozást, így a hiba összesen pár alkalommal lépett fel az első napon. Az előzetes tesztjeim alapján azonban 60 másodpercen belül mindig sikerült az első csatlakozás is, így újabb csalódás volt azt látni, hogy ez tovább is tarthat.

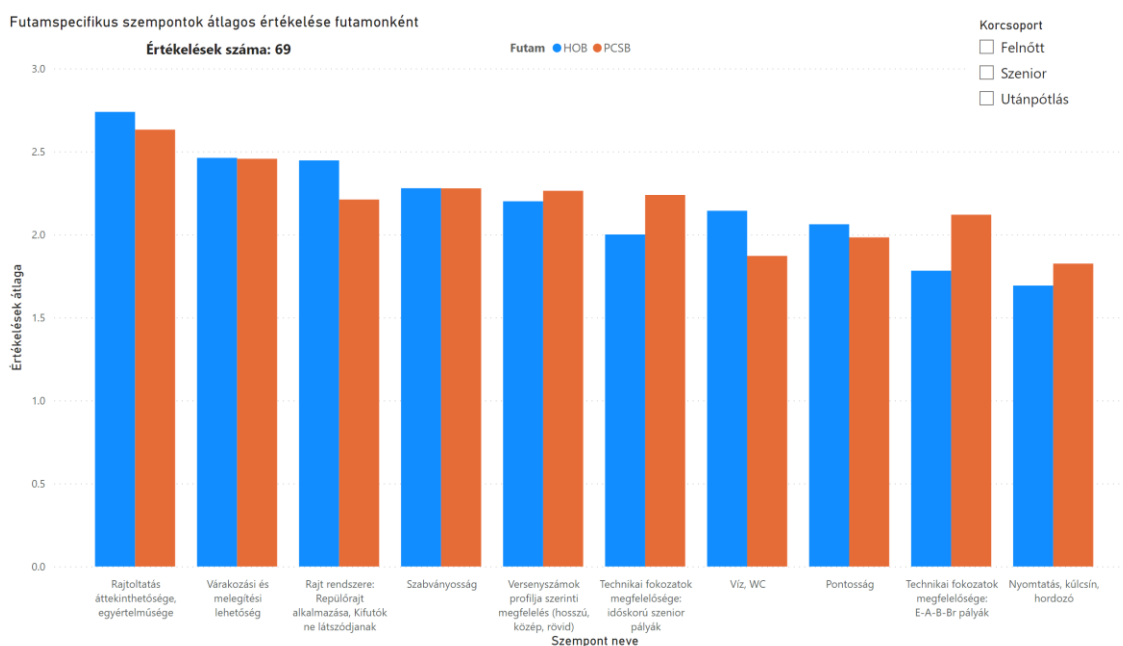
5.4 Beérkezett adatok

A projekt jelenlegi fázisában az a funkció még nem készült el, hogy a felhasználók beérkezett értékeléseit összegezzem és igényes formában megjelenítsem. Azonban minden bizonnyal a felhasználók is csalódtak volna, ha nem látják az értékelések eredményét, ezért Power BI-jal készítettem pár diagrammot ezek megjelenítésére. A Power BI tud közvetlen kapcsolódni az Azure SQL adatbázishoz, és magától elvégzi az értékelések csoportosítását és átlagolását. Ez a módszer annyira könnyű volt, hogy megvizsgáltam, lehetséges lenne-e a végleges alkalmazásba beépíteni ezt. Sajnos egy Power BI dashboard saját weboldalba beágyazása csak előfizetőknek lehetséges, és ez az előfizetés semmiképpen se fért volna bele a kiszabott keretbe. Mindenesetre a projekt következő fázisában meg fogom vizsgálni, hogy létezik-e *open-source* vagy ingyenes alternatívája a Power BI-nak, vagy magamnak kell implementálnom az értékelések összesítését.

Az értékelések eredményének megosztásához két diagrammot készítettem. Az elsőn azok a szempontok szerepelnek, amelyek a teljes versenyre vonatkoznak. Minden szemponthoz olyan magas oszlop tartozik, amilyen átlagos értékelést elért az értékelők körében. A másodikon pedig a futamspecifikus szempontok láthatóak, mindegyikhez két oszlop tartozik, futamonként egy-egy. Mindkét diagrammon szűrésként beállítható, hogy milyen korcsoportba tartozó felhasználók értékeléseit szeretnék beleszámolni.



5.5 ábra: Teljes versenyre vonatkozó szempontok átlagos értékelése



5.6 ábra: Futamspecifikus szempontok átlagos értékelése futamonként

5.5 Visszajelzések

A felhasználóktól emailben gyűjtöttem visszajelzéseket, ezekben leginkább a szempontokkal kapcsolatban voltak elégedetlenek, amelyek fogalmazását és kialakítását nem én végeztem. A felhasználói felülettel kapcsolatban az értékelő oldal átláthatóságáról kaptam egy kritikát, ezt mindenképpen szeretném javítani a projekt következő fázisában. Mivel az oldal teljesítményével vagy működésével kapcsolatban nem kaptam más kritikát, azt gondolom, hogy aránylag jól működött.

A megrendelőkkkel a projekt teljes lefutása alatt nehézkes volt a kommunikáció, sokszor kellett sokat várnom a válaszukra. A teszt után sem kaptam részletes visszajelzést, de alapvetően elégedettek voltak a projekt haladásával.

5.6 Költségek

A Szövetség Azure előfizetésében évi 3500 dollárt kaptam, ami jövő májustól 2000 dollárra csökken, így hosszútávon havi 160 dollár áll rendelkezésemre. Mivel serverless technológiák használata esetén a fizetendő költségek nagyban függenek az alkalmazás terhelésétől, az éles teszt arra is nagyon hasznos volt, hogy megbecsülhessem, a jelenlegi architektúrával bele fogok-e férni a havi keretbe. Sajnos az Azure a nonprofit előfizetések számára nem engedélyezi az Azure általános költség vizsgáló felületeit, hanem egy külön felületet kell használni erre a célra.⁵ Ezen a felületen egy dolgot látunk: egy adott időtartama vonatkozó költségeket, szolgáltatásokra lebontva.

SERVICE NAME	SERVICE RESOURCE	SPEND
SQL Database	vCore	\$42.98
Redis Cache	CD Cache	\$3.7
Log Analytics	Pay-as-you-go Data Ingestion	\$0.27
Functions	Standard Execution Time	\$0.05
Storage	LRS Write Operations	\$0.04
Storage	Protocol Operations	\$0.01
Storage	Read Operations	\$0
Storage	Read Operations	\$0
Storage	LRS Data Stored	\$0
Bandwidth	Standard Data Transfer Out	\$0
Functions	Standard Total Executions	\$0
Storage	LRS Data Stored	\$0
API Management	Consumption Calls	\$0
SQL Database	General Purpose Data Stored - Free	\$0
Storage	Batch Write Operations	\$0
Azure App Service	F1 App	\$0

5.7 ábra: Az Azure szolgáltatások költsége az éles teszt ideje alatt

Az éles teszt hét napja alatt összesen 47 dollárt költöttem el, tehát ilyen forgalom mellett a havi összeg körülbelül 190 dollárra jönne ki, persze csak akkor, ha minden hétvégén ilyen nagy létszámú verseny lenne. Ez egyáltalán nem jellemző, azonban a jövőben könnyedén előfordulhat, hogy a versenyzők nagyobb százaléka fogja értékelni a versenyt, így ezek a számok nem túl biztatóak a keretben maradás szempontjából. Látható, hogy a költségek több, mint 90%-át az adatbázis adja, még úgy is, hogy amikor nem volt forgalom, ezért nem kellett fizetnem. Heti 3,7 dollár a Redisért nem meglepő, ebből a legolcsóbb változatot választottam, mely mivel nem serverless elvek szerint

⁵ <https://www.microsoftazureponsorships.com/Balance>

működik, konstans 16 dollárba kerül egy hónapban. A többi szolgáltatás, köztük a teljes backendemet futtató Azure Functions ára elhanyagolható, ezek árazása szintén a terheléstől függ, ami az én esetemben olyan kicsi volt, hogy közel ingyen használhattam őket.

6 Serverless technológiák értékelése

Az 1.2-es fejezetben felsoroltam a serverless technológiák előnyeit, hátrányait, valamint hogy az előnyöket hogy tudná kihasználni ez a projekt. Ebben a fejezetben arról fogok írni, hogy körülbelül fél év fejlesztés és egy hétnyi valós forgalom alatt ezek mennyire valósultak meg, mennyire tudtam kihasználni az előnyöket és hogy mennyire voltak fájdalmasak a hátrányok.

6.1 Azure Functions

A projektben a legfontosabb serverless komponens természetesen az Azure Functions, ami az alkalmazásom backendjeként szolgál. Ennek használata leginkább a fejlesztői élményt határozta meg, de természetesen a teljesítményre is hatással volt.

6.1.1 Fejlesztői élmény

A serverless fejlesztés egyik fő előnyének mondják, hogy a fejlesztők teljes mértékben a kód írására fókuszálhatnak, hiszen az üzemeltetési feladatok a felhőszolgáltatóra hárulnak. Hogy ez mennyire volt így, arról a következő fejezetben írok, de amikor a kódolásra fókuszálhattam, akkor is jelentett nehézségeket a serverless környezet. Leginkább abban, hogy az általam sokat használt keretrendszereket nem tudtam itt használni, ami miatt kevésbé voltam hatékony. Más projektben legtöbbször a NestJS⁶ nevű keretrendszert használtam REST API-k fejlesztésére, és ez után nagy visszalépés volt az Azure Functions Node támogatásával készíteni egy API-t. A NestJS-nek a gyakran előforduló problémákra, mint például validáció, hibakezelés vagy autentikáció, nagyon kiforrott és egyszerű megoldásai vannak, legtöbb használatához csak egy-egy dekorátort kell alkalmaznom a megfelelő helyen. Ehhez képest az Azure Functions esetében az ilyen problémák kezelését kézzel kellett megírnom. Természetesen használhattam könyvtárakat, mint például a korábban említett class-validator, de ezek futtatására nekem kellett elkészítenem egy keretet, hogy minél kevesebb duplikált kódrészlettel tudjam őket sok helyen használni. Szóval ha az lenne az egyetlen szempont, hogy milyen gyorsan lehet egy adott alkalmazást lefejleszteni, akkor az Azure Functions

⁶ <https://nestjs.com/>

egészen hátul lenne a sorban, vannak nála sokkal több támogatást nyújtó keretrendszerek is.

6.1.2 Üzemeltetés, telepítés

Az oldal Azure-be telepítése valóban nagyon egyszerű volt, az Azure Functions CLI-vel egyetlen parancs kiadásával a legfrisseb, lokális verzió kerül publikálásra, ami innentől elérhető az interneten. Persze ahhoz, hogy ez működjön, kell, hogy legyen egy Azure előfizetésünk, abban pedig egy Azure Functions App. A szokásos üzemeltetési feladatokat, mint például operációs rendszer frissítése, hálózati beállítások konfigurálása tényleg nem a mi dolgunk, vagy ha mégis, akkor az Azure portálon egyszerűen beállítható, bár azért kell egy kis idő, mire ennek a használatához hozzá szokik az ember.

Továbbá, mivel a futó alkalmazásért annak függvényében fizetünk, hogy milyen volt a terhelés rajta, arra is folyamatosan figyelniünk kell, hogy milyen összeg kerül kiszámlázásra. Ez normál előfizetésnél szintén megtehető az Azure portálon, ahol nagyon jó eszközök érhetőek el, lehet például értesítéseket beállítani, ha egy bizonyos összeg felett költünk. Ahogy azt már említettem, a nonprofit előfizetésben ezek a funkciók azonban nem elérhetőek, így a költségeket egy külső oldalon kell figyelni, ami jóval kevesebb kényelmi funkciót nyújt.

6.1.3 Elérhető irodalom

Az utóbbi években a serverless fejlesztés egészen elterjedt lett, ezért ebben a témában bőven talál az ember segítséget az interneten. Én azonban megnehezítettem a dolgomat azzal, hogy Azure Functionst használtam Node.js-szel és TypeScript-tel. Mivel az Azure Functions-t a Microsoft fejleszti, ezt leggyakrabban a C# nyelvvel szokták használni. Sőt, már a projektem elején a Node.js támogatásának új verziójára váltottam, ami a fejlesztés időtartamának nagy része alatt még előnézetben volt. Így meglehetősen kevés forrást találtam, ahol pontosan ilyen összeállításban használták az Azure Functionst. Azonban a Microsoft hivatalos dokumentációja, a Microsoft Learn a serverless fejlesztés legfontosabb részeiről részletes leírást adott, és ez az összes támogatott programozási nyelven elérhető volt. Azonban amit itt nem találtam meg, azt általában sehol máshol sem. Volt, hogy az Azure Functions Node.js támogatásának GitHub tárolójában kérdeztem, és meglepetésemre gyors és kielégítő választ kaptam a fejlesztőktől.

6.1.4 Cold-start

Arra már előre is számítottam, hogy az alkalmazásom nem fogja tudni kihasználni a felfele skálázódás lehetőségét. Ez így is lett, ilyen forgalom nem volt az oldalon a teszt ideje alatt, és várhatóan valódi használat mellett se lesz. Azonban a lefele skálázódás jól jött, azaz voltak hosszabb időtartamok, amikor egyáltalán nem volt forgalom az oldalon, ilyenkor az alkalmazásom egyáltalán nem futott az Azure szerverén, és így a számlám sem nőtt. A következő kérés beérkezésekor fellépő késleltetés, azaz a cold-start egyáltalán nem volt vészes. Olyan kéréseknél, amik nem fordultak az adatbázishoz, mindössze három másodperc volt a válaszügy. Ha minden kérés eddig tartana, az probléma lenne, de mivel ez csak bizonyos felhasználókat érint, és nekik is csak az első kérés ilyen hosszú, nem gondolnám, hogy ez jelentősen rontaná a felhasználói élményt. Persze nagyon kevés az olyan kérés, aminek nincs szüksége adatbázisra, ezért ennél jóval nagyobb válaszügyök is előfordultak, de erről majd az 6.2-es fejezetben írok bővebben.

6.1.5 Költségek

Mint azt a 5.6-os fejezetben láttuk, az Azure Functions szolgáltatásért nagyon minimálisat kellett fizetnem hét napnyi valós forgalom után. Nem volt könnyű előre megbecsülni az összeget, ezért eléggé meglepődtem, hogy ez ennyire olcsó.

Az Azure egy SL függvény futási költségét *GB-s* mértékegységben számolja. Az értéket a futás alatt felhasznált memória és a futási idő adja együttesen. Konstans memóriahasználatnál egy SL függvény futásának ára a felhasznált memória *GB*-ben szorozva a futási idővel, másodpercben. Természetesen a memóriahasználat sosem konstans, ezért a képlet valójában bonyolultabb ennél, de ez a két faktor játszik szerepet benne. Ezen kívül még minden SL függvény futtatásáért is fizetni kell egy kis összeget, 0,20 dollárt egy millió futásért. [9]

Havonta 400 000 *GB-s*, illetve 1 000 000 futtatás ingyenes, előtölt pedig 0,000016 dollár minden *GB-s*. Az egyhetes teszt alatt 3340 *GB-s*-t használtam és összesen 7480-szor futott SL függvényem. Ezek a számok messze az ingyenes szint alatt vannak, azonban a 5.7-es ábrán látható, hogy az Azure Functions-ért mégis fizettem 0,05 dollárt. Ha kiszámolom az ingyenes keret feletti képlettel az árat, akkor pont kijön a 0,05 dollár, tehát valószínűleg ez az akció nem érvényes a nonprofit előfizetésekre. Erről korábban nem tudtam és nem is találtam információt róla, de mivel így is nagyon minimálisak a költségek, nem nagy veszteség ez.

Az Azure Functions-nek van prémium verziója is, ami ugyanolyan, mint a serverless, azzal az egy különbséggel, hogy lefele sohasem skálázódik, azaz egy példány mindig fut az alkalmazásból, így sosem lesz cold-start miatt megnövekedett válaszidő. Ennek viszont az ára is sokkal magasabb, becsléseim szerint minimum 100 dollár havonta, ami a jelenlegi adatbázis konfigurációval biztosan nem fér bele a keretbe.

Az backendem működéséhez még több kiegészítő Azure szolgáltatást is felhasználtam, mint az API Management, a Storage Account és az Application Insights. Ezek költségei szintén elhanyagolhatóak voltak a teszt alatt, szóval összeségében azt mondhatom, hogy az adatbázist és gyorsítótárat nem tekintve nagyon olcsón, havi pár száz forintból kijött az alkalmazás.

6.2 Azure SQL

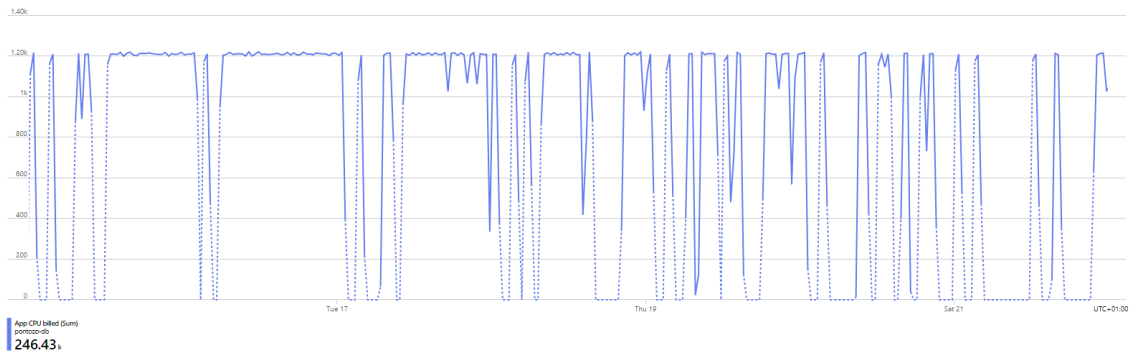
Azonban az adatbázistól természetesen nem szabad eltekinteni, hiszen előzetes becsléseim alapján is ezt vártam a legdrágábbnak, azonban az ára így is felülmúlta a várakozásaimat, és sajnos nem ez volt az egyetlen ok, amiért csalódtam a szolgáltatásban.

Az Azure SQL a Microsoft SQL Server adatbázismotorját használja, amit én korábban ritkán használtam, ha tehettem inkább PostgreSQL-t választottam. A kettő közötti különbségek közül egy tűnt fel fejlesztés közben, ez pedig az volt, hogy Azure SQL-ben nem lehetséges *enum* típusú mezőt felvenni. Ezt több helyen is használtam volna az alkalmazásomban, például egy futam rangsoroló típusa mindig négy érték közül pontosan egyet vesz fel. Szerencsére könnyen áthidaltam a problémát, ezeket a mezőket szöveges mezőnek vettem fel és egy SQL szintű *constraintet* alkalmaztam rájuk, hogy csak az a négy értéknek valamelyike kerülhessen a mezőbe.

Sokkal nagyobb hátrány volt a cold-start miatt fellépő késleltetés. Mivel a serverless opciót választottam, az adatbázis lekapcsolt, ha egy óráig nem jött kérés felé. Amikor aztán megint jött kérés, az adatbázis szervert újra kellett indítani, melynek időtartama attól is függ, milyen régóta volt lekapcsolt állapotban. Mint azt korábban említettem, 60 másodperc feletti válaszidők is előfordultak, amiket elfogadhatatlannak tartok, ennyit a felhasználók nem hajlandóak várni egy kérésre.

Előzetesen azt vártam, hogy a teszt későbbi napjaiban már nagyon alacsony lesz az alkalmazás forgalma, ezért az adatbázis sokat lesz lekapcsolt állapotban, így sokat fogok spórolni a serverless opcióval. A forgalom valóban alacsony volt, viszont meglehetősen elszórt, így napi szinten többször kapcsolt be újra az adatbázis szerver, ami

után mindig egy óráig futott is, pedig sok esetben ennek nagyrésztében már nem is volt terhelés.



4.8 ábra: Az adatbázisom vCore másodperceinek félóránkénti összege

Ezen az ábrán az látható, hogy a teszt hét napja alatt hány vCore (virtuális mag) másodpercet számlázott ki nekem az Azure, 30 percenként összegezve. Az adatbázisom úgy van beállítva, hogy terheléstől függően minimum fél, maximum egy vCore-t használhat fel. Folyamatos forgalom mellett 1200 körüli értékeket látunk, azaz ilyenkor körülbelül kétharmad vCore-t használt az adatbázisom. A grafikonon az is jól látszik, hogy milyen gyakran és milyen időközökben volt lekapcsolt állapotban az adatbázis. Látható, hogy az első két napon, a nappali órákban szinte sosem kapcsolts le, később azonban már gyakran, viszont nappal sosem maradt sokáig kikapcsolt üzemmódban.

Összeségében én úgy gondolom, hogy nem megfelelően használtam ki ezt a szolgáltatást. Azoknak a felhasználóknak a nagy része, akik a későbbi napokban használták az oldalamat, a gyorsítótár beüzemelése ellenére is csak egy nagyobb késleltetés után tudták rendesen használni az alkalmazást, de pénzt mégse spóroltam sokat, hiszen az idő többségében így is futott az adatbázis szerver. Valódi felhasználók által használt alkalmazásban a hosszú cold-start miatt a lefelé skálázódás nem használható, a felfelé skálázódást pedig itt sem tudtam igazán kihasználni.

Az éles teszt előtt olyan opcióval is próbálkoztam, hogy kikapcsolom azt a módot az adatbázison, hogy inaktivitás esetén lekapcsoljon. Így azonban a havi költség több száz dollár lenne. Az Azure nyújt különböző akciókat, például ha egy évre előre lefoglalsz egy adatbázist, de az én havi keretembe ezzel sem férne bele egy folyamatosan futó serverless adatbázis.

6.3 Összevetés egy virtuális szerver bérlésével

A projekt kezdete előtt volt egy olyan opció is, hogy az alkalmazást egy bérelt virtuális szerveren futtatnánk. Ez az opció végül el lett vetve, egyrészt mert ez plusz költséget jelentett volna a Szövetség számára, másrészt pedig mert így a szerver üzemeltetése is ránk hárult volna. Virtuális szervert már havi pár ezer forintért lehet bérelni, üzemeltetésben pedig nekem egy kis tapasztalatom már van szakkollégiumi projektekből. Természetesen ilyen környezetben nem tudtam volna serverless technológiákat használni, de mint az láttuk az előző fejezetben, nem is igazán voltak ezek szükségesek erre a projektre. Használhattam volna viszont az általam jól ismert keretrendszereket, így valószínűleg ugyanazt a funkcionalitást jóval hamarabb elkészítettem volna, a maradék időben pedig beletanulhattam volna az üzemeltetési feladatokba. Virtuális szerveren a saját adatbázisszerverem futtatása sem nagy feladat, ezért a havi körülbelül 200 dolláros Azure SQL helyett a virtuális szerver párezeres áráért kaptam volna egy olyan adatbázist, amit még csak le sem kell kapcsolni inaktivitás esetén. Persze ez nem tudna terheléstől függvényében felfelé skálázódni, de mint azt láttuk, erre nincs is szüksége az alkalmazásomnak a jelenlegi felhasználó bázis mellett.

Amit viszont vesztettem volna, az a serverless technológiák megismerése. A tervezési fázisban is szempont volt egy új technológia megismerése iránti vágy. Csak műszaki szempontokat nézve, valószínűleg jobban megérte volna virtuális szervert bérelni, azonban ez számomra is elsősorban egy tanuló projekt volt. Mivel pénzt nem kaptam érte, szabadon kísérletezhettem az Azure szolgáltatásaival, és így is egy abszolút működő és használható alkalmazást raktam össze, amit még tervezek fejleszteni és optimalizálni a jövőben. Ezzel szemben, ha az eddigi ismereteimre alapozva, már sokat használt technológiákkal rakom össze ezt az alkalmazást, jóval kevesebbet tanultam volna. Lehet, hogy ebben a projektben nem volt tökéletes választás a serverless, de a jövőben még biztosan tudom kamatoztatni ezt a tudást.

7 A projekt jövője

Az 1.1.2-es fejezetben szereplő specifikáció még nem fedi le a Szövetség minden igényét, azonban úgy gondoltam, hogy ezen tárgy keretében ennyi fog beleférni. Mielőtt az alkalmazás a tényleges felhasználók kezébe kerül, mindenképpen el kell készíteni egy oldalt, ahol azon versenyek látszódnak, melyek értékelése már lezárult. Itt meg lehetne tekinteni az értékelés eredményét, hogy az egyes szempontokra átlagosan milyen értékelések érkeztek, esetleg az értékelők korcsoportja és szerepköre függvényében. Továbbá a Szövetség mindenképpen szeretné, ha az értékelések beérkezése után azok ellenőrizve lennének. Például, ha valaki versenyzőként értékelt, akkor meg kellene keresni az eredménylistában, hogy valóban elindult-e a versenyen. Ennek azonban nem tudok nekiállni, amíg az MTFSZ nem bővíti a saját API-ját egy olyan végponttal, ahol egy verseny eredménylistáját tudnám lekérni.

Ezekén kívül szeretném még a projektet teljesítmény és UX szempontból javítani. Mivel a szakdolgozat témája a serverless fejlesztés volt, a frontenden a dizájnolás és a felhasználói élmény javítása háttérbe szorult. Ezek azonban nagyon fontosak egy ilyen alkalmazásnál, ha azt szeretném, hogy a tájfunók rendszeresen visszajárjanak és mindig értékeljék azokat a versenyeket, amiken részt vettek.

A teljesítményen elsősorban úgy szeretnék javítani, hogy tüzetesebben megvizsgálom az Azure SQL nem serverless opcióit. Ugyanis itt kínálnak konstans teljesítményű adatbázis szervereket is már nagyon alacsony ártól. Mivel a serverless adatbázis skálázódását nem tudtam kihasználni, a cold-start pedig nagy hátrányt jelentett, valószínűleg ez a megoldás jobb teljesítményt nyújtana, ráadásul olcsóbban is. Havi 5 dollártól 73 dollárig terjedő skálán van 3 olyan lehetőség, amiket inkább csak hobbi projektekre, tesztverziókra ajánlanak, azonban az alacsony forgalmam miatt lehet, hogy ezek is elegendőek lennének ehhez a projekthez. A vCore típusú, serverless adatbázisokhoz hasonlítva ezek nem érik el az egy vCore teljesítményét, azonban ahogy az 6.2-es fejezetben láttuk, az én alkalmazásomnak se volt erre szüksége. [8] A legolcsóbb olyan lehetőség, amit már *production* alkalmazásokhoz is ajánlanak, havi 147 dollárba kerül, ami még épphogy beleférne a keretbe, hiszen így nem lenne cold-start, ezért a Redis gyorsítótárra se lenne szükségem. Ezeknek a tesztje és kipróbálása sem fért bele ebbe a projektbe, hiszen ez már nem serverless technológia, de első rálátásra jobban illik ezen alkalmazás igényeihez.

8 Konklúzió

A projekt aktuális állapota a <https://pontozo.mtfsz.hu> linken érhető el. Mivel a télen nincsenek tájfutó versenyek, pár tesztversenyt beillesztettem az adatbázisba, amin kipróbálható az értékelés. Ehhez azonban be kell jelentkezni, ami csak az MTFSZ nyilvántartásában szereplő tájfutóknak elérhető.

A projekt fejlesztése egy nagyon érdekes és tanulságos időszak volt számomra. Mindig szívesen álltam neki a munkának, mert tudtam, hogy egy tényleges problémát megoldó alkalmazást készítek, amit sokan fognak használni. Emellett az is motivált, hogy megismerjem jobban az Azure szolgáltatásait és kiderüljön, mennyire volt jó döntés ezeket használni ebben a projektben. Habár az eredmény nem lett teljesen pozitív, a projektet mindenképpen sikernek könyvelem el, hiszen nagyon sokat tanultam belőle, és az elkészült alkalmazás a jelenlegi formában is teljesen használható, de minimális javításokkal még jobb is lehet.

Irodalomjegyzék

- [1] *About the shadow database.* (dátum nélk.). Forrás: Prisma Docs: <https://www.prisma.io/docs/concepts/components/prisma-migrate/shadow-database>
- [2] *Application Insights overview.* (2023. október 20). Forrás: Microsoft Learn: <https://learn.microsoft.com/en-us/azure/azure-monitor/app/app-insights-overview>
- [3] *Authorization Code Flow with Proof Key for Code Exchange (PKCE).* (dátum nélk.). Forrás: Auth0: <https://auth0.com/docs/get-started/authentication-and-authorization-flow/authorization-code-flow-with-proof-key-for-code-exchange-pkce>
- [4] *Azure SQL Database pricing.* (dátum nélk.). Forrás: Microsoft Azure: <https://azure.microsoft.com/en-us/pricing/details/azure-sql-database/single/>
- [5] *Beginner's Guide to React Query.* (2023. július 4). Forrás: Refine: <https://refine.dev/blog/react-query-guide/>
- [6] Bharadwaj, M. (2023. szeptember 27.). *Azure Functions: Node.js v4 programming model is Generally Available.* Forrás: Microsoft Tech Community: <https://techcommunity.microsoft.com/t5/apps-on-azure-blog/azure-functions-node-js-v4-programming-model-is-generally/ba-p/3929217>
- [7] Chavan, Y. (2022. október 27). *How to Create Forms in React using react-hook-form.* Forrás: freeCodeCamp: <https://www.freecodecamp.org/news/how-to-create-forms-in-react-using-react-hook-form/>
- [8] *DTU-based purchasing model overview.* (2023. július 13). Forrás: Microsoft Learn: <https://learn.microsoft.com/en-us/azure/azure-sql/database/service-tiers-dtu?view=azuresql>
- [9] *Estimating Consumption plan costs.* (2023. február 17). Forrás: Microsoft Learn: <https://learn.microsoft.com/en-us/azure/azure-functions/functions-consumption-costs?tabs=portal>
- [10] *How to configure monitoring for Azure Functions.* (2023. július 23). Forrás: Microsoft Learn: <https://learn.microsoft.com/en-us/azure/azure-functions/configure-monitoring?tabs=v2>
- [11] *Introduction to Redis.* (dátum nélk.). Forrás: Redis Docs: <https://redis.io/docs/about/>
- [12] *JavaScript data types and data structures.* (2023. szeptember). Forrás: MDN web docs: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Data_structures
- [13] Kannan, B. (dátum nélk.). *Simple Introduction to Chakra UI.* Forrás: Bharathi Kannan Blog: <https://www.bharathikannan.com/blog/simple-introduction-to-chakra>

- [14] *OAuth Client Credentials Flow*. (dátum nélk.). Forrás: Curity:
<https://curity.io/resources/learn/oauth-client-credentials-flow/>
- [15] *Overview of React.js*. (dátum nélk.). Forrás: patterns:
<https://www.patterns.dev/react/>
- [16] *Relational vs. NoSQL data*. (2022. július 4.). Forrás: Microsoft Learn:
<https://learn.microsoft.com/en-us/dotnet/architecture/cloud-native/relational-vs-nosql-data>
- [17] Rosencrance, L. (2022. november). *Serverless computing*. Forrás: TechTarget:
<https://www.techtarget.com/searchitoperations/definition/serverless-computing>
- [18] *Try Azure SQL Database for free (preview)*. (2023. október 10). Forrás: Microsoft Learn: <https://learn.microsoft.com/en-us/azure/azure-sql/database/free-offer?view=azuresql>
- [19] *What is Azure API Management?* (2023. szeptember 1.). Forrás: Microsoft Learn: <https://learn.microsoft.com/en-us/azure/api-management/api-management-key-concepts>
- [20] *What is Azure Static Web Apps?* (2023. április 24). Forrás: Microsoft Learn: <https://learn.microsoft.com/en-us/azure/static-web-apps/overview>