



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

Perlaki Tamás

IRC CHAT BOT ÉS FELÜGYELETI WEB ALKALMAZÁS FEJLESZTÉSE MODERN JAVASCRIPT ESZKÖZÖKKEL

KONZULENS

Dr. Asztalos Márk

BUDAPEST, 2017

Tartalomjegyzék

Összefoglaló	7
Abstract.....	8
1 Bevezetés	9
2 Specifikáció	10
2.1 Funkciók.....	10
2.1.1 Parancsok	11
2.1.2 Szavazás	11
2.1.3 Nyereményjáték	11
2.1.4 Események	12
3 Technológiák, eszközök, keretrendszerek	13
3.1 TypeScript.....	13
3.1.1 Fordító	14
3.1.2 Statikus Típusosság.....	14
3.1.3 Nyelvi elemek	16
3.1.4 Integrált fejlesztő környezet.....	16
3.2 Angular 2+	17
3.2.1 Architektúra	17
3.2.2 Angular Material 2	19
3.2.3 Angular Flex Layout	20
3.2.4 Angular CLI	20
3.3 Node.js	21
3.3.1 npm.....	22
3.3.2 Express.js	22
3.4 MongoDB.....	23
3.4.1 Mongoose.....	23
3.5 Redis.....	24
4 Twitch.....	25
4.1 IRC	25
4.2 Oauth 2	26
5 Fejlesztési folyamat	28
5.1 Környezeti változók	28

5.1.1 Szerver oldal	28
5.1.2 Kliens oldal	29
5.2 Lokális fejlesztés.....	30
5.3 Távoli, felhő környezet	31
5.3.1 GitHub.....	32
5.3.2 Travis	32
5.3.3 Heroku.....	33
5.3.4 Adatbázisok.....	34
6 Az alkalmazás felépítése	35
6.1 Architektúra	35
6.2 Adatstruktúrák.....	36
6.2.1 Mongoose sémák	36
6.2.2 Redis típusok.....	39
7 Funkciók	41
7.1 Fogadó oldal.....	41
7.2 Bejelentkezett felhasználók felületei	44
7.3 Központi oldal (Dashboard).....	45
7.3.1 Események (Events).....	45
7.3.2 Szavazás (Poll).....	48
7.3.3 Nyereményjáték (Raffle).....	53
7.4 Parancs felület (Command).....	56
7.4.1 Parancsok (Commands)	56
7.4.2 Parancsidőzítók (Timers)	60
7.4.3 Parancs fedőnevek (Aliases)	64
7.4.4 Kommunikáció a komponensek között.....	66
8 Tesztelés	68
8.1 Unit tesztelés	68
8.2 End-to-end tesztelés	69
9 Biztonság	70
9.1 Angular.....	70
9.1.1 Cross-Site Scripting (XSS)	70
9.1.2 Cross-Site Request Forgery (CSRF)	71
9.2 Express	72
9.2.1 Transport Layer Security (TLS).....	72

9.2.2 Helmet	73
9.2.3 Munkamenet süti.....	73
9.2.4 Alkalmazás függőségek	74
9.2.5 Cross-Site Request Forgery (CSRF)	76
9.3 Injection	77
9.3.1 MongoDB	78
9.3.2 Redis.....	79
10 Összefoglalás.....	81
Irodalomjegyzék.....	82
Függelék.....	85

HALLGATÓI NYILATKOZAT

Alulírott **Perlaki Tamás**, szigorló hallgató kijelentem, hogy ezt a diplomatervet 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, 2017. 12. 08.

.....
Perlaki Tamás

Összefoglaló

A JavaScript, mint a böngészők uralkodó programozási nyelve, az elmúlt, bő egy évtizedben rohamos fejlődésnek indult. Immár nem csak kliens, de szerver oldalon is jelen van, és sorra jelennek meg hozzá a különböző, széleskörűen használt eszközök, keretrendszerek. Ahhoz, hogy a szakmai életben megállhassuk a helyünket és versenyképes tudásra tegyünk szert, szükséges ezeknek az új eszközöknek a megismerése és alkalmazása. A munkám célja tehát ezeknek a megismerése volt egy webes alkalmazás elkészítése segítségével, amely a népszerű Twitch élőadás közvetítő, „stream” platformhoz kapcsolódik.

A dolgozat először bepillantást enged a JavaScript fejlődésének egyik irányába, vagyis a TypeScript programozási nyelv és az Angular keretrendszer ökoszisztémájába. Ezek után bemutatja a JavaScript egyik legelterjedtebb szerver oldali futtató környezetét, a Node.js-t és a vele gyakran párba állított MongoDB adatbázis kezelőt, valamint a Redis adatbázist. A további részekben a dolgozat leírja az alkalmazás fejlesztésének folyamatát, felépítését és az elkészült funkciókat. Végül kitér az alkalmazás tesztelési megoldásaira, illetve a biztonsági óvintézkedéseire.

Abstract

JavaScript, as the lead programming language of browsers, has begun a rapid evolution in the last decade. At this time, it is not only present in the clients, but also on server side, and different, widely used tools and frameworks appear for it each day, one after another. To be able to keep up with the demands of professional life, and acquire a competitive knowledge, we need to get to know and use these new technologies. The goal of my work was to get familiar with these technologies, along the development of a web based application, which connects to the popular Twitch streaming platform.

The thesis first shows one of the directions of JavaScript's development, namely the ecosystem of the TypeScript programming language and the Angular framework. After that, it presents one of JavaScript's most popular server-side runtime environment, Node.js, its common partner, the MongoDB database management system, and the Redis database. In the next parts, the thesis writes about the process of the application development, its architecture, and the implemented features. At last, it presents the application's testing solutions and its security precautions.

1 Bevezetés

Az utóbbi időkben a webes klienssel rendelkező szoftverek egyre nagyobb teret nyernek az informatikán belül. Weblapok természetesen évtizedek óta készülnek az internet számára, de a böngészők fejlődésének köszönhetően egyre nagyobb, komplexebb, intelligensebb rendszerek születhetnek. Ennek a fejlődésnek azonban megvan az a hátránya, hogy a növekvő bonyolultságú alkalmazások fejlesztésére és karbantartására újabb, modernebb eszközöket és technológiákat kell használnunk, hogy kezelni tudjuk azok rohamosan növekvő méreteit. Ezeket az új eszközöket, tapasztalataim szerint, az iparban tevékenykedő vállalatok is előszeretettel alkalmazzák, ezért megismerésük létfontosságú ahhoz, hogy valaki a szoftverfejlesztés ezen ágában tudjon elhelyezkedni.

A webes klienssel rendelkező alkalmazásoknak általában szükségük van egy szerver oldalra. Az utóbbi évtizedben a böngészők uralkodó nyelve, a JavaScript elterjedt a szerverek között is és egyre szélesebb körbe használják, ezért ennek technikai megoldásainak megismerése szintén igen fontossá vált. Végül a többretegű modell legalsó rétege, az adatrétegről is meg kell említenünk, hogy a hagyományos SQL adatbázisok mellett egyre nagyobb teret nyernek a NoSQL megoldások.

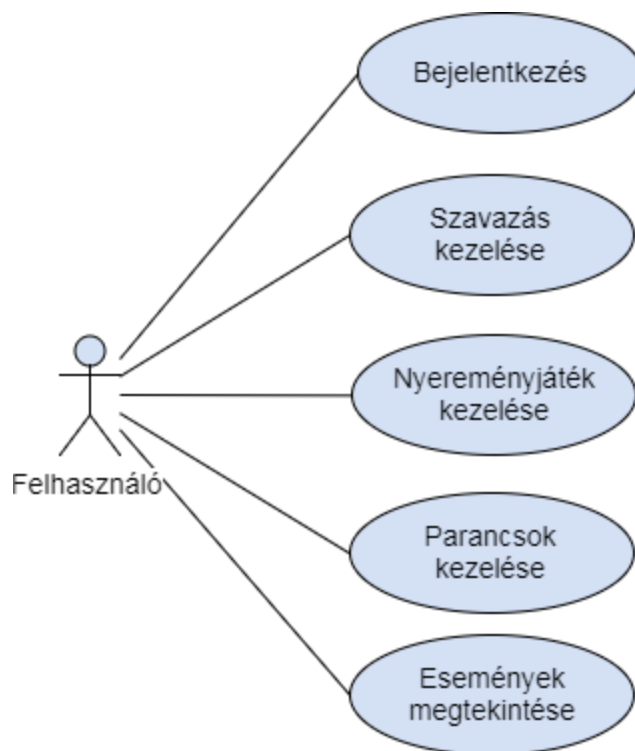
Munkám során a fent említett újdonságokba igyekeztem beleásni magam egy többretegű alkalmazás elkészítésével. Az alkalmazás szorosan kapcsolódik a Twitch [1] élő műsorközvetítő oldalhoz, amin főleg számítógépes játékokról szóló adások sugároznak, különböző csatornákon. Minden közvetítő csatornához tartozik egy-egy chat szoba is, amelyekben több ezer, vagy nem ritkán több tízezer felhasználó is tartózkodhat egyszerre, és ezeknek a csatornáknak a moderációs és egyéb munkáit gyakran chat botok segítségével végzik. Az elkészített alkalmazás tehát egy IRC [2] chat bot felügyeleti rendszer, vagyis egy olyan webes alkalmazás, aminek a segítségével a leírt chat-es robotokat lehet irányítani, konfigurálni.

A dolgozat a specifikáció megfogalmazása után leírja azokat az eszközöket, illetve technológiákat, amelyekkel az alkalmazás elkészült. Ezek után a fejlesztési folyamat leírása, az alkalmazás felépítése és az elkészült funkciók leírásai következnek. Végül a dolgozat kitér az alkalmazás tesztelésének módjaira, illetve a használt biztonsági megoldásokra.

2 Specifikáció

A feladat egy chat robotok kezelésére és irányítására szolgáló, webes alkalmazás elkészítése. Az alkalmazásnak csatlakozni kell tudnia a Twitch platform által üzemeltett, IRC protokollt használó chatrendszerhez, robotok, vagyis röviden bot-ok segítségével. A chat rendszerbe becsatlakozott bot-ok a felhasználók által konfigurált módon, meghatározott műveleteket képesek elvégezni. A rendszerbe a felhasználók kötelezően a Twitch rendszerben létrehozott felhasználói fiókjaikkal léphetnek be. A felhasználók első belépésekor létrejön az alkalmazás adatbázisában a felhasználó profilja a beállításainak tárolására, illetve elkészül számára egy külön bot példány, ami automatikusan csatlakozik a felhasználó Twitch fiókjához kapcsolódó chat szobába. A felhasználó az első belépés után megváltoztathatja a személyes chat bot-jának a beállításait, ezzel befolyásolva a viselkedését, illetve néhány azonnali akciót is végrehajthat vele.

2.1 Funkciók



1. ábra Felhasználói használati esetek

2.1.1 Parancsok

A botok egyik alapvető része a parancsok kezelése, tehát létrehozása, megtekintése, szerkesztése, törlése. A parancsok olyan szövegek, amiket a bot a chatre ír be, amikor a felhasználók aktiválják azt. Két szövegből állnak tehát, egyik az, amit a felhasználóknak kell a chatre írniuk, hogy a bot végrehajtsa a parancsot, a másik pedig az a szöveg, amit a bot válaszul ír a parancs kiadásakor.

A parancsokhoz kapcsolódnak az időzítők. Az időzítő parancsok, névvel ellátott halmazai, amikhez egy időzítő értéket rendelünk. Az időzítők az értékben megadott időközönként lefuttatják az összefoglaló parancsokat. A felhasználók az időzítőket létrehozhatják, szerkeszthetik, törölhetik, listázhatják.

A parancsok második kiegészítője a fedőnevek. Egy fedőnév a parancshoz rendelt kiegészítő név, arra ad lehetőséget, hogy egy parancsra több néven is hivatkozzunk, és ne kelljen ugyanazzal a hatással több különböző parancsot is létrehoznunk. A fedőnevek egy egyszerűbb funkció, ezért szerkesztésük nem lehetséges, csupán létrehozásuk, listázásuk és törlésük.

2.1.2 Szavazás

Az alkalmazásba bejelentkezett felhasználóknak lehetőségük van a chat-en lévő felhasználók közötti szavazás lebonyolítására. A szavazás megnyitásához a csatorna tulajdonosnak meg kell adnia azokat a lehetőségeket, amire a felhasználók szavazhatnak. A felhasználók a szavazás megnyitása után chatüzenetek segítségével tudnak szavazni a lehetőségek közül. A csatornatulajdonos a szavazás állását egy kördiagram segítségével tudja figyelemmel kísérni. Lehetőség van a szavazás újrakezdésére, aminek keretében törlődnek az aktuális eredmények, de a szavazás nem zárul le, illetve le lehet a szavazást teljesen is zárni, ami után a szavazatokat nem veszi figyelembe a rendszer.

2.1.3 Nyereményjáték

A nyereményjáték szintén a chaten keresztül zajló művelet, ahol a felhasználók közül tud a csatornatulajdonos nyerteseket sorsolni. A nyereményjáték első fajtája a jelentkezésese játék, ami azután indul, hogy a csatornatulajdonos megnyitotta. A megnyitás után a felhasználók chat üzenetekkel tudják jelezni a szándékukat, hogy részt szeretnének venni a sorshúzáson. Mikor a csatornatulajdonos úgy gondolja elegendő

jelentkeztek a játékra, egy gombnyomással véletlenszerűen sorsolhat közülük egy vagy több felhasználót, amíg a játékot le nem zárja. A nyereményjáték másik fajtája az azonnal sorsolás, amikor nem szükséges a játékra jelentkezni, a chaten tartózkodó összes felhasználó közül fog a rendszer választani.

2.1.4 Események

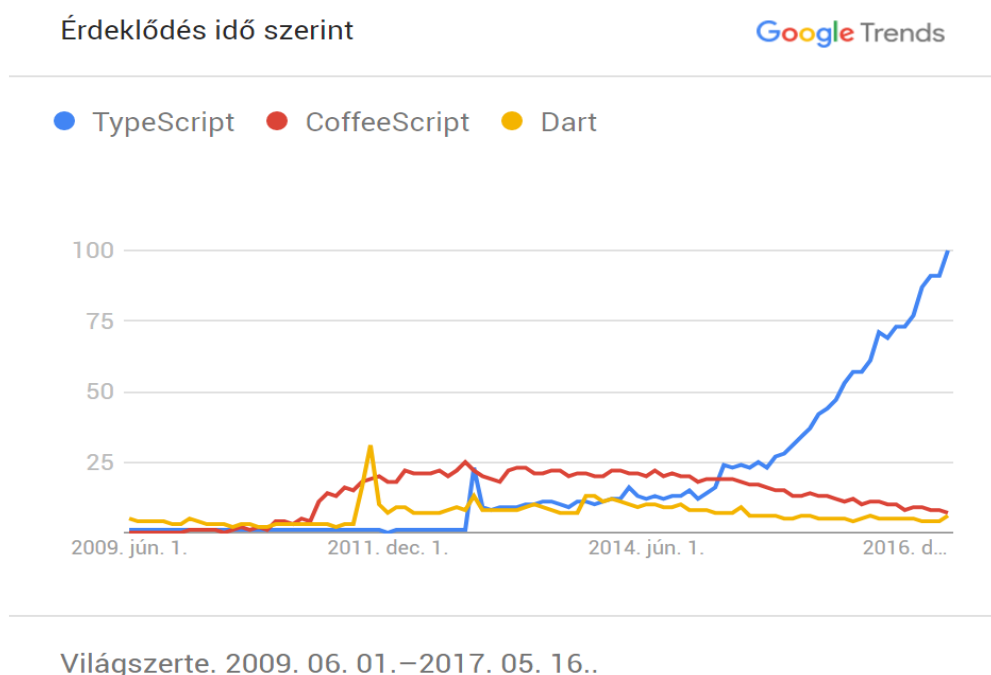
A felhasználó és bot eseményekről a rendszer naplót vezet a csatornatulajdonos részére. Itt olyan események szerepelhetnek, mint például a nyereményjáték megnyitása, vagy egy új parancs hozzáadása a bothoz. A felhasználó az eseményeket egy külön mezőben tudja megtekinteni és azok valós időben frissülnek egy művelet elvégzésekor.

3 Technológiák, eszközök, keretrendszerek

A munkám során első lépésként megismerkedtem azokkal a technológiákkal, amelyek segítségével később a tényleges alkalmazás elkészült.

3.1 TypeScript

A TypeScript [3] egy viszonylag újonnan (2012 végén) megjelent open source programozási nyelv, amelyet a Microsoft kezdett fejleszteni és azóta is a cég felügyeli. A nyelv célja, hogy kibővítse a JavaScript [4] által nyújtott lehetőségeket, illetve megkönnyítse a fejlesztők munkáját mind a szerver oldalra, mind a kliens oldalra szánt programok elkészítésében, főleg nagyméretű, nehezen karbantartható kódbázisoknál. A TypeScript a JavaScript bővített halmaza, tehát minden, ami megengedett JavaScript-ben az TypeScript-ben is legális és elérhető. Ezen kívül a nyelv további nyelvi eszközöket és egyéb támogatásokat ad a JavaScript repertoárjához, mint például típusok, interface-ek, statikus kódelemzés. Továbbá érdemes megemlíteni, hogy egyebek mellett a későbbiekben bemutatásra kerülő, Google által fejlesztett Angular 2+ keretrendszer egyik támogatott nyelve, ami manapság az egyik legelterjedtebb webes rendszer, tehát látható, hogy a nyelv komoly támogatottságra tett szert.



2. ábra A Typescript népszerűségének növekedése

3.1.1 Fordító

A nyelv remek tulajdonságai és újításai önmagukban azonban még nem lennének elegendőek, hiszen a böngésző motorok valószínűleg nem kezdenének egy új nyelvet támogatni a meglévő JavaScript helyett, legalábbis csak hosszú idő és bonyodalmak után. A nyelv készítői ezt úgy oldották meg, hogy a JavaScript-ben nem, de TypeScript-ben létező elemeket egy fordító segítségével érvényes JavaScript kódra alakítják. Mivel azonban a különböző típusú és verziójú JavaScript motorok is más-más elemeket támogatnak, ezért a TypeScript a JavaScript-et szabványosítani igyekvő EcmaScript [5] szabványokhoz igazodik. Azt, hogy az EcmaScript hányas verziójára szeretnénk fordítani a programunkat a fordító konfigurálásával tudjuk szabályozni, így régebbi és újabb verziójú böngészőket, vagy akár szerver oldali motorokat is tudunk támogatni egyszerre. Ehhez hozzá tartozik, hogy maga a TypeScript nyelv pedig mindig igyekszik a legújabb EcmaScript szabványoknak megfelelni, amihez hozzá teszi a saját kiegészítéseit.

3.1.2 Statikus Típusosság

A nyelv talán legkomolyabb újítása azon kívül, hogy az újabb EcmaScript elemeket elérhetővé teszi régebbi JavaScript motorokon, az a statikus típusok bevezetése. A JavaScript-nek, mint interpretált és dinamikus típusokkal rendelkező nyelvnek nagy hátránya, hogy a hibáink jó része csak futtatási időben tud kiderülni, ami adott esetben lényegesen le tudja lassítani a fejlesztést. Ezzel szemben a TypeScript-ben opcionálisan használható statikus típusok segítségével az elkövetett hibák jó része egy arra alkalmas IDE segítségével, még a fordítás megkezdése előtt kiderülhet a statikus kódelemzés által. Például megadhatjuk egy függvényünknek, hogy hány darab és milyen típusú paramétert vár. Ez JavaScript esetében semmilyen garanciát nem jelentett, valójában bármilyen paraméterrel meg lehet hívni egy függvényt és a hiba a kód lefutásakor keletkezett, sőt, előfordulhatott, hogy a hiba csak néha jelentkezett, így nehezítve a javítását. Ezzel szemben a TypeScript legkésőbb fordítási időben megállít minket, ha rossz paramétereket adunk meg, ezzel jelentősen gyorsítva a fejlesztési folyamatot.

```
main.ts
1 class Person {
2   name: string
3   age: number
4 }
5
6 [ts] Supplied parameters do not match any signature of call target.
7
8 constructor Person(): Person
9 let p = new Person(10, 20);
```

3. ábra Hibás paraméterek jelzése

A hibák korai detektálásán kívül még egy nagy előnye a statikus típusok használatának az intelligens kódkiegészítés (IntelliSense, Intelligent code completion), amely szintén a fejlesztők munkáját gyorsítja. A technológia lényege, hogy az arra alkalmas szerkesztőkben a kódunk írása közben felugró ablakokban segítséget kapunk többek között típusos változók, függvényparaméterek használatához, ezáltal pedig a Java, vagy C# világhoz hasonló fejlesztői élményben lehet részünk.

```
7 var cookieParser = require('cookie-parser');
8 var bodyParser = require('body-parser');
9 var http = require('http');
10 http.
11 Agent
12 var r createClient
13 var u createServer (function) http.createServer(requestListener?: (request:
14 get
15 var a globalAgent
16 request
17 // vi STATUS_CODES
18 app.set('views', path.join(__dirname, 'views'));
19 app.set('view engine', 'jade');
```

4. ábra Intelligens kódkiegészítés

A statikus kódellenzés további nagy előnye, hogy nemcsak az általunk írt kódot képes elemezni, hanem a használt, akár JavaScript-ben megírt külső könyvtárakra is elérhetővé tehető. Ehhez mindössze egy típusdefiníciókat tartalmazó csomagot kell telepítenünk a projektünkben, amit azután az arra alkalmas fejlesztőeszközök képesek használni és elérhetővé tenni általuk a korábban említett szolgáltatásokat.

3.1.3 Nyelvi elemek

Ahogy említettem, a TypeScript másik nagy újítása az újabb EcmaScript szabvány elemek elérhetővé tétele és saját nyelvi elemek bevezetése. Ezekből néhány fontosabbat említek meg, amelyek véleményem szerint a jelenben és a jövőben is nagyban meghatározzák a fejlesztések irányvonalát.

Az első nagyon komoly változás a JavaScript világban bukkant fel, még jóval az EcmaScript általi szabványosítás, és a TypeScript megvalósítás előtt, ez pedig a modulok támogatása. A modulok kisebb, önálló egységet képező JavaScript kódrészletek, amiket azután importálni lehet egymásba, így karbantarthatóbb, átláthatóbb, újrafelhasználható kódot eredményezve. A modul alapú fejlesztésre rengeteg megoldás áll a fejlesztők rendelkezésére, ezeknek mind a saját előnyeivel és hátrányaival. Ezt próbálja meg kiküszöbölni az EcmaScript 2015-ös verziója, ami egy egységes, szabványos megoldást próbál meg nyújtani a modulok kezelésére, amit a TypeScript is támogat. Ez TypeScript fejlesztés esetén abból áll, hogy minden fájl egy modul. Egy modul ezután exportálhat tetszőleges számban függvényeket, osztályokat, interfészeket, amiket ezután más modulokban importálhatunk és használhatunk. Importálhatunk fájlrendszer elérés alapján, a számítógépünkön elhelyezett, általunk írt modulokat, illetve név alapján, telepített külső csomagokat is.

Az EcmaScript 2015-ös verziójában jelent meg szintén az osztály, mint nyelvi elem, amellyel a JavaScript korábbi nehézkes, nehezen átlátható öröklődési rendszerét igyekeztek könnyebben használhatóvá tenni. Ehhez kapcsolódóan vezette be a TypeScript az interfészeket, illetve az interfészek és osztályok más nyelvekben megszokott megvalósításon alapuló hierarchiáját. A két nyelvi elem és a korábban említett statikus típusosság segítségével ezek után pedig ténylegesen a Java, vagy C# világban megszokott fejlesztési élményhez hasonlóan lehet részünk.

3.1.4 Integrált fejlesztő környezet

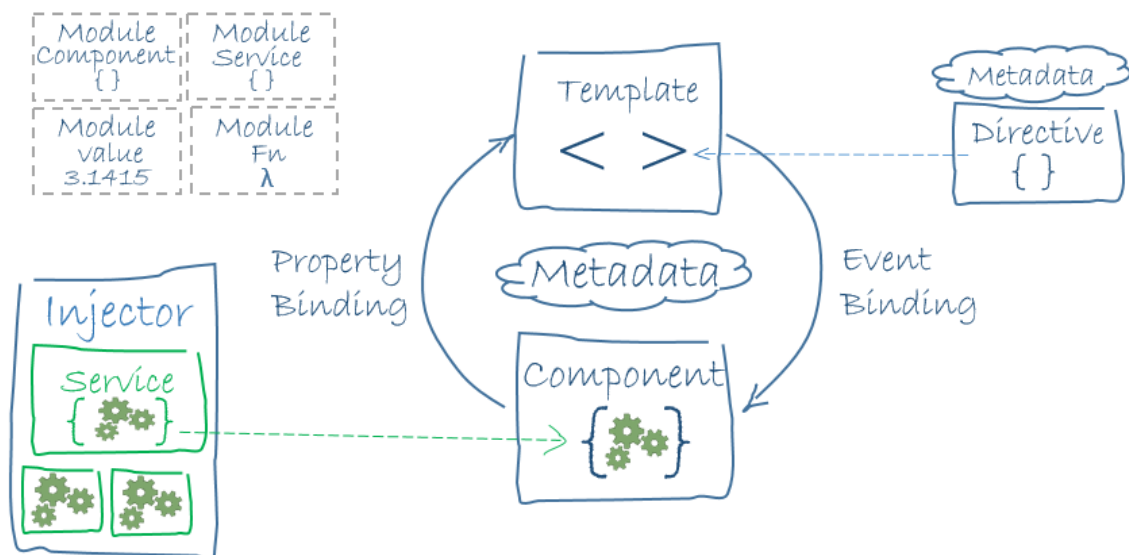
A TypeScript fejlesztését támogatják a ma legelterjedtebb IDE-k, például a Visual Studio [6], Webstorm [7] és az Eclipse [8]. A fejlesztés során egy szintén újonnan (2015 első felében) megjelent eszközt használtam, a Visual Studio Code-ot [9]. Ez egy open source fejlesztő környezet, ugyancsak a Microsoft gondozásában, ami erősen támogatja a web fejlesztéséhez szükséges nyelveket és az ahhoz kapcsolódó elterjedtebb eszközöket és megoldásokat. Ilyenek például a git [10] verziókezelővel

való integráció beépített parancssor, vagy a JavaScript debugger. Érdekessége még, hogy Electron-ban [11] fejlesztették, ami egy JavaScript-re, HTML-re és CSS-re épülő, asztali ablakos környezet fejlesztésére használható keretrendszer. Ennek az eszköznek a segítségével lehetett tehát igazán kihasználni a TypeScript által nyújtott lehetőségeket a gyorsabb, egyszerűbb és átláthatóbb kódfejlesztésre.

3.2 Angular 2+

Az Angular 2+ [12] egy a Google által kezdeményezett és felügyelt, open source, kliens oldali webes keretrendszer, elődjének, az AngularJS keretrendszernek a teljes újragondolásra és újraimplementálása. Model-View-Controller és Model-Viewmodel-Model architektúrája Single Page Application (SPA) weboldalak fejlesztését teszi lehetővé. Ez azt jelenti, hogy egyetlen kezdő HTML oldal kiszolgálása után a böngészőben futó JavaScript program lesz az, ami az alkalmazás további részeit lekéri a szervertől, ellentétben a hagyományos modellel, ahol minden navigáció a szerver egy újabb HTML oldal betöltésével jár. A keretrendszer elsődlegesen támogatott nyelvei a JavaScript és a TypeScript, tehát fejlesztés közben a TypeScript minden eddigi előnye élvezhető.

3.2.1 Architektúra



5. ábra Az Angular keretrendszer architektúrája

3.2.1.1 Nézetek

Egy Angular alkalmazás nézetek (View) hierarchájából áll, egy kitüntetett gyöker nézettel. Minden nézet az oldal egy részének megjelenítéséért felel. Az oldal megnyitásakor a gyöker nézet fog először betöltődni, ami azután felépíti a további nézetek hierarcháját, amikből az oldal áll, és amik között a felhasználók navigálhatnak. Egy nézet három alkotóeleme a sablon, a komponens és a hozzá tartozó meta adatok.

A sablon egy egyszerű HTML fájlból, illetve opcionálisan több CSS fájlból áll. A HTML fájlban az Angular által definiált módon elhelyezhetünk speciális elemeket, attribútumokat, amiket aztán a keretrendszer értelmezni fog számunkra és átalakítja a megfelelő módon, a böngészők által is megérthető kimenetre. Ilyen lehet például hivatkozás egy másik nézetre, ami aztán a mi nézetünkbe fog beágyazódni, vagy adatkötések, amivel a kódunkban lévő változók értékeit tudjuk megjeleníteni.

A nézeteink mögött álló logikát (ViewModel / Controller) komponensek segítségével tudjuk megvalósítani. Ezek egyszerű TypeScript osztályok, amelyeknek a feladata a nézet vezérlése, adatok nyújtása a sablonnak, és a tőle érkező események kezelése. Illetve szintén ezen az osztályon keresztül történik a modellel való kapcsolat megteremtése.

A sablonok és a mögöttük álló komponensek adatkötések segítségével tartják a kapcsolatot. Ennek három fajtáját használhatjuk:

- Egy irányú: A komponensben lévő adatok megjelenítése a sablonban, ami a komponensben lévő változás hatására automatikusan frissíti a sablont, ellenben a sablonban történő változásra a komponens nem frissül.
- Eseménykötés: A sablonban történő eseményekre, mint például kattintás feliratkozhat a komponens eseménykezelő függvényekkel és reagálhat rájuk.
- Két irányú: A komponensben lévő adatok megjelenítése a sablonban, ami a komponensben lévő változás hatására automatikusan frissíti a sablont, és a sablonban történő változásra a komponens is frissül.

3.2.1.2 Szolgáltatások

Az Angular terminológiája szerint a szolgáltatás egy elég tág fogalom, lényegében bármi lehet szolgáltatás: egy érték, egy függvény, de tipikusan egy osztály. Egy jól felépített alkalmazásban szolgáltatások adják a modellt, az adatelérést és az üzleti logikát, konfigurációt és bármi egyebet, amire az alkalmazásnak szüksége lehet. A szolgáltatásokat tipikusan komponensekben és más szolgáltatásokban használjuk, legtöbbször függőség injektálás segítségével. A függőség injektálás szétválasztja a logikáinktól a függőségek megszerzésének a módját, ami számos előnnyel jár, például elősegíti az automatizált tesztelést és az elemek lazább csatolását. Ahhoz, hogy egy szolgáltatásunk injektálható legyen egy úgynevezett provider-t kell létrehozunk, ami tartalmazza a szolgáltatás előállításának módját. Ezután a provider-ünket be kell regisztrálnunk egy, a következő pontban ismertetett, modulba, vagy abba a komponensbe, ahol használni szeretnénk, és innentől használhatjuk is.

3.2.1.3 Modulok

Komponenseinket, szolgáltatásainkat és egyéb építő elemeinket, a TypeScript modulokkal nem azonos Angular modulok segítségével tudjuk definiálni. Ezek, a nézetekhez hasonlóan, hierarchiába szervezhetőek. Használatukkal egy átláthatóbb, karbantarthatóbb kódbázist tudunk létrehozni.

3.2.2 Angular Material 2

A Material Design [13] a Google által 2014-ben bejelentett „vizuális nyelv”. Egy élő, folyamatosan fejlesztett specifikáció arról, hogyan kell a platformok között egységesen kinéző, letisztult kinézetet adni az alkalmazásoknak. Definiálja mi az, ami egy modern, a specifikációt követő alkalmazás felhasználói felületéhez szükséges a stílusoktól és elrendezésektől kezdve, a komponenseken át a többnyelvűségig és akadálymentesítésig mindent.

A Material Design-hoz az Angular 2 keretrendszer fejlesztői maguk adnak támogatást egy külön projektben [14], amiben megvalósítják a specifikációba foglalt irányelveket és létrehozzák a meghatározott komponenseket, animációkat, stílusokat. A dolgozat írásának idején a projekt még béta állapotban van, de már használható, nagyon sok komponens elérhető az oldal szerkezetének kialakításához, űrlapok elkészítésére, navigációra. A végső alkalmazás felülete tehát nagyrészt ennek a projektnek a segítségével, a Material Design elveknek megfelelően lett kialakítva.

3.2.3 Angular Flex Layout

A flexible box [15] a CSS3-ban megjelenő új elrendezés típus. A korábbi, CSS-sel nem, vagy csak nagyon nehezen, trükközésekkel megoldható fejlesztési problémákra igyekszik választ adni, a korábbi „block” és „inline” elrendezések kiegészítéseként. Főleg a korábban nehézkesen megoldható függőleges és vízszintes igazításokra próbál egyszerű választ adni, illetve az elemek reszponzív megjelenítését is nagyban támogatja. Működésének alapja, hogy kijelölünk egy flex konténer elemet, aminek gyerekelemei fognak az elrendezésben részt venni. A konténer elemnek meg kell adnunk egy elrendezési irányt: vízszintes, vagy függőleges, ez lesz a fő irány, a másik pedig a keresztirány. Ezek után megadhatunk a konténeren elrendezési irányonként minden belső elemre vonatkozó szabályokat, például, hogy a fő irányon minden elem a konténer elején legyen, a keresztirányon pedig nyúljanak ki a konténer teljes hosszában vagy széltében. A minden elemre vonatkozó szabályokon kívül megadhatjuk elemenként is, hogy hogyan viselkedjenek, például, hogy egy elem pontosan a fő irány 2/3-át foglalja el, vagy, hogy mennyire zsugorodjon, csökkenjen, ha újra kell számolni az arányokat, mondjuk az ablak újraméretezésekor. Az így megadott szabályok alapján a böngésző motor elrendezi az elemeket, a maradék területet pedig arányosan osztja fel. Így tehát egy, a korábbi elrendezési formáknál jóval rugalmasabb, és egységesebb elrendezést használhatunk az alkalmazásunkban.

A CSS3 flexbox elrendezést használva az Angular csapat egy saját, Flex Layout [16] nevű rendszert fejlesztett, ami az Angular Material-től független, de vele együtt használható. A projektnek a célja, hogy egy Angular-ból használható API-t nyújtson az alkalmazásunknak a flex elrendezés kezelésére a HTML sablonjainkban. Az API deklaratív és imperatív részekből áll, segítségével dinamikusan tudjuk alakítani az alkalmazásunkban a flex tulajdonságokat, illetve az alap reszponzivitást kiegészíti töréspontok bevezetésével, amelyeknél még jobban testre tudjuk szabni alkalmazásunk kinézetét.

3.2.4 Angular CLI

Az Angular CLI [17] egy hivatalos, az Angular csapat által készített Node.js [18], tehát szerver oldali JavaScript, program alkalmazások fejlesztésének támogatásához és gyorsításához. Segítségével elsőként komplett projekteket lehet inicializálni, ami azt jelenti, hogy az eszköz legenerálja nekünk a könyvtárstruktúrát,

elkészíti a konfigurációs fájlokat és beállítja a fejlesztéshez szükséges automatizált eszközöket, mint a TypeScript fordító. Egy létező alkalmazás kezeléséhez ezek után további támogatásokat nyújt, például új Angular elemeket lehet létrehozni, mint komponens, vagy modul, amiknek legenerálja a vázait, illetve a hozzájuk szükséges tesztfájlokat is. A generáláson kívül futtatható vele maga az alkalmazás is egy fejlesztői szerver segítségével, ami folyamatosan figyeli a fájljainkban történő változásokat és automatikusan frissíti a böngészőnket a fordítás lefutása után, ezzel gyorsítva a fejlesztést. Az eddigieken továbblépve használhatók vele CSS preprocesszorok, mint a LESS [19] és a SASS [20], futtathatók vele Karma [21] és Protractor [22] alapú automatizált tesztek, valamint statikus kódelemezést végző linter-ek, illetve előállíthatók vele a kiadásra szánt, végső fájlok is.

3.2.4.1 Webpack

A Webpack [23] egy úgynevezett „module bundler”, ami az egyik ajánlott eszköz Angular alkalmazások készítéséhez, amit az Angular CLI alapértelmezetten használ. Működésének lényege, hogy az alkalmazásunkat alkotó erőforrás fájlokat a bennük lévő „import” és „require” utasítások mentén összezsomagozza egy nagy csomaggá, „bundle”-lé, ezzel jelentősen csökkentve a szerverlekérések számát. Ami a Webpack nagy előnye, hogy nagymértékben testre szabható és bővíthető, a csomagok készítése mentén. Ilyen bővítés tipikusan a preprocessor-ok és fordítók alkalmazása, például az általam használt TypeScript és SCSS fájlok átalakítására JavaScript és CSS fájlokká.

3.3 Node.js

A Node.js [23] egy pehelykönnyű, szerver oldali, aszinkron eseményvezérelt JavaScript futtatókörnyezet, elsősorban hálózati alkalmazások, webkiszolgálók létrehozására. A platform magjában a Google által fejlesztett, Chrome böngészőben is futó V8 JavaScript motor van, amit a készítők felkészítettek a szerver oldali fejlesztések támogatására megfelelő API-k létrehozásával és egyéb kiegészítésekkel. A szerver oldali JavaScript nagy előnye a kliens oldallal szemben, hogy itt egyetlen verziót kell támogatnunk, nem pedig a különféle klienseken futó, különböző böngészők különböző motorjait. Ez, és az, hogy a V8 folyamatosan valósítja meg a legújabb EcmaScript szabványok által specifikált elemeket, lehetővé teszi, hogy a lehető legmodernebb JavaScript verzióval dolgozzunk. Hálózati alkalmazásokon kívül rengeteg egyéb

program is készíthető vele, például webes, vagy egyéb alkalmazások fejlesztését támogató eszközök is. Ilyen a Grunt és a Gulp programok, amik a fejlesztés közben folyamatosan ismétlődő feladatok elvégzésében segítenek minket, mint TypeScript fordítás, CSS preprocessálás, JavaScript fájlok tömörítése, obfuszkálása, egybefűzése.

3.3.1 npm

Az npm [24] a Node.js által használt függőség és csomagkezelő program. Segítségével a fejlesztők megoszthatják a JavaScript-ben írt kódjukat, amiket csomagok, vagy más néven modulok formájában tehetnek elérhetővé mások számára. A Java-s világ Mavenjéhez [25] hasonlóan a csomagok tipikusan egy központi tárolóban (repository-ban) vannak, amikre mi konfigurációs fájlok segítségével függőségeket fogalmazhatunk meg. Ezeket a függőségeket azután az npm segítségével kielégíthetjük, ami letölti számunkra a megfelelő JavaScript csomagokat, illetve a csomagok megfelelő verziókezeléséről is gondoskodik. A Node.js világban lényegében minden külső függőséget ennek a csomagkezelőnek a segítségével telepíthetünk, például az Angular-t is, illetve saját csomagot is készíthetünk, és regisztráció után feltehetjük a központi tárolóba.

3.3.2 Express.js

Az Express.js [26] egy pehelysúlyú keretrendszer Node.js platformra, amely a beépített API-ra építve könnyen bővíthető módon ad támogatást web alkalmazások fejlesztésére. Egy egyszerű Express alkalmazás nem áll másból, mint a webes elérés gyökerétől vett útvonalak, úgynevezett route-ok definíciójából, illetve az ezekre beregisztrált kezelő függvényekből. Minden útvonal egy olyan végpont, ami a megfelelő metódussal HTTP protokollon keresztül elérhető. Amikor például egy böngészőben megnyitnak egy adott URL-t, akkor az arra beregisztrált kezelők, úgynevezett middleware-k a beregisztrálás sorrendjében lefutnak, ezáltal egy láncot alkotva. Az egy láncba tartozó middleware-ek egy lekérés közben adatokat adhatnak át egymásnak, így a kérés feldolgozása több lépcsőn keresztül történhet, szétválasztva a logikát, elősegítve a karbantarthatóságot.

Az Express egyik előnyös tulajdonsága még az egyszerű bővíthetőség. Ha valami újabb tulajdonsággal akarjuk felruházni a szerverünket, akkor általában egyszerűen csak regisztrálnunk kell egy npm által telepített middleware-t az összes route-ra, ami innentől már képes is működni. Ilyen módon lehet telepíteni az

alkalmazásunkba például a session kezelést, biztonsági bővítményeket, vagy épp szerver oldali HTML generálást elősegítő sablon kezelést.

3.4 MongoDB

A MongoDB [27] egy dokumentum alapú NoSQL (not only SQL) adatbázis kezelő rendszer, tehát nem az informatikában hagyományosnak tekinthető relációs adatmodellre épül. Használatával félig strukturált, bináris JSON formátumú adatokat tárolhatunk, amiknek szerkezete nem kötött, könnyen változtatható. Tervezésének egyik központi célja a könnyű skálázhatóság és a nagy rendelkezésre állás volt, amiért viszont cserébe a relációs adatbázisokban jellemző Atomicitás, Konzisztencia, Izoláció, Tartósság tulajdonságokat részben feladták. Mindezek ellenére támogatja a megszokott lekérdezési módokat, aggregációkat, indexeket és az elterjedt programozási nyelvek mindegyikén hozzáférhető.

Adatmodellje nagyon egyszerű. Minden adatbázisban létrehozhatunk tetszőleges számú gyűjteményt (collection), ami azonos sémájú adatok gyűjteménye. Ez a relációs adatmodellben megszokott táblákkal analóg, azonban egy fontos különbség, hogy itt a dokumentumoknak nem kötelező a sémának megfelelni. A gyűjteményekben az adatok tárolásáért a dokumentumok felelnek. A dokumentumok között ezután kapcsolatok is felépíthetők kétféle módon. Az első a már megszokott relációk használata, vagyis a hivatkozni kívánt dokumentumok azonosítójának a tárolása. A másik egy újfajta módszer, a beágyazott dokumentumok használata, amivel jelentős teljesítmény javulást érhetünk el, ha sok olyan lekérdezést fogalmazunk meg, amiben több dokumentum is szerepel.

3.4.1 Mongoose

A mongoose [28] egy JavaScript könyvtár, amit MongoDB adatbázisok elérésére lehet használni. Habár a MongoDB önmaga is ajánl egy JavaScript API-t, ennek használata jóval egyszerűbb a mongoose segítségével, ami magába foglalja és kiegészíti az alap funkciókat. Segítségével könnyen létrehozhatunk például validátorokat, amikkel ellenőrizhetjük, hogy az adatbázisba beszúráskor az alkalmazás logikánknak megfelelő adatok kerülnek-e, hiszen a MongoDB-ben nincsenek beépített megkötései. Egy másik példa a „population” mechanizmus. Mivel a MongoDB-ben nincs a relációs adatbázisokban lévővel megegyező JOIN művelet, ezért ezt a műveletet

a mongoose a population-nel igyekszik helyettesíteni. Ez nem más, mint dokumentumok lekérdezésekor a referenciaazonosítók kicserélése a ténylegesen hivatkozott dokumentumokra.

3.5 Redis

A Redis [29] egy nyílt forráskódú, „in-memory”, NoSQL adatbázis. Teljes mértékben szakít a relációs adatbázisok relációkra és a köztük lévő kapcsolatokra épülő rendszerével, helyette saját, de a szoftverfejlesztési világban jól ismert adattípusokat támogat, illetve a rajtuk értelmezett műveletek halmazát. Magát az adatbázist úgy kell elképzelni, mint egy kulcs-érték tárolót, amiben a kulcsok valamilyen bináris szekvenciák, de tipikusan string-ek, az értékek pedig valamilyen adatstruktúrák, attól függően, hogy milyen adattípusról van szó. Ilyen adattípusok például a lista, vagy a halmaz, vagy a hashtábla, illetve rendelkezik néhány egyénibbel is, mint a „hyperloglog”, vagy a „bitmap”. A Redis típusain a műveletek mind egyszavas parancsok, amiknek a paraméterei határozzák meg, hogy melyik elemmel mi történjen. Mivel egyszerű kulcs-érték párokról van szó, ezért az elemek között a kapcsolatokat nem támogatja az adatbázis, ha ilyenre van szükségünk, akkor nekünk kell az információt belevinnünk a rendszerbe, tipikusan a kulcs nevének megfelelő megválasztásával. A kulcs-érték párok tárolásán kívül az adatbázis rendelkezik még néhány egyéb beépített funkcióval is, mint egy Publish/Subscribe rendszer, aminek a segítségével, gyorsan, memória alapon tudnak különböző alkalmazások kommunikálni egymással, illetve támogatja utasítások tranzakcióban, vagy pipeline-ban végrehajtását is.

4 Twitch

A Twitch platform egy élő videó közvetítésére szolgáló weboldal, amely elsősorban számítógépes játékokra és játékosokra koncentrál. A platformnak a videó közvetítésen kívül erős közösségépítő szerepe is van, amelynek elsődleges helyszíne a közvetítő csatornákhöz tartozó, IRC alapú chat szobák. A szakdolgozat egyik feladata az IRC rendszerben tevékenykedő automatizált bot implementálása, amihez szükséges volt megismerni a platform fejlesztőknek nyújtott lehetőségeit és a hozzájuk kapcsolódó protokollokat.

4.1 IRC

A Twitch által is használt Internet Relay Chat (IRC), egy, az Internet Engineering Task Force (IETF) által szabványosított, internetes üzenet közvetítő protokoll. Szigorú kliens-szerver architektúrában működik, gerincét szerverek hálózata adja, amikre a kliensek (csevegők) csatlakozni tudnak. A szerverek által létrehozott hálózaton belül ezután tetszőleges számú csatorna hozható létre, ami nem más, mint kliensek névvel ellátott csoportja. Miután egy felhasználó csatlakozott egy csatornához, onnantól kezdve üzeneteket küldhet neki, amit minden egyéb felhasználó is megkap, aki a csatornában van. Az üzenetváltás módja minden esetben szöveges parancsok segítségével történik, mind a szerverek között, mind a szerverek és a kliensek között. Habár a szabvány által nincs megkövetelve, de a kommunikáció *de facto* módon TCP/IP hálózati struktúra felett működik.

A Twitch platform készítői némiképp testre szabták a már szabványosított protokollt. Elsőként az elérhető parancsok halmazát egészítették ki saját parancsokkal, illetve a már meglévők paraméterezését igazították úgy, hogy jobban illeszkedjen az igényekhez. Másodjára ahhoz, hogy csatlakozni tudjunk a hálózathoz, regisztrálnunk kell egy Twitch fiókot. A már elkészült fiókhoz ezek után igényelhetünk egy OAuth2 token 4.2, aminek a segítségével, a felhasználónevünkkel együtt azonosítani tudjuk magunkat a szerver felé. Sikeres azonosítás után már az IRC szabványban, és néhány későbbi kiegészítésben megadott módon szövegesen jönnek az üzenetek a szerver felől. Ez azt jelenti, hogy egy bot programnak, ami ebben a közegben szeretne működni, legnagyobb részt szöveges üzenetek elemzésével kell foglalkoznia.

4.2 Oauth 2

Az Oauth 2 [30] egy IETF által szabványosított autorizációs protokoll, amely lehetővé teszi HTTP erőforrások limitált hozzáférését harmadik fél által. Lényege, hogy egy azonosítási folyamat során, mi, mint harmadik fél kliens alkalmazás úgy érjük el a felhasználó védett erőforrásait, egy erőforrás szerveren, hogy közben a felhasználónak a mi oldalunkon ne kelljen megadnia az erőforrás szerverhez tartozó azonosítóit, ezáltal növelve az erőforrás biztonságát.

A Twitch kétféle azonosítási folyamatot támogat: az egyik az Implicit, amelyet a böngészőben és mobilokon futó, szerver nélküli alkalmazások használhatnak, a másik pedig az Authorization Code folyamat, amely szerverek számára lett kitalálva. Az általam fejlesztett alkalmazás rendelkezik szerver oldallal, ezért célszerűbb volt az Authorization Code-ot jobban megismerni, már csak azért is, mert nagyobb biztonságot nyújt azáltal, hogy a böngészőbe nem jut el az erőforrás eléréséhez használható token, az végig a szerver oldalon marad. Az azonosítási folyamat a következő:

1. Az alkalmazásunkat regisztrálni kell az erőforrás szerverhez tartozó rendszerben, vagyis a Twitch rendszerében. Regisztráció után kapunk egy kliens azonosítót (`client_id`), titkos kulcsot (`client_secret`), és meg kell adnunk egy URL-t, ahova majd visszairányít a folyamat (`redirect_url`).
2. Mikor a felhasználó védett erőforrást akar elérni a böngészőjében, átirányítjuk az erőforrás szerverhez tartozó autorizációs szerverhez, tehát egy tőlünk külön álló, megbízható domain-re (jelen esetben a Twitch oldalára).
3. Itt a felhasználó megadja az azonosítóit, a mi alkalmazásunk számára teljesen láthatatlan módon.
4. Az autorizációs szerver visszairányítja a böngészőt a regisztrációban megadott URL-ünkre, amihez query paraméterben hozzáfűz egy autorizációs kódot. A böngésző ezzel az URL-lel bekérdez a web szerverre.

5. A webszerverünk a kapott autorizációs kóddal bekérdez az autorizációs szerverre, ami visszaad, most már a HTTP válaszban egy tokent (access_token).
6. Ezek után, attól függően, hogy a felhasználótól milyen jogokat kértünk meghívhatjuk a Twitch API megfelelő végpontjait. A kérések fejlécében minden esetben el kell helyoznunk a kapott tokent, aminek a hatására a Twitch erőforrás szervere úgy veszi, mintha a felhasználó nevében járnánk el, az ő jogaival.

5 Fejlesztési folyamat

A fejlesztés folyamán igyekeztem részben olyan folyamat mentén dolgozni, ahogy véleményem szerint egy modern, több tagból álló csapaton és egy korszerű technológiákkal és módszerekkel készülő projekten belül elvárnák.

5.1 Környezeti változók

A fejlesztési folyamat környezeteinek kialakításához elsőként meg kellett határozni azokat a pontokat, amik mentén a különböző környezetek szétválasztása szükséges volt. Ilyen például az adatbázis elérés URL, vagy az OAuth azonosításhoz szükséges azonosítók, titkos adatok. Annak érdekében, hogy a környezeteket szét tudjam választani ezeknek a pontoknak a konfigurációját környezeti változókká vezettem ki, bízva abban, hogy ezek viszonylag biztonságos, de könnyen konfigurálható mivoltukban.

5.1.1 Szerver oldal

Az első hely, ahol környezeti változókat hoztam létre, az a szerver oldal, mivel itt a már jól megszokott módokon, az operációs rendszeren belül adhatók ezek meg, amiket aztán a Node.js beépített módon támogatva ad át alkalmazásunknak. Ezeknek a környezeti változóknak nagy előnye, hogy minden operációs rendszeren elérhetőek, illetve, hogy a felhő alapú hosting szolgáltatások is támogatják létrehozásukat.

- **MONGODB:** A MongoDB adatbázis elérés connection string-ként, ami tartalmazza az autentikációs adatokat és magát a szerver címet is. Konfigurálhatóság alapkövetelmény volt, hiszen nagy valószínűség szerint minden telepítés másik adatbázist használ.
- **REDIS_URL:** A Redis adatbázis elérés, a MongoDB-hez hasonlóan teljes connection string-ként autentikációs adatokkal együtt. Hasonlóan az előzőhöz, ez is általában mindig változik.
- **TWITCH_CLIENT_ID:** A Twitch OAuth szolgáltatáshoz szükséges clientId. Mivel a felhőn futó alkalmazás külön id-t használ, ezért ezt is ki kellett emelnem ide.

- **TWITCH_CLIENT_SECRET:** A Twitch OAuth szolgáltatáshoz szükséges titkos jelszó
- **TWITCH_REDIRECT_URI:** Szintén a Twitch OAuth szolgáltatáshoz szükséges megadni azt az URL-t, ahova bejelentkezés után visszairányítják a felhasználót. A felhőn futó alkalmazásnak természetesen teljesen más az elérése, mint a helyi gépen futónak, ezért ezt is szükséges volt kiemelni.
- **NODE_MODULES_CACHE:** Ez a beállítás csak Heroku-n szükséges, mégpedig azért, mert e nélkül telepítéskor a Heroku nem telepíti újra az alkalmazás függőségeit, hanem, ha lehet az előző telepítésből maradt node_modules mappát másolja át. Ez viszont az én esetemben anomáliákat okozott, ezért kikapcsoltam.

5.1.2 Kliens oldal

Az Angular CLI által nyújtott szolgáltatással nem csak a szerver oldalon, hanem a kliens oldalon is egyszerű módon tudunk konfigurációs beállításokat végezni. Ehhez csupán egy json file-t kell az alkalmazásunk mellé csatolni és beregisztrálni az Angular CLI konfigurációs állományába, ezután az eszköz elérhetővé fogja tenni számunka az Angular alkalmazásunkban, kliens oldalon is, environment néven. A dolog érdekessége, hogy egy egyszerű json file-t Angular CLI nélkül is probléma nélkül be tudunk húzni moduljainkba a Webpack alkalmazása miatt. A CLI viszont lehetővé teszi, hogy több konfigurációs állomány közül transparens módon mindig az aktuális környezetnek megfelelő legyen az, amelyik az environment név mögött áll. Azt, hogy melyik környezet változóit szeretnénk éppen használni az Angular alkalmazás fordításakor kell megadnunk, az „env” argumentum segítségével, például:

```
ng build --env=staging
```

Az általam használt kliens oldali környezeti változók tehát:

- **production:** egy boolean kapcsoló, ami magának az Angular keretrendszer működését befolyásolja, azáltal, hogy bekapcsolt állapotban az alkalmazás induláskor meghívja az enableProdMode Angular metódust.

- `clientId`: A szerver oldali `TWITCH_CLIENT_ID` változóval megegyező, Twitch OAuth azonosításhoz szükséges azonosító. Azért van rá szükség kliens oldalon, mert a kliens oldal irányítja át a felhasználót a Twitch bejelentkezési oldalára, ahol szükséges ez az információ paraméterként.
- `deployUri`: Az URL, ahova az alkalmazás jelenleg ki van telepítve. Egyrészt szükséges a korábban említett Twitch OAuth azonosításhoz, mivel a bejelentkezési folyamat erre az URL-re fog minket visszairányítani. Másrészt szükséges az alkalmazásban használt Websocket működéséhez, ami alapból localhost-ra szeretne csatlakozni, de ez például felhőbe való telepítéskor gondot okoz.

5.2 Lokális fejlesztés

Az alkalmazás helyi gépen való fejlesztéséhez a legfontosabb követelmények a Node.js és az npm (amit tartalmaz a Node.js csomag) telepítése a számítógépre. Ezen kívül természetesen be kell állítani sorban a korábban említett környezeti változókat, azonban, az hogy az adatbázisok hol vannak telepítve az nem fontos, ha az elérés biztosított. Mivel én Windows operációs rendszeren fejlesztettem, ezért például Redis adatbázis telepítése nem volt lehetséges a helyi gépen, én ebből egy felhőbe telepített verziót használtam a helyi gépen is, míg MongoDB adatbázis telepítve volt a helyi gépen.

Az alkalmazást lokálisan végig fejlesztői módban futtattam, amihez megfelelő npm script-eket állítottam be a `package.json` állományban. Az alkalmazás szerver és kliens oldalát külön-külön kell futtatni ebben az esetben, mivel az Angular CLI-ba beépítve megvan a lehetőség (a Webpack által), hogy az alkalmazás fájljait figyelje, és változás esetén újratöltse a böngészőt, viszont ez nem tudja kezelni a szerver oldal futtatását. A szerver oldal futtatását végző parancs:

```
"api": "cd server && nodemon --ext ts --ignore *.spec.ts --exec ts-node --project tsconfig.app.json --no-cache app.ts"
```

A parancs tehát az „api” nevet kapta, első lépése, hogy belép a „server” könyvtárba, ahol értelemszerűen a szerver oldal állományai találhatók. Működésének alapja a `nodemon` nevű program, ami úgy indít el egy Node.js programot, hogy, csakúgy, mint az Angular CLI, folyamatosan figyeli annak fájljait, és ha bármelyik változik, akkor újraindítja a programot a változásokkal együtt, ezzel gyorsítva a

fejlesztést. A nodemon-nak ezek után a megfelelő argumentumokkal beállítottam, hogy ne JavaScript, hanem TypeScript állományokat kell most kezelnie (ehhez szükséges plusz egy eszköz, a ts-node), illetve megmondtam neki, hogy a teszt állományokat ne vegye figyelembe, melyik TypeScript konfigurációs fájlt töltse be és melyik az alkalmazás belépési pontja. Egy fontos argumentum még a „--no-cache” kapcsoló, amivel egy anomáliát kellett kiküszöbölöm, ami azt okozta, hogy tartalomtól függően bizonyos fájlokat nem volt képes újrafordítani változáskor a ts-node, amivel természetesen az egész konfiguráció elvesztette volna az értelmét.

Ahhoz, hogy ne kelljen külön terminál ablakban futtatni a szerver és kliens oldalt, létrehoztam egy összefoglaló script-et is az alábbi tartalommal, „dev” néven:

```
"dev": "cross-env PORT=3000 concurrently \"ng serve --port 4200 --progress false --proxy-config proxy.conf.json\" \"npm run api\" "
```

Ennek első utasítása a cross-env nevű Node.js program futtatása, ami platform független módon beállítja a megadott környezeti változót, jelen esetben a PORT nevűt 3000-es értékre, majd futtatja az utána álló utasításokat. A port beállítására a szerver oldal-nak volt szükséges, ugyanis a kliens oldalt jelen esetben nem a szerver szolgálja ki, hanem külön fut egy másik porton és egy proxy-n keresztül csatlakozik a szerverre. A parancs második része a concurrently program végrehajtása, ami az argumentumban kapott utasításokat párhuzamosítva hajtja végre, tehát lehetővé vált a kliens és a szerver párhuzamos kiszolgálása. A concurrently-nek megadott második parancs az „npm run api”, ami a szerver oldalt indítja a korábban megismert script segítségével. Az első parancs lesz az, ami a kliens oldalt fogja kiszolgálni az Angular CLI segítségével a 4200-as porton, illetve annak a szerver oldali hívásait a „proxy.conf.json” állományban konfigurált módon továbbítja a külön futó szerver felé a már említett proxy segítségével.

5.3 Távoli, felhő környezet

A helyi környezeten kívül, bemutatási és tesztelési célokra létrehoztam egy felhőben futó környezetet is egy Platform as a Service szolgáltató, a Heroku segítségével, illetve egy odáig vezető Continuous Integration csővezetékét.

5.3.1 GitHub

A távoli környezet első pontja a GitHub [31] segítségével létrehozott git repository. Ennek az elsődleges funkciója természetesen a verziókezelés, hiszen az a szoftverfejlesztésben egy nagyon régóta fenn álló probléma, hogy változtatásainkat ne veszítsük el valamilyen rendszerhiba esetén, sőt, alkalmazásunk korábbi verzióit is elő szeretnénk venni adott esetben. A verziókezelés történetében természetesen nagyon sokféle megoldás született, talán az egyik legújabb, de így is már elég régi és kiforrott megoldás erre a git eszköz, aminek felhő alapú támogatására jött létre a GitHub. Esetemben két branch-et vezettem az alkalmazás fejlesztése során, egy „master” és egy „develop” ágat, ahol a fejlesztés mindig a develop ágra történt és csak a stabil állapotokat vittem át („merge-öltem”) a master ágra. Egy másik általam használt funkciója a GitHub-nak a feladatok, hibák, vagyis az issue-k kezelése. A felmerülő feladatokat már a fejlesztés elejétől ebben a rendszerben tároltam, illetve ezeket mérföldkövekhez, milestone-okhoz rendeltem, amik nemcsak csoportosítják a feladatokat, hanem a csoportnak határidőket is képes szabni. Az utolsó, általam kihasznált nagyon fontos tulajdonság, a külső rendszerekkel való integrálás támogatása, mint például a most következő Travis-szel.

5.3.2 Travis

A Travis [32] egy felhő alapú Continuous Integration, vagyis CI szolgáltatás. Mint ilyen, célja, hogy a kódbázisunkat ért legkisebb változások is, vagyis minden egyes commit azonnal be legyenek építve az alkalmazásunkba megbízható, tesztelt módon. Működését úgy kell elképzelni, hogy meg kell adni egy forrás kódbázist, esetünkben egy GitHub repository-t, amit a Travis folyamatosan figyel. Ha változást érzékel, akkor a repository jelenlegi állapotát lemásolja magához és egy sor feladatot hajt végre rajta a beállításainktól függően. Ezek a feladatok lehetnek például az alkalmazás fordítása, tesztelése, statikus elemzése, telepítése valamilyen környezetben, majd pedig az eredményekről értesítések küldése. Azt, hogy a Travis konkrétan milyen utasításokat hajtson végre egy Travis specifikus konfigurációs állományban tudjuk meghatározni, amit a git repository-nkban kell elhelyeznünk. A konfigurációs fájlokkal azonban problémákba is ütközhetünk, ugyanis a Travis minden ágon megköveteli a jelenlétüket, amelyiken szeretnénk a Travis szolgáltatásait használni. Esetemben, az én két ágamon a master-en és a branch-en különböző működést szerettem volna elérni,

mivel a develop ágon szerettem volna az ellenőrzések futtatását, viszont a master ágon ennél többet akartam, az alkalmazás telepítését is a felhő környezetbe. Ehhez tehát úgy kellett commit-olnom, illetve merge-ölnöm a konfigurációs fájlokat, hogy a két ág közötti különbségeket meg tudjam tartani, viszont a jövőbeli merge-öléseknél ne kelljen minden egyes alkalommal törődni azokkal. Ennek eredményeként a két ágon a konfigurációs állomány eleje megegyezik. Itt arra utasítom a Travis-t, hogy egy Linux-ra telepített Node.js alkalmazást telepítsen fel a repository-ból megszerzett kódból, majd futtasson rajta lint segítségével statikus kódelemézést és futtassa a teszteket. Ezek az utasítások tehát mindkét ágon való módosulás után lefutnak, a master ágon viszont a konfigurációt kiegészítettem egy deploy résszel, amiben utasítom a Travis-t, hogy sikeres ellenőrzések után telepítse az alkalmazást a Heroku megfelelő alkalmazására. Ha a Travis a lint, vagy a tesztek futtatása közben hibát észlel, akkor a teljes folyamatot leállítja és az alkalmazás nem fog települni a tesztkörnyezetbe.

5.3.3 Heroku

A Heroku [33] egy felhő alapú Platform as a Service (PaaS) szolgáltatás, ami több különféle programozási nyelvet és platformot is támogat, például, Node.js, Java, Ruby, Python. Mint PaaS tehát, a fejlesztők dolgát nagymértékben megkönnyíti a telepítés során. Nincs szükség az operációs rendszerek és egyéb eszközök bonyolult konfigurációjára, egyszerűen megszabják a módokat, ahogy telepíthetjük ide az alkalmazásunkat, illetve megadják a lehetőséget a környezet beállítására. A telepítés pontos menete az alkalmazásunk nyelvétől és platformjától függ, de alapesetben úgy történik, hogy valamilyen módon el kell juttatnunk a forráskódunkat az alkalmazáskonténerbe. Ez sokféleképpen lehetséges, például közvetlenül GitHub repository-ból, vagy a Heroku parancssori alkalmazásának használatával, vagy esetemben a Travis segítségével. A forráskód alapján a Heroku telepíti az alkalmazásunkat az általunk beállított parancs, vagy script segítségével. Az alkalmazás esetében ez például először az „npm install” parancs kiadása, ami telepíti az alkalmazás függőségeit, majd a „npm run heroku-postbuild” parancs futtatása, ami lefordítja és csomagolja a kliens és a szerver oldal TypeScript állományait. Miután ezekkel végzett a Heroku elindítja az alkalmazást, például Node.js alkalmazás esetén alapértelmezetten az „npm start” parancs segítségével. A Heroku ezek után a fizetési tervünktől függően rengeteg egyéb teendőt elvégezhet az alkalmazással kapcsolatban, például életciklus kezelés, mérések, log bejegyzések gyűjtése.

5.3.4 Adatbázisok

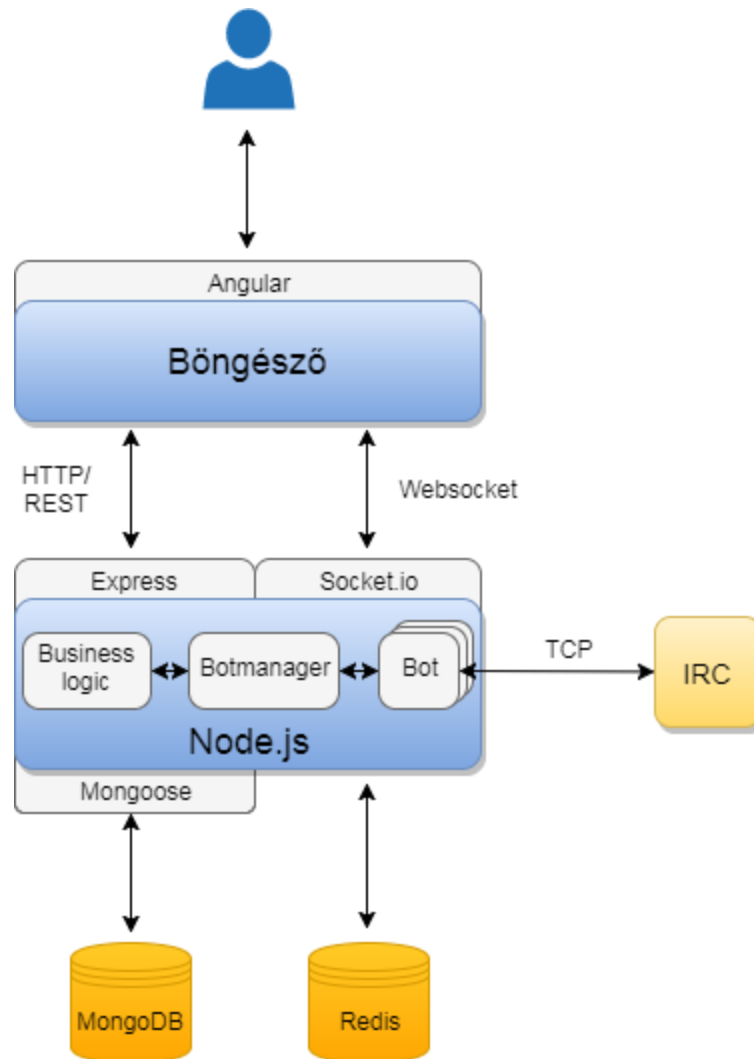
Mivel a Heroku alapesetben csupán egy alkalmazás konténert ad nekünk, ezért az egyéb szolgáltatásokról, például adatbázisokról nekünk kell gondoskodnunk. Ezeket általában a Heroku maga képes rendelkezésünkre bocsátani, ha kérjük, de olyan is elképzelhető, hogy más, az internet felől elérhető helyen létrehozott szolgáltatást használunk innen.

Esetemben a két adatbázis, amit használok a Redis és a MongoDB. Ezekből mindkettő megtalálható a Heroku által nyújtott Add-on-ok között, de én csupán a Redis-t telepítettem a Heroku-n belül. A Heroku Redis hozzákötése az alkalmazásunkhoz nagyon egyszerűen történik. Először is meg kell adnunk egy bankkártyánk adatait, hogy érvényesítsük az fiókunkat. Másodjára meg kell keresnünk a kiterjesztések között a hozzáadni kívántat, tehát most a Redis-t. Végül választanunk kell egy fizetési tervet, attól függően, hogy mekkora teljesítményt akarunk magunknak, esetemben ez ingyenes volt. A fizetési terv választása után hozzá is rendelődik az alkalmazáshoz az adatbázis, amihez megkapjuk a kapcsolódási adatokat.

A MongoDB adatbázist nem a Heroku-n keresztül tettem elérhetővé az alkalmazás, hanem azt egy külső szolgáltató, a MongoDB Atlas rendszerén belül készítettem el. Az Atlas több népszerű Cloud platformon is szolgált, például Amazonon vagy Google-ön, ezekből bármelyik választható, illetve itt szintén elérhető ingyenes próbaverzió is, korlátozott kapacitással. Regisztráció után hasonlóan az előzőhöz, itt is megkapjuk a kapcsolódási információkat.

6 Az alkalmazás felépítése

6.1 Architektúra



6. ábra Az alkalmazás architektúrája

Az alkalmazás kialakítása a többretegű architektúra segítségével valósult meg. Az első réteg a megjelenítés, ahol is a felhasználó a böngészőjében futó Angular alkalmazással kommunikál a számítógép interfészei segítségével. A megjelenítési réteg HTTP/REST kéréseket küld a szervernek, illetve a „Socket.io” [38] könyvtárral megvalósított, WebSocket protokollon keresztül is tartják a kapcsolatot. A szerver oldalon az alkalmazás egy Node.js futtató környezetből áll. Ezen fut az Express webservert, ami a HTTP kéréseket fogadja és az üzleti logika egy részét is tartalmazza, valamint a Socket.io könyvtár szerver oldali része. Az üzleti logika egy segédmodul, a

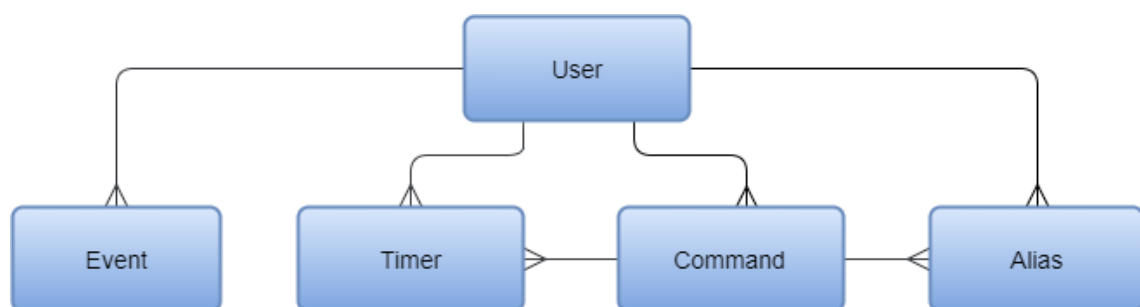
BotManager segítségével kezeli az éppen futó botokat, amik a tényleges kommunikációt végzik az IRC protokollt végző Twitch chat szerverrel, TCP-n keresztül. Az üzleti logika kommunikál még a MongoDB adatbázissal a mongoose könyvtáron keresztül, és a Redis-szel, a kiajánlott JavaScript Api segítségével.

6.2 Adatstruktúrák

Az alkalmazás adatainak tárolása két NoSQL adatbázis, a MongoDB és a Redis használatával lett megvalósítva. Habár mindkettő NoSQL adatbázis, tehát nem a relációs adatbázisok tábláira és kapcsolataira épülnek, ezek is rendelkeznek adatstruktúrákkal. Az adatbázisok struktúráin kívül további fontos szereplő még a MongoDB elérésére szolgáló mongoose könyvtár, ami megköveteli sémák létrehozását az adatbázis használata előtt.

6.2.1 Mongoose sémák

A MongoDB adatstruktúrái a dokumentumok, amik egymáshoz kötődő adatok halmazát tartalmazzák, hasonlóan a relációs adatbázisok rekordjaihoz. Az azonos célra használt dokumentumokat gyűjteményekben lehet csoportosítani, hasonlóan a relációs adatbázisok tábláihoz, azzal a kivételleve, hogy itt nem kell az attribútumoknak megegyezniük. A mongoose erre építve vezeti be tehát a séma fogalmát, ami a gyűjteményeknek felel meg, de ellentétben a gyűjteményekkel, a sémákat előre meg kell készítenünk, és ezek szigorú struktúrát írnak elő.



7. ábra A mongoose sémák kapcsolatai

6.2.1.1 User

A felhasználók tárolására szolgáló séma, a Twitch API-ja által visszaadott adatstruktúrára építve.

Attribútum	Attribútum leírása
_id	MongoDB által automatikusan generált, ObjectID típusú azonosító
name	A felhasználó csatornájának az azonosítója, illetve ebből a névből származik a felhasználóhoz tartozó chat szoba elérése is
display_name	A felhasználó megjelenítési neve, ami a többi felhasználó számára látható, például chat-en.
email	A felhasználó e-mail címe.
created_at	Időbélyeg, a felhasználó regisztrálásának az időpontja a Twitch rendszerbe
token	Az OAuth token, amit a program szerez, amikor a felhasználó belép a portálra

6.2.1.2 Command

A felhasználók által használható, bot által megtanult parancsok.

Attribútum	Attribútum leírása
_id	MongoDB által automatikusan generált, ObjectID típusú azonosító
name	A parancs neve, másképpen fogalmazva az a szöveg, ami alapján a felhasználók futtathatják a parancsot. Egy csatornán belül egyedinek kell lennie.
text	A szöveg, ami a parancs futásakor elküldésre kerül.
enabled	Engedélyezve van-e a parancs az adott csatornán.
user	A parancshoz tartozó felhasználó, csak az ő csatornáján lesz érvényes a parancs. Ez a mező egy MongoDB specifikus referenciát jelent, vagyis lényegében a hagyományos SQL adatbázis idegen kulcsának a megfelelője.

6.2.1.3 Timer

A parancsok időzítésének tárolására szolgáló típus

Attribútum	Attribútum leírása
_id	MongoDB által automatikusan generált, ObjectID típusú azonosító
name	Az időzítő neve, ami alapján a felhasználóknak megjelenik. Egy csatornán belül egyedinek kell lennie.
enabled	Engedélyezve van-e az időzítő futása
timeInMinutes	Hány percenként hajtódjanak végre az időzítőhöz tartozó parancsok.
user	Az időzítőhöz tartozó felhasználó, csak az ő csatornáján lesz érvényes az időzítő. Szintén egy MongoDB specifikus referencia a User kollekcióra.
commands	Az időzítőhöz tartozó parancsok listája, amik le fognak futni, mikor az időzítő elsül. A mező egy MongoDB lista referencia a Command típusra.

6.2.1.4 Alias

A parancsok fedőneveinek tárolására szolgáló típus.

Attribútum	Attribútum leírása
_id	MongoDB által automatikusan generált, ObjectID típusú azonosító
name	A fedőnév neve, ez lesz az a parancs, amit a felhasználónak ki kell adnia, hogy a fedőnévhez tartozó parancsot futtatni tudja.
user	Az fedőnévhez tartozó felhasználó, csak az ő csatornáján lesz érvényes az fedőnév. Egy MongoDB specifikus referencia a User kollekcióra.
command	A fedőnévhez tartozó parancs, ami le fog futni, amikor egy felhasználó futtatja a fedőnevet, mint parancsot. A mező egy MongoDB referencia a Command típusra.

6.2.1.5 Event

A felhasználói és bot események naplózására szolgáló típus

Attribútum	Attribútum leírása
_id	MongoDB által automatikusan generált, ObjectID típusú azonosító
timestamp	Időbélyeg, az esemény keletkezésének időpontja
level	Az esemény súlyossága, például: info, error, warning
message	Az eseményhez tartozó üzenet
meta	Az eseményhez tartozó, bármi egyéb meta adat, például az eseményhez kapcsolódó csatorna

6.2.2 Redis típusok

Az alkalmazásomban a MongoDB melletti adatbázis a Redis, aminek beépített típusaival kell megoldanunk a számunkra szükséges funkcionalitást, ugyanis saját típus létrehozása nem engedélyezett. A két típus, amit alkalmazásomban használtam a Set és a Sorted Set voltak.

A Set, vagyis halmaz típus string-ek sorrend nélküli, egyedi gyűjteménye. Értelmezettek rajta egyrészt az olyan egyszerű műveletek, mint elemek hozzáadása, eltávolítása, vagy annak eldöntése, hogy egy elem benne van-e a halmazban. Másrészt megtalálhatók rajtuk a matematikai halmazok műveleti is, mint a metszet, unió és a különbségképzés.

A Sorted Set, vagyis rendezett halmaz típus ugyanúgy string-ek egyedi gyűjteménye, de ellentétben a sima halmazzal, itt minden elemnek kötelező jelleggel meg kell adnunk egy pontértéket, ami alapján a halmaz elemei sorrendbe vannak állítva. Ebben a típusban is megtalálhatók a sima halmazok műveletei, viszont kiegészülnek az index és a pont alapján működő parancsokkal is, mint például csak bizonyos pontú elemek keresése.

6.2.2.1 polls Set

A „polls” nevű Set célja, hogy nyilván tartsa azokat a csatornákat, amelyekben éppen szavazás folyik. A halmaz elemei azoknak a felhasználóknak a MongoDB beli

azonosítói, akiknek a csatornájában szavazás van folyamatban. Használatával elkerülhető, hogy véletlenül egy csatornán belül két szavazás is induljon.

6.2.2.2 poll:voters:<userId> Sets

A „poll:voters:<userId>” név valójában nem egy halmazt, hanem halmazok halmazát jelenti, ahol a <userId>-t mindig be kell helyettesíteni annak a felhasználónak a MongoDB azonosítójával, akinek a csatornájában a szavazás folyik. Minden egyes szavazásnak tehát létrejön egy külön poll:voters:<userId> halmaz, aminek a célja, hogy azoknak a chat felhasználóknak a nevét gyűjtse össze, akik már az adott felhasználó csatornáján szavaztak a jelenleg futó szavazásban. A nyilvántartás célja, hogy egy adott szavazáson belül egy felhasználó ne szavazhasson többször.

6.2.2.3 poll:<userId> Sorted Sets

A „poll:<userId>” az előző halmazokhoz hasonlóan szintén nem egy rendezett halmazt jelent, hanem minden szavazást futtató felhasználóhoz egyet, ahol a <userId> a csatorna gazdájának az azonosítója. A halmaz feladata azoknak a lehetőségeknek a tárolása, amikre szavazni lehet, illetve annak a tárolása, hogy egyes opciókra hányan szavaztak eddig. Ennek megvalósítására ideális a rendezett halmaz, ahol a halmaz elemei az opciók, a hozzájuk tartozó értékpontok pedig azoknak a száma, akik az adott lehetőségre szavaztak.

6.2.2.4 raffles Set

A „raffles” halmaz hasonlóan a „polls” halmazban azoknak a felhasználóknak az azonosítóit tárolja a rendszer, akiknek a csatornájában éppen aktuális nyereményjáték folyik. Célja szintén az, hogy egy csatornán belül egy időben csak egy nyereményjáték lehessen aktív.

6.2.2.5 rafflers:<userId> Sets

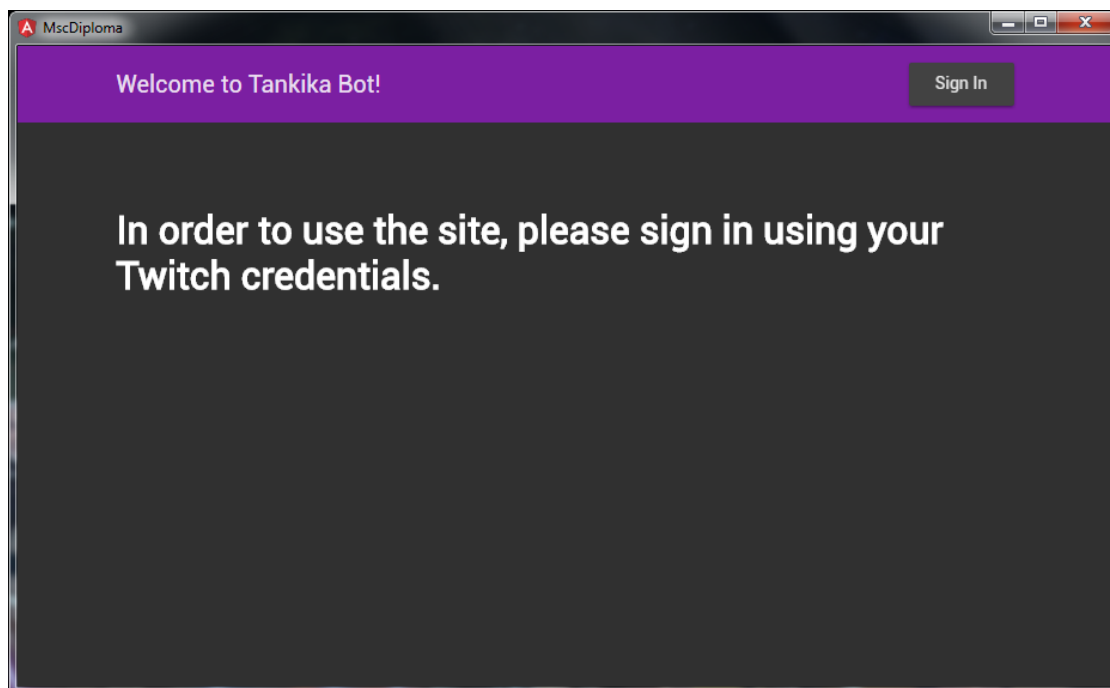
A „rafflers:<userId>” halmazok halmaza azoknak a felhasználóknak a chat nevét tartalmazza, akik az aktuális nyereményjátékokra regisztráltak. Itt is minden csatornához tartozik egy külön halmaz, ahol a <userId> a csatorna tulajdonosának a neve. Szerepük, hogy egy adott felhasználó nyereményjátékában egy felhasználó csak egyszer szerepelhessen.

7 Funkciók

A program több különálló felületből áll, ezek között az Angular által alaptól támogatott és javasolt módon, routing segítségével tudunk navigálni. Habár az Angular Single Page Application weboldalak fejlesztését támogatja, az oldal felhasználója számára ez transzparens módon tud megjelenni, vagyis például mikor egy új oldalra navigálunk, az Angular frissíti a böngésző címsávjában megjelenő URL-t. Az, hogy melyik nézetünk melyik URL-hez tartozik a routing segítségével adható meg. Alkalmazásomban két darab routing module található, egy az anoním, nem bejelentkezett felhasználók kiszolgálására, illetve egy a bejelentkezetteknek.

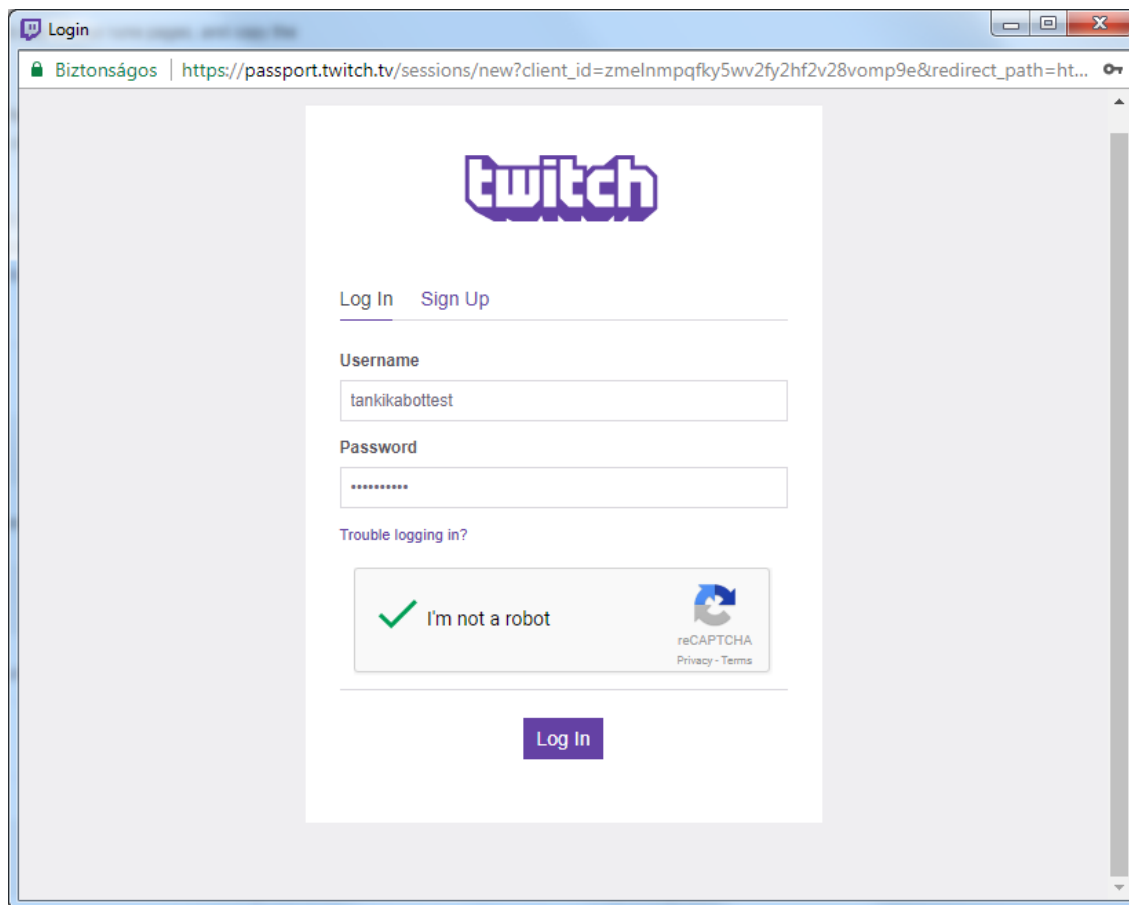
7.1 Fogadó oldal

A fogadó oldal az a felület, amit a felhasználó először lát mikor az oldalra látogat, és nem rendelkezik a böngészője aktív session-nel. Ez egy nagyon egyszerű oldal, itt lényegi funkcionalitás nem található, szerepe csupán annyi, hogy elkalauzolja a felhasználót a Twitch portál által nyújtott OAuth bejelentkező felületre, ahol azonosítani tudja magát, az alkalmazás pedig meg tudja szerezni egy, a felhasználóhoz tartozó authentication token-t.



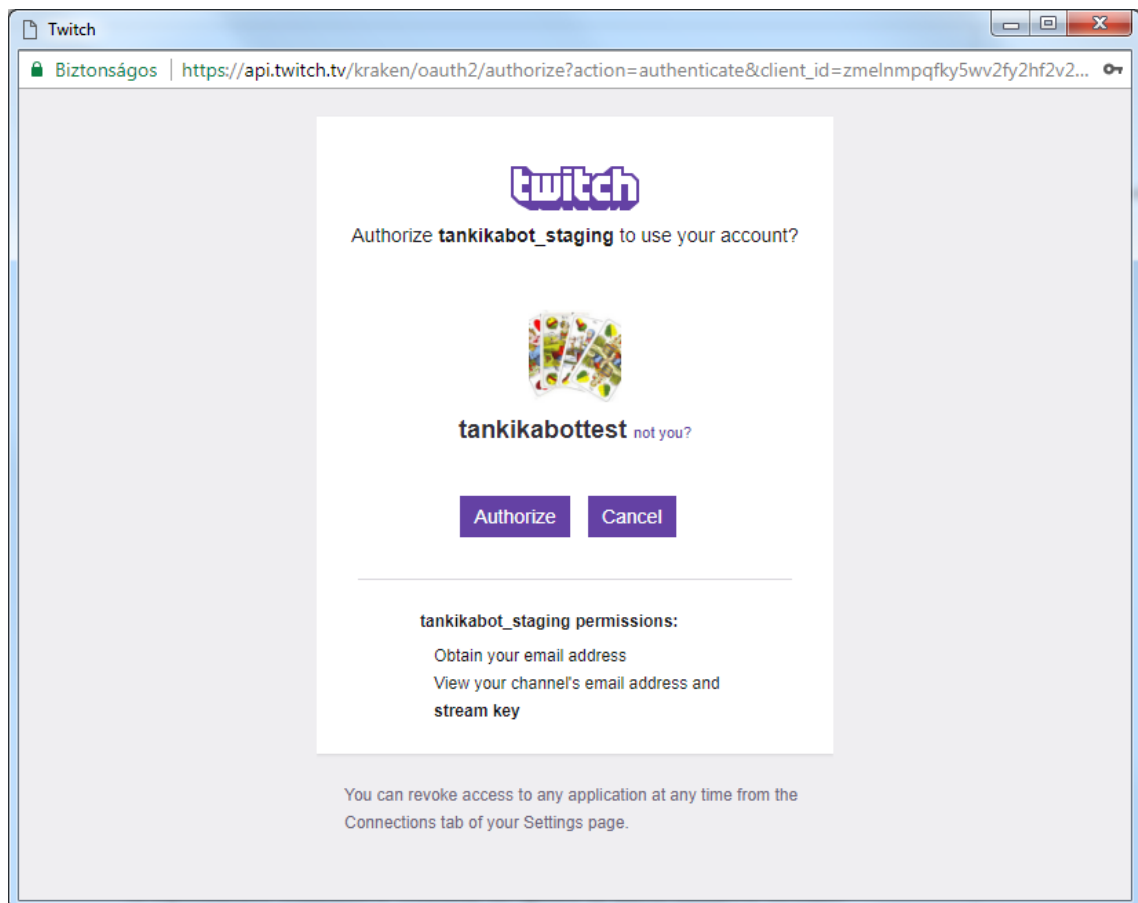
8. ábra Fogadó oldal

A felhasználónak az oldalon a két lehetősége van: vagy elhagyja azt, vagy pedig megnyomja a „Sign In” feliratú gombot. Ennek hatására, ha a böngésző nem rendelkezik aktív session-nel a Twitch portálhoz, akkor a felhasználó átirányításra kerül annak a bejelentkező felületére.



9. ábra Twitch OAuth bejelentkező felület

Miután a felhasználó megadta a nevét és jelszavát (vagy regisztrált), még engedélyezni kell, hogy a megfelelő jogokat tényleg meg szeretné-e adni a portálnak, hogy az ő nevében tudjon cselekedni.



10. ábra Twitch autorizációs felület

Ha a felhasználó megadta az oldalnak a kívánt jogokat, akkor visszairányításra kerül az alkalmazás oldalára (a visszairányítási URL-t előre regisztrálni kell a Twitch-nél, csak ide irányíthat vissza a bejelentkező felület). Az alkalmazáson belül viszont már nem a Fogadó oldalra kerül, hanem a bejelentkezett felhasználók oldalára.

A bejelentkezett felhasználók oldalának navigálása az alábbi esetekben történhet meg:

- A felhasználó aktív session-nel rendelkezik az oldalon, ekkor a Fogadó oldal meg sem nyílik, a felhasználó egyből a bejelentkezettek felületére kerül
- A felhasználó az oldalon korábban már bejelentkezett, tehát benne van a rendszerben és aktív Twitch session-nel rendelkezik, de nincs az oldalon aktív session-je. Ekkor a felhasználónak rá kell nyomnia a „Sign In” gombra és átirányításra kerül a twitch oldalára, azonban mivel ott már be van jelentkezve, egyből visszairányításra is kerül és beléphet a rendszerbe, a bejelentkezési adatok megadása nélkül.

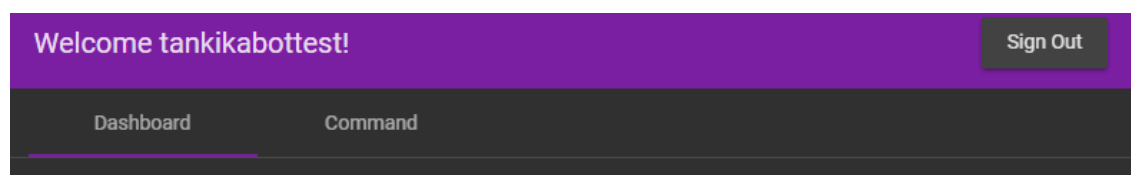
- A felhasználó először jár az oldalon, nincs benne a rendszerben, tehát nincs is aktív session-je. Ekkor lezajlik a korábbiakban leírt teljes bejelentkezési folyamat és a felhasználó bekerül a rendszerbe, ez tekinthető a regisztrálás műveletének.

Az, hogy a kliens oldalon milyen eset áll fenn éppen, egy Angular nyújtotta routing szolgáltatás, a CanActivate guard [34] segítségével van eldöntve. A guard egy felhasználó által írt TypeScript osztály, aminek implementálnia kell a CanActivate interface-t, aminek egyetlen metódusa a canActivate visszatérési értéke eldönti, hogy az adott felhasználó a megadott nézetre navigálhat-e vagy sem. Ezt az osztályunkat kell regisztrálnunk a routing module-ban a védeni kívánt nézethez és a keretrendszer a nézetre való váltás előtt minden esetben ellenőrizni fogja, hogy a guard szerint megtörténhet-e a navigáció. Ebben a konkrét esetben az általam írt AnonymGuard canActivate metódusa meghívja az AuthService szolgáltatás getUser metódusát. Ez aktív session esetén visszaadja az éppen bejelentkezett felhasználó adatait, ha van ilyen, a guard pedig tovább navigáltat a bejelentkezett felhasználók oldalára, ha pedig nincs, akkor maradunk a jelenlegi oldalon.

7.2 Bejelentkezett felhasználók felületei

A bejelentkezett felhasználók számára elérhetővé válik az oldal által nyújtott összes szolgáltatás.

Az oldal keretét routing szempontból egy külön nézet, a center adja, minden egyéb nézet ennek a leszármazottja. A center nézet két részből, a fejlécből és a tartalomból áll. A fejlécben kap helyet az üdvözlő szöveg, illetve ami fontosabb a menüsor, amivel azt mondhatjuk meg, hogy éppen melyik felületre akarunk navigálni. Navigáláskor az oldal nem fog teljes mértékben újratöltődni, a menüsor sosem töltődik újra, csupán az erre a célra létrehozott tartalom elem fog lecserélődni azzal a nézettel, amit az éppen aktuálisan betöltött URL reprezentál.



11. ábra Alkalmazás fejléc

7.3 Központi oldal (Dashboard)

Az alapértelmezetten megnyíló nézet a Központi oldal / Dashboard, ahol a felhasználók főként azonnali műveleteket tudnak végrehajtani. Az oldal elrendezése az Angular-os FlexLayout által csomagolt flexible box CSS elrendezés segítségével lett megvalósítva, ezáltal alacsony felbontásokon is szépen jelenik meg, nem csúsznak el az elemek. Az elrendezés alapja egy oszlop irányú flex container, amiben tetszőleges számú sor lehet, mindegyik egy 40 pixel magas sávval elválasztva, úgy, hogy a sorok magasságát önmaguk határozhatják meg tartalmuk alapján. Minden sor két további oszlopból áll, egy bal oldali, nagyobb, 70% szélesből és egy jobb oldali, kisebb, 28% szélesből. A két oszlop között egy olyan sáv lett beillesztve, aminek nincs megadva explicit szélessége, ezért ez a maradék helyet tölti ki, igazodva az éppen aktuális oldalmérethez. Az oldal betöltődése előtt, az Angular guard-ok egy másik fájtája, a Resolve [34] segítségével töltődnek le az oldal megjelenítéséhez szükséges adatok. A Resolve guard létrehozása a CanActivate-hez hasonlóan úgy történik, hogy egy saját osztályt készítünk, ami implementálja a Resolve interface-t és, ami egyetlen metódusában, a resolve-ban visszaad egy Promise-t, ami tetszőleges adatot szolgáltat. Ha létrehoztuk a guard-unkat, akkor azt beköthetjük tetszőleges route-ra, aminek eredményeképpen a megadott nézet nem fog addig betölteni, amíg a hozzá beregisztrált resolve guard-ok promise-ai nem resolve-olódnak. Ilyen resolve guard-ot hoztam létre minden, az oldalon található funkcióhoz kapcsolódóan:

- **CommandsResolveGuard:** A csatornához tartozó parancsokat adja vissza, a parancs gyorsfuttatása funkcióhoz.
- **EventsResolverGuard:** Az események betöltéséért felelős az esemény listázó komponenshez.
- **RaffleResolverGuard:** A nyereményjáték-hoz adja meg, hogy éppen jelenleg fut-e olyan, vagy sem.
- **PollResolverGuard:** Visszaadja, hogy jelenleg fut-e szavazás a csatornán, illetve, ha igen, akkor az éppen aktuális szavazat állásokat is.

7.3.1 Események (Events)

Az esemény panel feladata, hogy a felhasználó által generált események naplózásra kerülése után a felhasználó vissza tudja őket nézni. Bármely funkció bármely

eseményébe beköthető az események naplózása, a bot és a kezelő alkalmazás oldalán is, csupán egy szöveget és némi meta adatot kell megadnunk, ezek után meg fog jelenni az esemény a panelen.

7.3.1.1 Eseménynaplózó szolgáltatás

Az események naplózása szerver oldalon történik, a Winston [36] logoló JavaScript könyvtár segítségével. A Winston röviden arra képes, hogy segítségével különböző szintű log eseményeket generálhatunk, amiket aztán a könyvtár kiír a megfelelő helyekre, a log szintjétől és egyéb tényezőktől függően. A szinteket névvel ellátott számok sorrendjeként képzelhetjük el, ahol a legsúlyosabb szint az error, 0, a legenyhébb a silly 5-ös értékű. A logger beállítható, hogy bizonyos szint felett ne foglalkozzon az üzenetekkel, például a 4-es debug szinten nem fogja figyelembe venni az 5-ös, silly szinten létrehozott üzenetekkel. Habár elsődleges feladata általános események logolása, megfelelő konfigurációval el tudtam érni azt a viselkedést, amire szükségem volt az alkalmazás specifikusabb események naplózásához. Ehhez elsőként létrehoztam egy eventLogger nevű Winston logger példányt, amit egy TypeScript modulban kiejátsoltam, így minden szerver oldali modulban elérhetővé tettem. Az eventLogger-nek két, úgynevezett transport-ot állítottam be, amiknek a feladata, hogy kiírják a logokat a transport típusától függő helyre, például adatbázisba. A két transport közül az első egy egyszerű Console transport, csupán debug célokra létrehozva, színesen, időbélyeggel ellátva kiírja a debug és kisebb szintű üzenetet a konzolra. A második transport, ami a lényegi működést hordozza, egy MongoDB transport, amit külön npm csomagként kellett telepíteni „winston-mongodb” modul néven. Ez a MongoDB transport, ahogy a neve is mutatja, mongodb adatbázisba tárolja a logokat, egy általa definiált sémával (ezt a sémát képeztem le a saját, Event adattípussal). A MongoDB transport használatához meg kellett adni az adatbázis elérést, amit a MONGODB környezeti változó képében lett eltárolva, illetve, opcionálisan a MongoDB collection nevét, amibe a logok kerülni fognak, ez az „events” nevet kapta az alkalmazás esetében. Az eventLogger legtöbb esetben az express middleware-ekbe lett bekötve, például egy parancs sikeres hozzáadásához, törléséhez, stb, illetve a bot közvetlenül használja a csatornához csatlakozáskor és kilépéskor. Annak érdekében, hogy azonosítani lehessen, melyik csatornához tartoznak az események, az eventLogger használatakor még arra kell figyelni, hogy meta adatként meg legyen adva a csatorna neve, illetve a releváns esetekben az eseményhez kapcsolódó entitás neve, például a

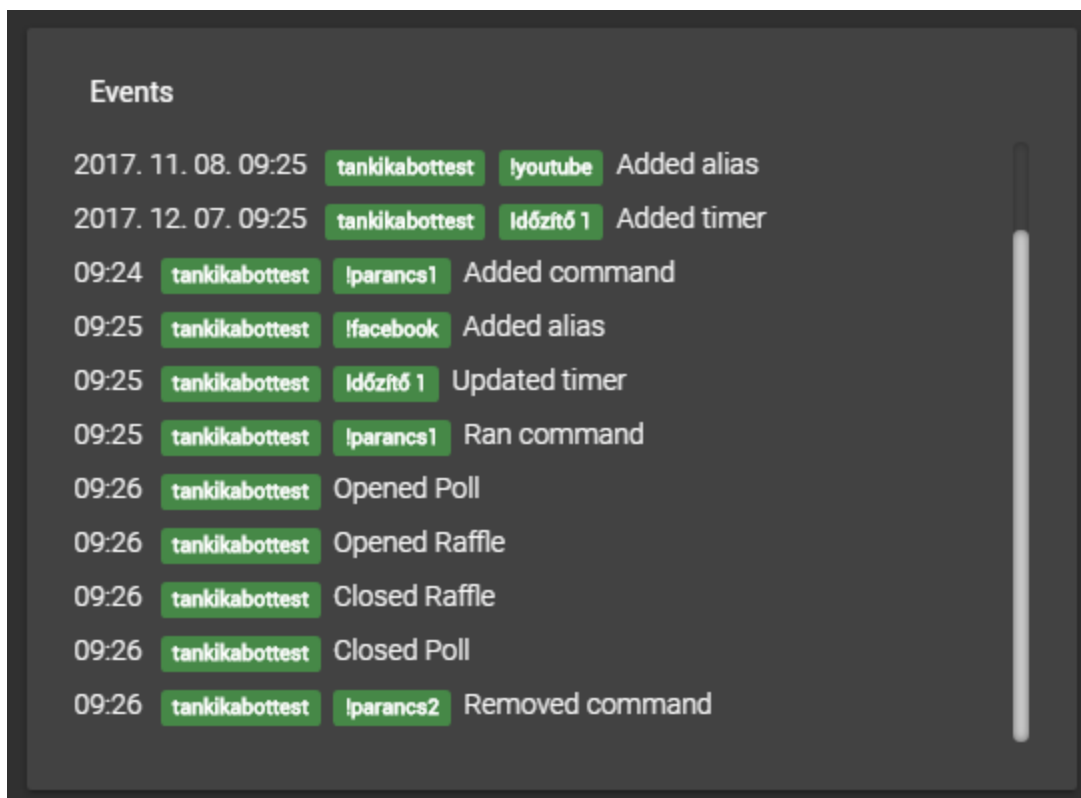
parancs. A MongoDB-be naplózott eseményeket végül GET paranccsal lehet lekérni az /api/events REST szolgáltatástól, ami visszaadja időrendi sorrendben az 50 legfrissebb eseményt.

7.3.1.2 Azonnali értesítés eseményről

Az eventLogger egy további bővítésével, egy rewriterrel lehetővé tettem, hogy az események keletkezésükkor ne csak, hogy az adatbázisba tárolódjanak, de azonnal megjelenjenek a felületen is, abban az esetben, ha éppen a Központi oldal van megnyitva a felhasználó böngészőjében. A probléma az volt, hogy, mivel a HTTP egy kapcsolat és állapotmentes protokoll, ezért valahogy meg kellett oldani, hogy a kliensoldal értesülni tudjon szerver oldali eseményekről. A két felvetődött megoldás a polling volt, vagyis időközönkénti lekérdezés a szerverre és a WebSocket szabvány [37] használata, ami néhány éve jelent meg a böngészőkben és kétirányú, full duplex kapcsolat támogatására képes TCP segítségével. A két megoldás közül a WebSocket-et választottam, hiszen ez állt közelebb a dolgozat témájához, a modern webes eszközök használatához. Mivel a Node.js API nem tartalmaz WebSocket támogatásra elemeket, ezért a funkció megvalósításához egy külső könyvtár, a socket.io-t használtam fel, ami kliens és szerver oldalon is megvalósítja a szabványt. Probléma volt a socket.io-s kapcsolat összekötése az express session-nel, hiszen azok teljesen külön jöttek létre, a socket.io-s kapcsolat létrejötte nem egy express által kezelt middleware kódban történt, hanem attól külön. Ennek áthidalására a socket.io kapcsolat létrejöttekor le az alkalmazás lenyúl a MongoDB adatbázisba, ahol az express által kezelt session-ök tárolódnak, és a session-höz tartozó felhasználó azonosítót kulcsként használva egy Map struktúrában tárolja le a socket azonosítóját, hogy a későbbiekben elő lehessen venni. Maga a WebSocket használata szerver oldalon a Winston logger már említett rewriter kiegészítésével történt, ami a log üzenetekhez tartozó meta adatok átírására szolgál, mielőtt a transporterek meghívódnának. Én egy saját rewriter-t adtam az én eventLogger nevű Winston példányomnak, ami, a meta adatban lévő userId alapján előkeresi a socket.io kapcsolatot a korábban említett Map struktúrából, és ha létezik nyitott kapcsolat, vagyis a felhasználó éppen a Központi oldalon tartózkodik, akkor elküldi a csatornán az üzenetet. Kliens oldalon az eseményeket megjelenítő DashboardEventsComponent komponens létrejöttekor azonnal csatlakozik a szerver által kiajánlott WebSocket-re, ekkor kerül be a kapcsolat a Map-be, illetve ez a komponens lesz az, ami értesül végül az új eseményről és megjeleníti azt.

7.3.1.3 Események a felületen

A felületen a már említett DashboardEventsComponent részeként az események egy egyszerű, görgethető Angular Material listaként jelennek meg. A sor elején az egyébként ISO 8601 [39] szabvány string időbélyeg, az Angular-ba beépített Date Pipe [40] segítségével van formázva, attól függően, hogy mai-e az esemény a dátumot és időt, vagy csak az időt mutatva. A mellette lévő színes háttérrel rendelkező címke szöveg minden esetben tartalmazza a csatorna nevét, illetve az eseményhez kapcsolódó egyéb meta adatot, például a parancs nevét, amihez kapcsolódik az esemény. A címkék háttere egyébként attól függően változik, hogy milyen szintű esemény történt, például error szintűnél piros, warn-nál sárga lesz a háttér. Utoljára az eseményhez kapcsolt üzenet jelenik meg.



12. ábra Események ablak

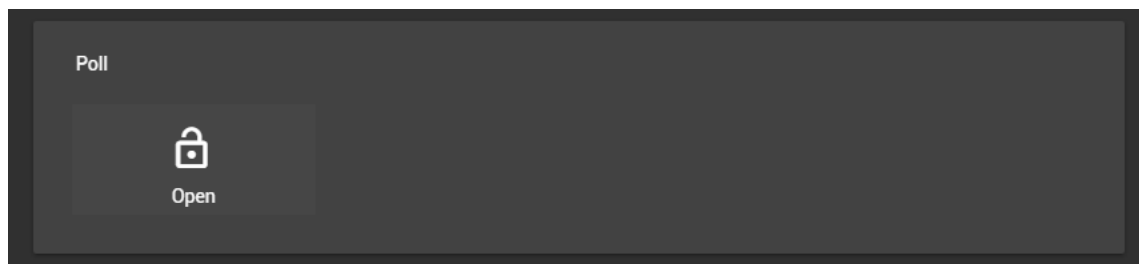
7.3.2 Szavazás (Poll)

A felületen elérhető következő komponens a szavazás funkció. A felhasználó ezen keresztül egy chat-en folyó szavazást tud kezelni, amin az aktuálisan a szobában lévő felhasználók megfelelő üzenetek küldésével tudnak részt venni. A szavazás

lebonyolítása szerver oldalon Redis segítségével történik, mivel az eredményeket nem kell megőrizni, ellenben azoknak gyorsan lekérdezhetőnek és frissíthetőnek kell lennie, mégpedig sűrűn, másodpercenként többször is akár. Az oldal betöltésekor a szavazás aktuális állapotát a szervertől lekéri a PollResolverGuard a komponens pedig ez alapján fog megjeleníteni nyitott vagy zárt állapotban. A szavazással kapcsolatos szolgáltatások a /api/polls URL-en érhetőek el a felület számára.

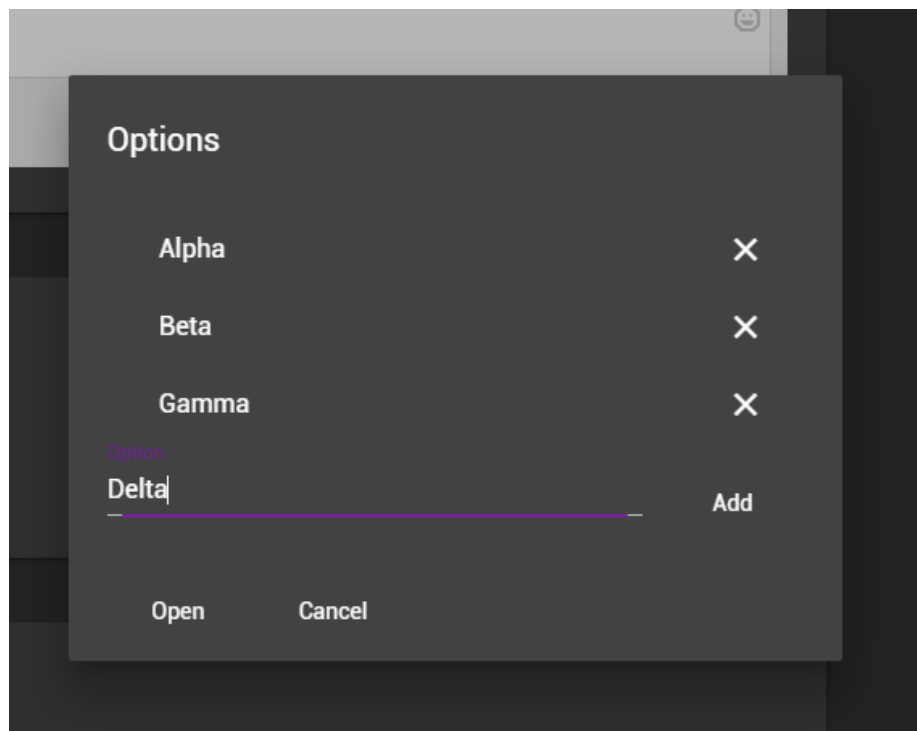
7.3.2.1 Megnyitás

Zárt szavazás esetén a felhasználó számára az egyetlen elérhető művelet az új szavazás indítása. Ezt a nyitott lakat ikonnal ellátott „Open” feliratú gombbal lehet megtenni.



13. ábra Szavazás panel zárt szavazáskor

A gomb megnyomása után megjelenik egy dialógus ablak, ahol a felhasználónak meg kell adnia azokat az opciókat, amikre a chat-en lévő felhasználók szavazni tudnak majd. Végül az Open gombra nyomva a felület elküldi az új szavazást tartalmazó kérést a szerver számára.

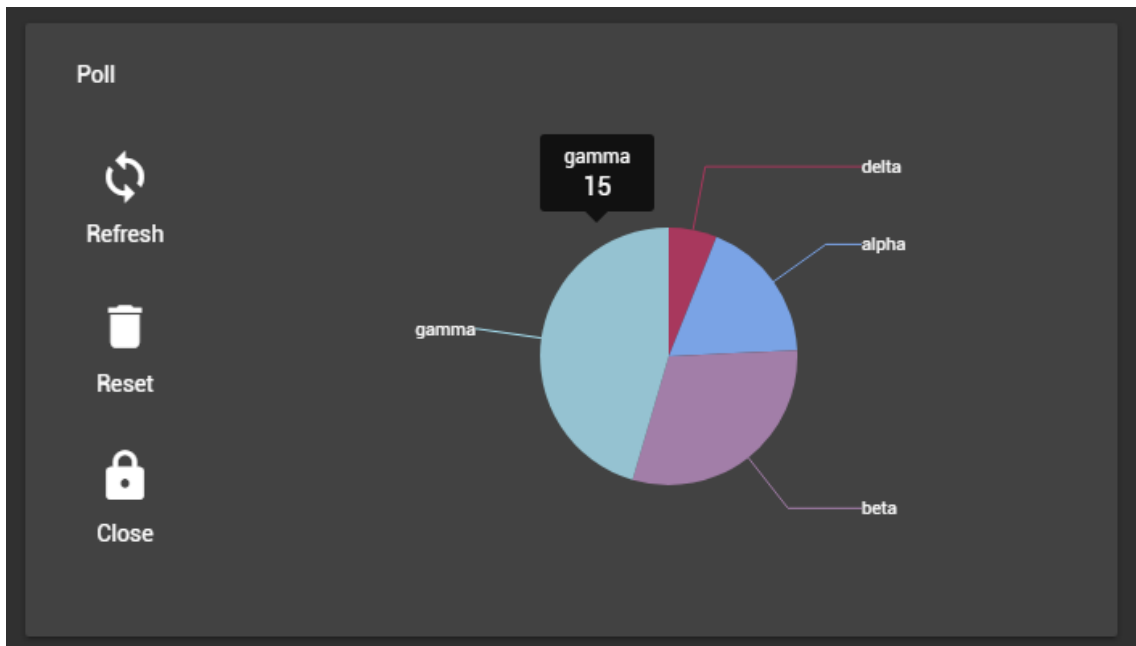


14. ábra Szavazás opciói dialógus ablak

Szerver oldalon az openPoll nevű middleWare az, aki elsőként kezeli a kérést. Ez egy batch utasításban hozzáadja a felhasználó azonosítóját a „polls” Redis halmazhoz, illetve a megadott opciókat 0-s értékkel a „poll:<userId>” rendezett halmazhoz. A redis hívás válaszából a middleware ellenőrzi, hogy ténylegesen megtörtént-e a hozzáadás, mert, ha nem, akkor az azt jelenti, hogy a szavazás már nyitva volt és hibát kell jelezni. Ha sikeres volt a szavazás elindítása az adatbázisban, akkor jelzi a bot számára is a szavazás elindulását, majd visszaadja a szavazás aktuális, üres állását a következő rétegnek. A következő middleware, a mapPollResults jelenléte azért szükséges, hogy a Redis által visszaadott struktúrát átalakítsa egy könnyebben feldolgozhatóvá. Sikeres szerver oldali hívás után a kliens oldal megnyitja a szavazás panelt és elérhetővé válnak a felhasználó számára a további funkció gombok.

7.3.2.2 Eredmények megjelenítése

Az eredmények megjelenítése egy kördiagram segítségével történik az ngx-charts [41] könyvtár használatával.



15. ábra A szavazás panel nyitott állapotban

A megjelenítéshez kapcsolódóan a Refresh gomb az, ami lekéri a szervertől az éppen aktuális eredményeket és befrissíti a diagramot velük. A Refresh gomb megnyomásakor szerver oldalon elsőként a `getPollResults` middleware az, amelyik a „poll:<userId>” rendezett halmazból pontokkal együtt lekéri az elemeket, majd továbbadja a következő rétegnek, ugyanannak a `mapPollResults`-nak, amelyik a szavazás megnyitásakor is az eredményeket átemeli egy könnyebben feldolgozható struktúrába.

7.3.2.3 Újraindítás

A „Reset” gomb hatására a szavazás újraindul, ami azt jelenti, hogy törölődnek az eddigi szavazatok, illetve a szavazók, de a szavazás nyitva marad. Szerver oldalon elsőként a `resetPoll` middleware réteg egy batch utasításban elsőként törli a „poll:voters:<userId>” kulcsot, vagyis az egész adatstruktúrát, ezzel törölve azokat a felhasználókat, akik már szavaztak. A batch utasítás másik részeként lekéri a „poll:<userId>” rendezett halmazból a szavazás opcióit. Ezek után frissíti a „poll:<userId>” minden elemét úgy, hogy azok értéke 0, majd átadja a szavazás aktuális, üres állását a következő rétegnek. A következő middleware a már látott `mapPollResults`, ami átalakítja az eredményt és visszaadja azt a felület számára. A felületen az új értékek hatására a kördiagram automatikusan frissül.

7.3.2.4 Lezárás

Az utolsó elérhető művelet a szavazás lezárása a „Close” feliratú gombbal érhető el a felhasználók számára, amivel törlődnek a szavazás eredményei és maga a szavazás is. A szerver oldalon a művelet végrehajtásáért a closePoll middleware felel. A réteg egyetlen batch utasítás keretén belül törli az adatbázisból a szavazás adatait. Elsőként törli a „poll:voters:<userId>” halmazt, ezzel megszüntetve a szavazók gyűjteményét. Ezek után törli a „poll:<userId>” rendezett halmazt is, amivel eldobásra kerülnek a szavazás eredményei. Végül törli a felhasználó azonosítóját a „polls” halmazból, ezzel lezárva véglegesen a szavazást. Ha az adatbázis sikeresen frissült, akkor a réteg utasítja a bot-ot is, hogy törölje a szavazást, majd jelzi a sikeres végrehajtást a felületnek. A felületen a törlést követően a komponens visszaáll zárt állapotba, újra csak az „Open” gomb lesz látható.

7.3.2.5 Bot megvalósítás

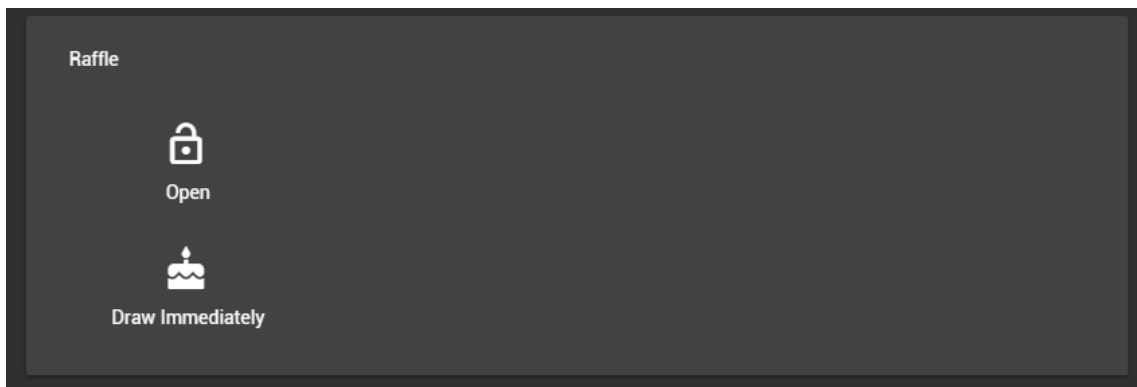
A bot oldalán a megvalósítás ebben az esetben azért érdekes, mert a bot maga nem kommunikál az adatbázissal, de ő az, aki a chat-en detektálja, ha valaki szavazott. Ennek megoldásaként a bot leszármazik a Node.js API részét képező EventEmitter osztályból, ami lehetővé teszi, hogy a bot eseményeket hozzon létre, amikre más modulok feliratkozhatnak. Ebben az esetben a bot tehát garantálja, hogy abban az esetben, ha szavazatot észlel a chat-en, akkor elsüt egy eseményt, amit a botManager osztály kezelni fog. Magának a szavazatoknak a detektálása egy reguláris kifejezés segítségével történik. Mikor a bot értesül róla, hogy a szavazás meg lett nyitva, akkor elkezdi figyelni az üzeneteket a csatornán és megpróbálja rájuk illeszteni a reguláris kifejezést. Ha sikerült, akkor capture group-ok segítségével meghatározza, hogy ki szavazott és milyen opcióra és ezt az információt adja tovább azoknak, akik feliratkoztak az eseményre. A botManager miután észleli az eseményt, először is ellenőrzi, hogy valóban folyamatban van-e szavazás az adott csatornán a „polls” halmaz segítségével. Ezek után megnézi, hogy a kapott felhasználó szavazott-e már a „poll:voters:<userId>” halmaz alapján. Végül pedig, hogy a megadott opció valóban létezik-e a „poll: <userId>” rendezett halmaz alapján. Ha minden kritérium teljesül, akkor elmenti a szavazás tényét az adatbázisba úgy, hogy hozzáadja a szavazó nevét a „poll:voters:<userId>” halmazhoz, illetve növeli az opció pontértékét egyel a „poll:<userId>” rendezett halmazban.

7.3.3 Nyereményjáték (Raffle)

A szavazások melletti másik Redis-t használó funkció a nyereményjáték, amiben a feliratkozók közül sorsolhat ki győzteseket a felhasználó. A komponens kezdeti állapotának lekéréséért ebben az esetben a `RafflesResolverGuard` felel, a funkcióval kapcsolatos szolgáltatások pedig a `/api/reffles` URL-en érhetők el.

7.3.3.1 Megnyitás

Ha az oldal betöltésekor nincs még futó nyereményjáték, akkor a komponens csukott állapotban fog inicializálódni. Ekkor a felhasználó rendelkezésére álló első művelet az új nyereményjáték megnyitása az „Open” gomb segítségével.

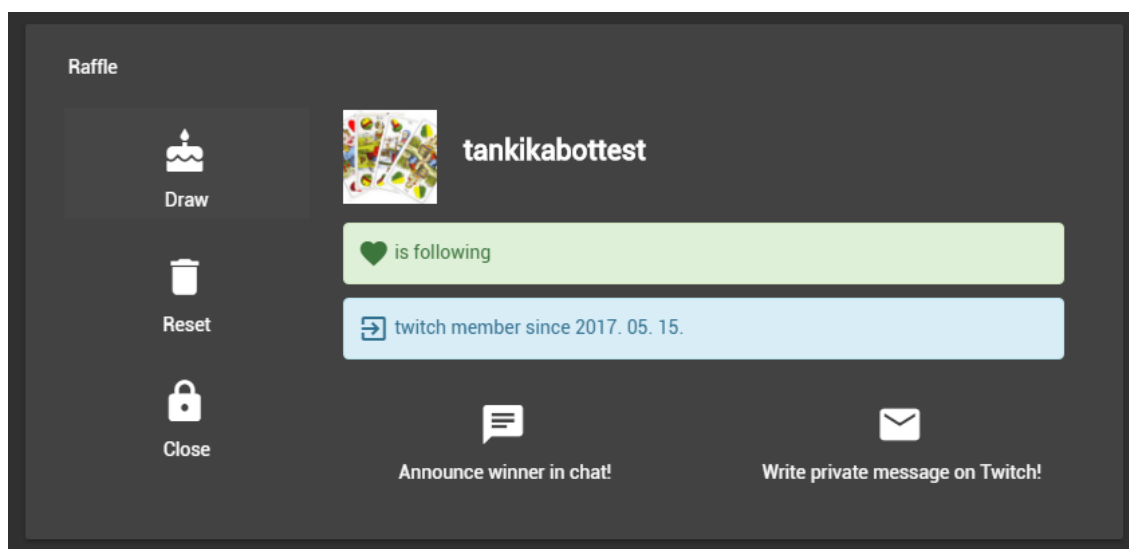


16. ábra Csukott állapotú nyereményjáték panel

Szerver oldali megvalósítása elég egyszerűen, az `openRaffle` middleware réteggel történt. A kérés érkezésekor a réteg megpróbálja hozzáadni a „raffles” halmazhoz a felhasználó azonosítóját és ellenőrzi a hívás válaszát. Ha nem sikerült hozzáadni az azonosítót a halmazhoz, akkor az azt jelenti, hogy már hozzá volt adva egyszer, tehát a szavazás már fut és hibát kell dobni. Ha sikeresen volt az adatbázis módosítás, akkor a middleware utasítja a bot-ot is a nyereményjáték megnyitására és visszatér a sikert jelző válasszal. A kliens oldal sikeres válasz esetén felfedi a felhasználónak a nyitott nyereményjátékkal kapcsolatos műveleteket.

7.3.3.2 Sorsolás

Az első ilyen művelet a sorsolás, aminek keretében az alkalmazás kiválaszt a nyereményjátékra feliratkozó chatel-ők közül egy nyertest.



17. ábra A sorsolás eredményének megjelenítése nyitott panelen

Szerver oldalon a hívást kezelő middleware-ek közül az első a drawRaffler. Ez elsőként ellenőrzi, hogy a csatornán valóban fut-e szavazás, vagyis session-höz tartozó felhasználó azonosító benne van-e a „raffles” halmazban és hibát dob, ha nincs. Ha a nyereményjáték valóban folyamatban van, akkor a middleware a nyereményjátékra feliratkozott felhasználókat tartalmazó „rafflers:< userId>” halmazon egy pop műveletet hajt végre. A pop művelet véletlenszerűen visszaadja és törli a halmaz egy elemét, ezzel teljes egészében azt a funkcionalitást adva, amire a sorshúzáshoz szükség volt. A második middleware réteg a kihúzott felhasználó neve alapján a hivatalos Twitch API-t használva lekéri a felhasználó adatait és, hogy követi-e a csatornát, majd ezeket visszaküldi a böngészőnek. Kliens oldalon, a panelen megjelennek a nyertes felhasználó adatai, illetve alattuk két újabb akció gomb. A második a „Write private message on Twitch” hatására a böngésző egyszerűen nyit egy másik ablakot a Twitch weboldal üzenetküldő oldalára, előre kitöltve a címzett részét a nyertes nevével. Az első akciógomb, az „Announce winner in chat!” hatására a bot kiírja chat-be a nyertes nevét. Erre egy külön szolgáltatás lett létrehozva, aminek az announceRaffleWinner middleware rétege egyszerűen utasítja a bot-ot a megfelelő utasítás kiírására.

7.3.3.3 Azonnali Sorsolás

Zárt nyereményjáték esetén a komponens másik elérhető művelete a nyertes azonnali sorsolása. Ezt szerver oldalon ismét két middleware réteg fogja kezelni, amiből az első, a drawImmediately fogja a konkrét nyertes kiválasztani. Ezt úgy teszi meg, hogy először a Twitch API-jának a segítségével lekéri a csatornán éppen tartózkodó

felhasználók listáját. Ezek után egy véletlenszerű index-et generál 0 és a felhasználók száma – 1 között, amivel aztán egyszerűen megcímzi a Twitch-től kapott tömböt és már meg is határozta a győztest. A második middleware a szolgáltatáson a már megismert `getRaffleWinnerData`, ami ismét lekéri a Twitch API-tól a felhasználó adatait és továbbítja azokat a kliens oldalnak. A felületen a győztes sikeres kiválasztása után ugyanaz az eredmény képernyő fog megjelenni ugyanazokkal az akciógombokkal, azzal a különbséggel, hogy a nyereményjáték nem lesz nyitott állapotú és továbbra is a megnyitás és az azonnali sorsolás marad a két elérhető művelet a felhasználó számára.

7.3.3.4 Újraindítás

A nyereményjáték újraindítását a felhasználó a „Reset” gomb megnyomásával kérheti. Ennek hatására szerver oldalon a `resetRaffle` middleware réteg egyszerűen törli a nyereményjátékokra feliratkozó felhasználókat tartalmazó „`rafflers:<userId>`” halmaz minden elemét. Az újraindítás eredményeként tehát a nyereményjáték nem zárul le, újra lehet rá jelentkezni, csupán az eddig jelentkezettek fognak törlődni.

7.3.3.5 Lezárás

A felhasználó számára elérhető utolsó művelet a lezárás, amit a „Close” gombra nyomva lehet végrehajtani. A szerver oldalon a kérést a `closeRaffle` middleware kezeli két lépésben. Elsőként törli a „`raffles`” halmazból a session-höz tartozó felhasználó azonosítót, ezzel törölve a nyereményjátékot. Másodikként törli a teljes „`rafflers:<userId>`” halmazt is, amivel a nyereményjátékban részt vevő felhasználók gyűjteményét üríti. A réteg végül pedig utasítja a bot-ot is a nyereményjáték lezárására.

7.3.3.6 Bot megvalósítás

A nyereményjátékok megvalósítása a bot oldalán nagyban hasonlít a szavazások megvalósításához. Mikor a nyereményjáték megnyílik, a bot elkezd figyelni a chat-en érkező üzeneteket egy reguláris kifejezés segítségével. Ha észleli, hogy egy felhasználó csatlakozik a nyereményjátékhoz, akkor a reguláris kifejezés `capture group`-ja segítségével meghatározza a felhasználó nevét, majd azt egy eseménnyel továbbítja a `botManager` számára. Ebben az esetben a `botManager` működése is jóval egyszerűbb. Mikor megkapja az eseményt, akkor a feliratkozó nevét egyszerűen hozzáadja a „`rafflers:<userId>`”, mivel abba kétszer úgysem kerülhet bele a halmaz tulajdonságai miatt.

7.4 Parancs felület (Command)

A parancsok kezelése a bot egyik alap funkciója, ezeknek a Parancs felületen lett meg a helye. Ez egy külön nézet, a menüsorban lévő Commands gomb megnyomásával navigálhatnak ide a felhasználók. A központi oldalhoz hasonlóan, itt is több resolve guard-ot regisztráltam, aminek mindegyike az oldal egy-egy alfunkciójának az inicializálásához szükséges adatokért felel:







- `CommandsResolverGuard`: A Központi oldalon beregisztált guard-dal megegyező osztály, a csatornához tartozó parancsokat adja vissza.
- `TimersResolverGuard`: A parancs időzítőket kéri le a szervertől.
- `AliasesResolverGuard`: A parancs fedőneveket adja vissza a szervertől lekérve.

7.4.1 Parancsok (Commands)

A parancsok képezik a botok működésének egyik alapját. A parancsok rövid szöveges üzenetek felkiáltó jellel kezdődve, amiket a felhasználók beírhatnak a chat-be, ezekre válaszul pedig a bot elküldi ugyanabba a csatornába az előre beállított választ. A parancsokhoz tartozó szolgáltatások mindegyike az `/api/commands/commands` URL-ből kiindulva érhetők el.

7.4.1.1 Megjelenítés

Az első művelet a parancsokkal kapcsolatban a listázás. Ezt, a már korábban említett, `CommandsResolverGuard` végzi, tehát minden alkalommal megtörténik, amikor a nézet megnyílik. A nézet betöltésekor késlekedés nélkül beöltődnek az erre a célra felkonfigurált, Angular Material-os táblába a rendelkezésre álló parancsok, illetve a hozzájuk tartozó műveletek.

Commands		
Name	Text	
!social	https://www.facebook.com/	<input type="checkbox"/>  
!welcome	Welcome to the channel!	<input checked="" type="checkbox"/>  
!info	BME, 2017	<input checked="" type="checkbox"/>  

18. ábra Parancsok megjelenítése

7.4.1.2 Létrehozás

Új parancsokat létrehozni a már létező parancsokat tartalmazó tábla melletti kis form segítségével tudunk. Itt meg kell adni az új parancs nevét, illetve az üzenetet, ami elküldésre kerül a parancs futtatásakor, majd az Add gombra nyomva elküldjük az alkalmazásnak a létrehozni kívánt parancsot. Szerver oldalon a hívás több middleware rétegen is keresztülhalad. Az első kettő a `getCommandByName` és a `getAliasByName` a híváshoz tartozó session felhasználójánál próbálja meg megkeresni a megadott néven a már létező parancsot, vagy fedőnevet, és azt továbbadja a következő middleware-nek. Az utolsó middleware a `createCommand` megnézi, hogy az előző kettőből talált-e valamelyik réteg a megadott névvel valamit, ha igen, akkor hibát jelez, ha nem akkor pedig lementi adatbázisba az új parancsot és utasítja a csatornához tartozó botot, hogy állítsa be magának az új parancsot. A sikeres művelet után a felületen frissül az összes olyan komponens, amelyiknek szüksége van a parancsok aktuális listájára.

Add Command

Name *

! creator

Text

Perlaki Tamás

Add

19. ábra Parancs létrehozása

7.4.1.3 Törlés

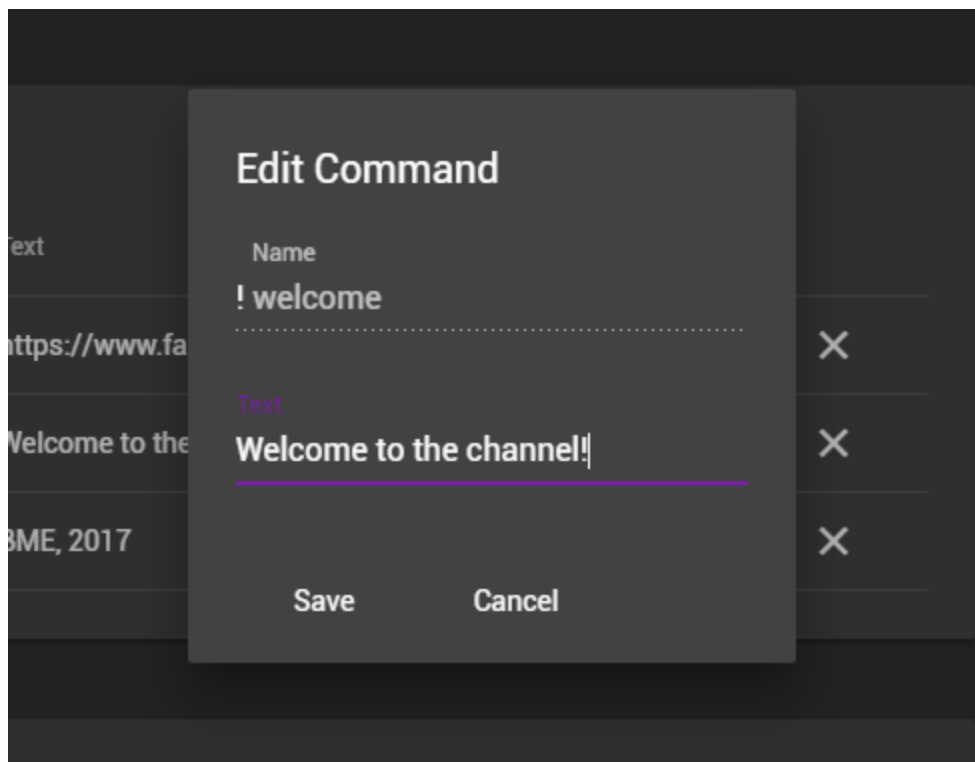
A következő művelet, ami a felhasználók rendelkezésére áll, egy látszólag egyszerű dolog, a parancsok törlése. Ezt kliens oldalon a parancs sorában lévő X jelre kattintva lehet megtenni, amikor is feljön egy Angular Material-lal készített dialógus ablak annak megerősítésére, hogy tényleg törölni akarja-e a felhasználó a parancsot. Ha a felhasználó megerősíti a törlést, akkor szerver oldalon a `getCommandById` middleware réteg előkeresi a törlendő parancsot adatbázisból és átadja a következő rétegnek. A tényleges törlést végző `deleteCommand` middleware-nek összetettebb feladata van, ugyanis, miután törölte az adatbázisból a parancsot, még frissítenie kell azokat az időzítőket, amikhez a parancs meg van adva, illetve törölnie kell azokat a fedőneveket, amik a parancsra mutatnak. Ehhez kapcsolódóan egy érdekesebb kódrészlet az időzítők frissítése, amiben minden olyan Timer típusú dokumentum `commands` lista attribútumából törlődik a törölt parancs, ezzel karbantartva a dokumentumok közötti kapcsolatot. Itt láthatjuk működés közben a MongoDB API-ra erősen építő mongoose API-t, mennyire eltér a hagyományos SQL parancsoktól:

```
Timer.update(  
  {user: req.session.userId, commands: removedCommand._id},  
  {$pull: {commands: removedCommand._id}},  
  {multi: true}  
)
```

Miután frissül az adatbázis, a middleware utasítja a felhasználóhoz tartozó botot, hogy frissítse önmagát, ezek után pedig, ha nem történt hiba, a felület újratölti a parancsok listáját a megfelelő komponensekben.

7.4.1.4 Módosítás

A parancsok módosítására, a felületen, két helyen van lehetőség, azonban mind a kettő ugyanazt az egy darab, általános UPDATE URL-t használja. Az első ilyen pont a parancs sorában lévő Angular Material slide-toggle elem, a Material Design-nál, iOS-en már megszokott csúszó, ki-be kapcsoló gomb. A második pont a parancs sorában lévő ceruza ikonra kattintva előhozható Szerkesztés dialógus. Itt a felhasználó lecserélheti a parancshoz tartozó szöveget, ami futtatáskor kiírásra kerül, azonban a parancs nevét nem.



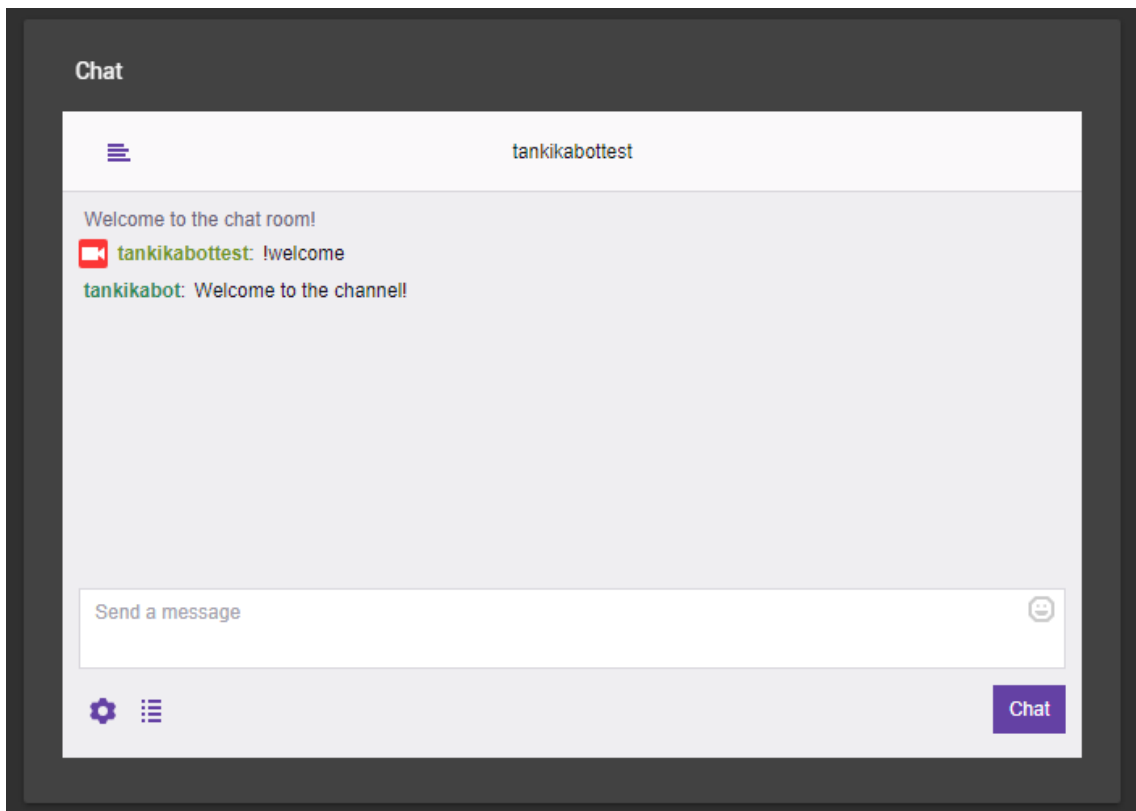
20. ábra Parancs szerkesztése dialógus

Ahogy említettem, szerver oldalon mindkét pont végül azonos szolgáltatást használ, ami egy általános UPDATE szolgáltatás, a paraméterben kapott parancs adott tulajdonságait képes lecserélni. A szolgáltatás első middleware-je itt is a `getCommandById`, ami a megtalált parancsot ebben az esetben az `updateCommand` middleware-nek adja tovább. Az `updateCommand` middleware feladata szintén nem csupán annyi, hogy az adatbázisban frissítse a parancsot a megadott adatokkal, hanem ezek után még utasítania kell a bot-ot az időzítők és fedőnevek frissítésére is, hiszen egy kikapcsolt parancsot nem használhat egyik sem.

7.4.1.5 Bot megvalósítás

A parancsok megvalósítása a bot modulban egy egyszerű objektum segítségével történik, amiben az attribútumok a parancsok nevei, míg az értékek a parancsok szövegei. A bot folyamatosan figyeli a csatornára érkező üzenetet és egy reguláris kifejezés segítségével ellenőrzi, hogy azok egy parancs formátumát veszik-e fel, tehát felkiáltó jellel kezdődnek-e. Ha igen, akkor a reguláris kifejezés catching group-ja segítségével a bot meghatározza a parancs nevét, amit kiadtak és megpróbálja előkeresni a regisztrált parancsok között. Ha megtalálta, aki a parancs, amit kiadtak be van regisztrálva a bot-hoz és lefuttatja, vagyis a megadott szöveget elküldi válaszként a

chat-be. Ha nem található a parancs, akkor nem történik semmi, csupán alacsony szinten logolja a tényt.







21. ábra Parancs futtatása a chaten keresztül

7.4.2 Parancsidőzítők (Timers)

A parancsokat kiegészítő egyik funkció a parancsidőzítő, ami, ahogy a neve is mutatja parancsok periódikusan ismétlődő futtatására képes. Az időzítőkkel kapcsolatos szolgáltatások, mivel a parancsokhoz kötődnek, ezért a `/api/commands/timers` URL-en érhetők el.

7.4.2.1 Megjelenítés

A parancsokhoz hasonlóan az első művelet ebben az esetben is a listázás, ami itt is egy `resolve guard`, a `TimersResolverGuard` segítségével történik. Az időzítők a már látott módon egy Angular Material könyvtárbeli táblázatba töltődnek be az oldal megnyitásakor. A táblázatban nincsenek listázva az időzítőhöz tartozó parancsok, csupán azoknak a számossága van feltüntetve.

Timers			
Name	Trigger Time	Number of Commands	
Üdvözlés	5:00	1	<input checked="" type="checkbox"/>  
Információk	10:00	2	<input type="checkbox"/>  

22. ábra Időzítők megjelenítése

7.4.2.2 Létrehozás

Időzítőket létrehozhat a felhasználó a táblázat mellett található kártyán, ahol meg kell adnia a létrehozandó időzítő nevét, illetve azt a periódusidőt percekben, amennyi időközönként az időzítőhöz kapcsolt parancsok le fognak futni. Az Add gombra nyomva elküldésre kerül a szervernek az új időzítő, ahol is két middleware fogja kezelni a hívást. Az első a `getTimerByName` a név alapján próbálja megtalálni az időzítőt az adatbázisban és továbbadni a következő rétegnek, mivel annak egyedinek kell lennie. A következő réteg a `createTimer` middleware, ami hibát jelez, ha az előzőekben sikerült megtalálni az időzítőt, ellenkező esetben pedig létrehozza azt az adatbázisban. Ebben az esetben az időzítő nem fog azonnal hozzáadódni a felhasználóhoz tartozó bot-hoz, ugyanis addig, amíg nincsenek parancsok az időzítőhöz kötve nem tud mit futtatni, így nincs értelme. Sikeres lefutás után a felület frissíti az időzítők táblázatát egy újabb listázás művelettel.

Add Timer

Name *

Információk

Trigger Time In Minutes *

10

Add

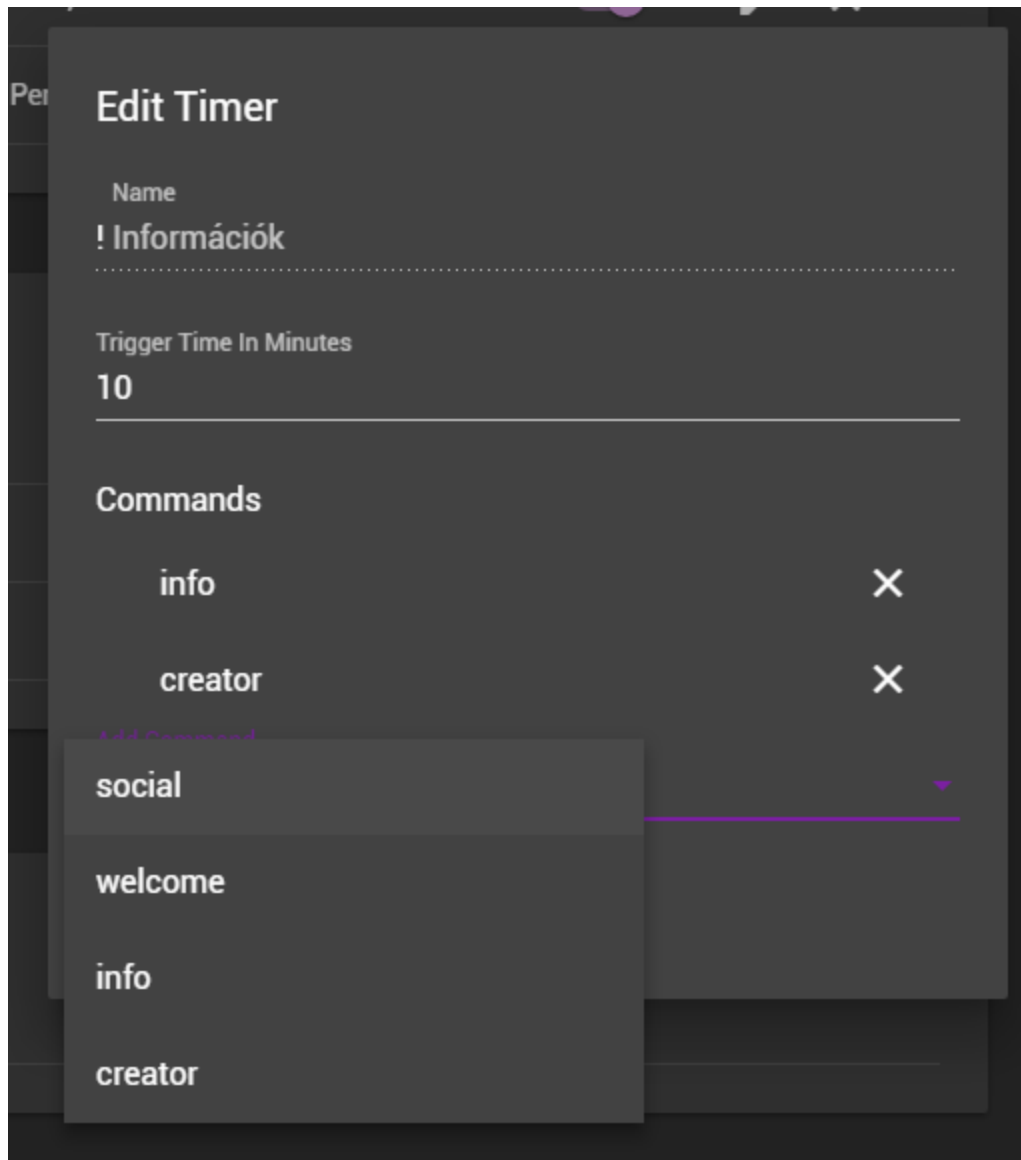
23. ábra Időzítő hozzáadása

7.4.2.3 Törlés

A következő művelet az időzítők törlése a táblázat egy sorában lévő X jelre kattintva indítható el. A jelre kattintva megjelenik a már korábban bemutatott törlés megerősítése dialógus ablak, amit elfogadva a kliens oldal elküldi a törlés utasítást a szervernek. A törlés szolgáltatáson az első middleware a `getTimerById` megpróbálja megkeresni a paraméterben kapott azonosító alapján az időzítőt az adatbázisban, majd továbbadja azt a következő rétegnek. A következő middleware a `deleteTimer` egyszerűen törli az adatbázisból az időzítőt, illetve utasítja a bot-ot is, hogy törölje az időzítőt, abban az esetben, ha időközben aktiválva lett volna. Ebben az esetben nincs szükség a dokumentumok közti kapcsolatok frissítésére, mivel a Timer típusra nem mutat másik típusból referencia, a Timer az, ami a referenciákat tartalmazza más típusokra. Sikeres művelet után a kliens oldal újra lekéri az időzítők listáját és frissíti az őket tartalmazó táblázatot.

7.4.2.4 Módosítás

Az időzítők módosítása itt is két külön módon történhet, hasonlóan a parancsokéhoz és ugyanúgy egy közös UPDATE szolgáltatást használ mindkettő. Az első itt is a már látott csúszkáló gomb, amivel az időzítő működését ki-be lehet kapcsolni. A második egy szerkesztő dialógus ablak, ahol meg lehet változtatni az időzítőhöz tartozó időt, illetve a hozzá tartozó parancsokat lehet hozzáadni, eltávolítani. A parancsokat egy lenyíló listából lehet hozzáadni az időzítőhöz, akkor is, ha a parancs nincsen engedélyezve. Ebben az esetben a parancs hozzá lesz rendelve az időzítőhöz, de az időzítő futásakor nem fog lefutni, illetve az az időzítő, amelyikhez nem tartozik egyetlen engedélyezett parancs sem a bot szempontjából ugyanúgy viselkedik, mint egy olyan, amihez nincs hozzárendelve egy parancs sem. Az időzítő tehát létrejöttkor még nem aktív, hiszen nincs hozzárendelve egy parancs sem, akkor fog beregisztrálódni a bot-hoz, amikor legalább egy engedélyezett parancsot hozzáadtunk. Miután az időzítőt számunkra tetszőleges módon beállítottuk, a Save gombra nyomva elküldi a böngésző a szervernek a módosított időzítőt. Szerver oldalon elsőként a már bemutatott `getTimerById` réteg fogja kezelni a kérést, majd átadja a vezérlést az `updateTimer` middleware-nek. Ez a réteg egyszerűen tárolja az adatbázisban a módosításokat, majd utasítja a bot-ot az időzítő frissítésére. A kliens oldal sikeres művelet esetén frissíti az időzítőket tartalmazó táblázatot.



24. ábra Időzítő szerkesztése

7.4.2.5 Bot megvalósítás

Az időzítők esetében külön érdekes és fontos a bot modulban való implementációjuk. Megvalósításukhoz a Node.js által beépítve támogatott `setInterval` nevű funkcionalitást használtam, ami egy olyan függvény, ami egy függvényt és egy számot vár paraméterként. A `setInterval` sikeres hívásakor beállít egy Node.js időzítőt, ami a második paraméterben, milliszekundumban átadott időközönként végrehajtja az első paraméterben átadott függvényt végtelen ciklusban. Visszatérési értéke egy azonosító, aminek a segítségével az időzítőt törölni lehet. Ennek nagy előnye természetesen, hogy miközben az idő leteltére várunk az alkalmazás nem blokkolódik, fut tovább, a kívánt idő leteltével mégis végrehajthatnak az általunk megadott utasítások. Az időzítők használatával persze adott a probléma, hogy azokat meg is kell

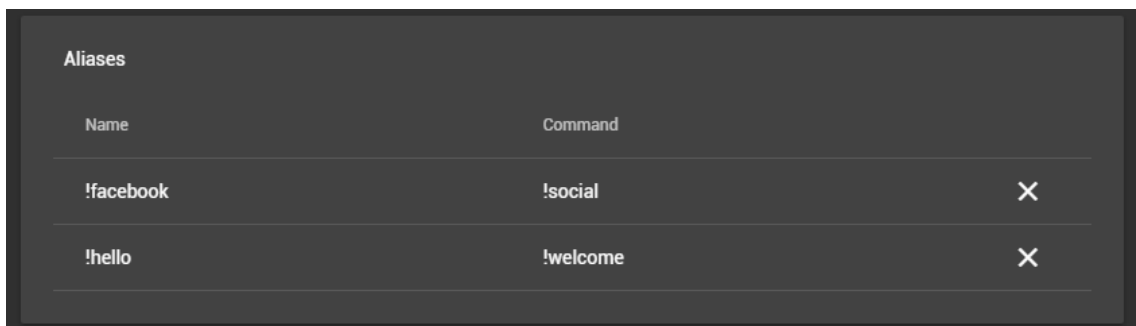
szüntetni, hiszen az alkalmazásunk egy idő után összeomlana, mivel a memória elfogyna, az egyéb funkciók kiéheződnének. stb. Ennek orvoslására az időzítők beállításakor azoknak a már említett azonosítóját eltárolja a bot egy objektumban, ahol az időzítők neve az attribútumok, értékeik pedig az azonosítók. Ezek után, mikor a bot egy olyan utasítást kap, hogy töröljön egy parancsidőzítőt, akkor megkeresi a parancshoz tartozó Node.js időzítő azonosítóját az objektumból, majd azzal meghívja a beépített `clearInterval` függvényt, ami leállítja az időzítőt.

7.4.3 Parancs fedőnevek (Aliases)

A parancsidőzítőkön kívül a másik parancsokhoz köthető kiegészítő funkció a parancs fedőnevek, amik nem mások, mint a parancsoknak adott további nevek, alias nevek. Ez az előbbieknél egy jóval egyszerűbb funkcionalitás. Mivel ez is a parancsokhoz köthető, ezért a hozzá tartozó szolgáltatások a `/api/commands/aliases` URL-en érhetők el.

7.4.3.1 Megjelenítés

A fedőnevek első művelete a már megszokott listázás, az eddigiekhez hasonlóan egy Angular Material táblázatban. Ahogy az eddig látott funkcióknál így itt is a fedőnevek megszerzéséről egy külön guard gondoskodik, az `AliasesResolverGuard`. A táblázatban megjelenítésre kerül a fedőnév neve illetve a parancs neve, ami le fog futni a fedőnév hatására.



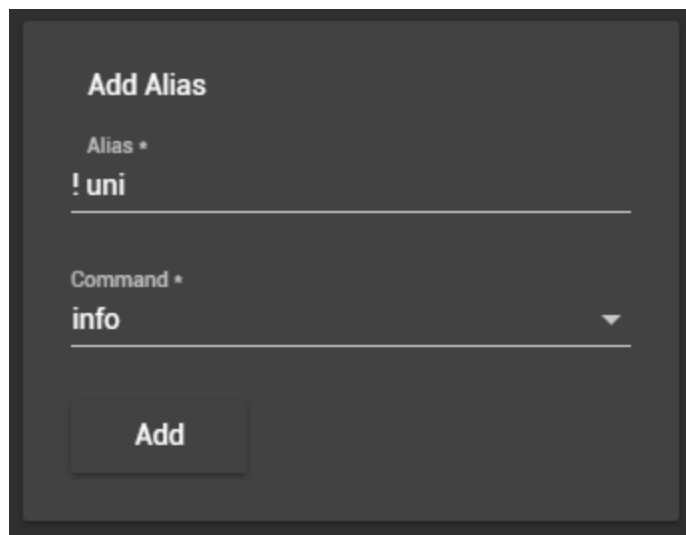
Aliases		
Name	Command	
!facebook	!social	X
!hello	!welcome	X

25. ábra Fedőnevek megjelenítése

7.4.3.2 Létrehozás

Új fedőnevet létrehozni a jobb oldali „Add Alias” fejlécű form segítségével tudunk. Itt meg kell adnunk egy nevet, amit a felhasználók majd használhatnak, illetve egy legördülő listából ki kell választanunk azt a parancsot, amire szeretnénk, hogy a fedőnév vonatkozzon. Az Add gomb megnyomásakor a kliens elküldi a létrehozni

kívánt fedőnevet a szerver oldalnak, ahol három middleware-n keresztül dolgozódik fel. Az első két réteg a már látott `getCommandByName` és `getAliasByName` middleware-k, amik a megadott név alapján próbálnak az adatbázisból előkeresni egy ugyanilyen nevű, már létező parancsot, vagy fedőnevet, a talált eredményt pedig továbbadják a következő rétegnek. A lényegi munkát végző réteg a `createAlias`, ami elsőként ellenőrzi, hogy talált-e az előző két réteg valamit, mert ha igen, akkor jelzi a hibát, hiszen a névnek egyedinek kell lennie. Ha nem talált az előző két réteg megegyező nevű dokumentumot, akkor a `createAlias` réteg létrehozza az adatbázisban az új fedőnevet és értesíti a bot-ot az új fedőnévről, majd pedig jelzi a felületnek az utasítás sikeres végrehajtását. A felület abban az esetben, ha nem történt hiba frissíti a fedőnevek listázására szolgáló táblázatot.



26. ábra Fedőnév hozzáadása

7.4.3.3 Módosítás

Mivel a fedőnevek funkció alapból egy elég egyszerű dolog, ezért itt úgy döntöttem, hogy a módosítás műveletét elhagyom. Ha a felhasználók módosítani akarnak egy fedőnevet, akkor azt ki kell törölniük, majd újból létrehozni.

7.4.3.4 Törlés

A fedőnevek törlése szintén egy nem túl bonyolult művelet. A felületen a felhasználó a törölni kívánt fedőnév sorában lévő X-re kattintva kezdeményezheti, amit a már megszokott felugró ablakban meg kell erősítenie. Sikeres megerősítés után a kliens leküldi a szervernek a kérést, amit azután először a `getAliasById` middleware dolgoz fel. Ez a réteg megpróbálja a fedőnevet a paraméterben kapott azonosító alapján megtalálni az adatbázisban, majd az eredményét továbbadja a következő rétegnek, a

deleteAlias-nak. A deleteAlias middleware ellenőrzi, hogy az előző rétegnek sikerült-e megtalálnia a fedőnevet és hibát dob, ha nem. Ellenkező esetben törli a fedőnevet az adatbázisból, frissíti a bot-ot, majd elküldi a sikeres választ a kliens oldalnak. A kliens oldal sikeres válasz hatására frissíti a fedőneveket listázó táblázatot.

7.4.3.5 Bot megvalósítás

A funkció bot oldali megvalósítása nagyban hasonló a parancsok megvalósításához és eléggé összefolyik vele. Az eddigiekben látott módon a fedőnevek is egy objektumban tárolódnak a bot-on belül, ahol az objektum egy attribútuma a fedőnév neve, értéke pedig annak a parancsnak a neve, amire a fedőnév hivatkozik. Annak detektálása, hogy mikor kell használni a fedőnevet ugyanabban a kódrészletben van, ahol a parancsok fogadása, ugyanazzal a reguláris kifejezéssel. Miután a reguláris kifejezés segítségével a program észreveszi, hogy a felhasználó egy parancsot akar futtatni, a már leírt módon ellenőrzi, hogy a parancs létezik-e a parancsok objektumában. Ha a parancs nem létezik, akkor valójában a korábban leírtakkal ellentétben nem fog még leállni az ellenőrzés, hanem a kód azt is megnézi, hogy nincs-e véletlen a megadott néven egy fedőnév. Ha van ilyen fedőnév az objektumban, akkor a hozzá tartozó parancsnév alapján a parancsokhoz tartozó objektumból kikeresi milyen szöveget kell elküldeni a chat-be. Ha se parancs, se fedőnév nem szerepel a bot-ban, akkor pedig a már leírt módon a bot naplózza a tényt, hogy ismeretlen parancsot kapott, de egyébként nem csinál semmit.

7.4.4 Kommunikáció a komponensek között

Egy külön érdekes probléma volt a kommunikáció a parancsok felületen lévő különböző komponensek között. Ez jelen esetben egyirányú kommunikációt jelent a parancsok komponens és a másik kettő között, de a lehetőség megvan, hogy bármelyik tudjon kommunikálni bármelyikkel üzenetszórással. A kommunikációra azért van szükség, mert az időzítők és a fedőnevek ráépülnek a parancsokra. Ahhoz, hogy velük műveleteket lehessen végezni sokszor rendelkezésükre kell álljon a parancsok aktuális listája, például fedőnév létrehozásánál a lenyíló listában van az összes parancs, ami viszont időben változhat és ezekről a másik két komponenst értesíteni kell. A kommunikáció alaptól azért nehézkes, mert az Angular komponens orientált filozófiája szerint a komponensek, csakúgy, mint az objektumok, egymástól független, amennyire lehet, elzárt egységeket alkotnak, amik viszont képesek lehetnek bizonyos módokon

kommunikálni. Esetemben egy olyan típusú kommunikációt kellett megoldani, amiben az egyik komponensben lévő változásról az összes többi értesülhet. Ehhez egy Angular service-t, a `CommandCommunicatorService`-t hoztam létre, amit az egyéb szolgáltatásokkal ellentétben nem tettem elérhetővé a teljes alkalmazáson belül, hanem a parancs nézetet adó `CommandComponent`-hez kötöttem, ezzel azt érve el, hogy csakis ezen a komponensen és a gyerekein belül legyen értelmezve. A service magja az Angular által ajánlott és sok helyen használt RxJS könyvtár két `Subject` típusú objektumából áll. A `Subject` lényege jelen esetben, hogy az observer pattern szerint fel lehet rájuk iratkozni, illetve eseményeket lehet rajtuk elsütni. A két általam használt `Subject`-ből az egyik arra szolgál, hogy egy parancs törlését jelezni lehessen vele, a másik pedig arra, hogy a parancslista bármilyen megváltozását lehessen vele jelezni. A két `Subject` ezek után ki lett ajánlva a `Service`-en, valamint a `Subject`-eken eseményeket létrehozó két metódus is. Ezekre a `Subject`-ekre tehát az időzítő és a fedőnév komponens létrejöttékor feliratkozik, a parancs komponens pedig a megfelelő művelet végrehajtásakor, például egy parancs törlésekor generál valamelyik `Subject`-en egy eseményt, amit a fogadó komponens le tud kezelni, például frissíti a parancsok listáját, amit karbantart.

8 Tesztelés

Az alkalmazás tesztelése műveletek sora, amelyek segítségével:

- Megakadályozhatjuk már létező kódrészletek elromoljanak, vagyis regressziós hibák keletkezzenek
- Ellenőrizhetjük kódunk működését várt és váratlan helyzetekben egyaránt.
- Tervezési és implementációs hibákat fedhetünk fel az alkalmazásunkban. Egy olyan kód esetében ugyanis, amit nehéz letesztelni sokszor hibás fejlesztői döntések állnak, amiket általában jobb újragondolni.

Az alkalmazás tesztelését érdemes kliens és szerver oldalon is elvégezni, amit én az Angular teszt útmutatója segítségével [42] végeztem. Mivel a szerver oldal TypeScript segítségével készült, ezért az útmutatóban lévő ajánlások és technológiák jó része itt is alkalmazható volt.

8.1 Unit tesztelés

A készített tesztek első fajtája a unit teszt volt. A unit teszt egy kis kódrészlet, ami egy másik, ugyancsak kis kódrészletet ellenőriz, ami tipikusan egy dologért felel. A teszt során a tesztelendő kód függőségeit tipikusan helyettesítjük, más komponenstől függetlenül nézzük a működését. A unit tesztek további fontos tulajdonsága, hogy automatizáltak, nagyon gyorsan futnak le, mert egy nagyobb alkalmazás esetén több ezer lehet belőlük és napjában többször is kellhet futniuk.

Az Angular két fajta unit teszt készítését támogatja. Az első fajta az Angular komponensek és direktívák tesztelése, aminek a segítésére a keretrendszer fejlesztői létrehozták az úgynevezett „testing utility API”-t, ami különböző szolgáltatásokból áll a tesztek könnyebb elkészítéséhez. Az API alapja a TestBed osztály. Az osztály segítségével tesztmodulokat hozhatunk létre, amik hasonlóan működnek az alkalmazás felépítéséhez használt modulokhoz. Ezekbe is importálhatunk külső modulokat, beregisztrálhatunk szolgáltatásokat, komponenseket, direktívákat. A tesztelendő komponensünk ennek a tesztmodulnak a környezetében fog izoláltan létrejönni, és ebben a környezetben fognak lefutni a teszteseteink. Mivel unit tesztről van szó, ezért a

komponens függőségeit helyettesíteni akarjuk, nem a valódi implementációkat akarjuk használni. Ezeket a helyettesítőket a tesztmodulba tudjuk beregisztrálni, amik függőség injektálás segítségével fognak átadódni a komponensnek, annak létrejöttékor.

A második fajta unit teszt, a szolgáltatások, csővezetékek ellenőrzésére szolgál teljesen izolált unit teszt. Ez az, amit hagyományos értelemben is unit tesztnek mondhatunk, ami a keretrendszerrel függetlenül tesztel egy kódrészletet, miközben annak függőségeit helyettesíti valamilyen módon. Mivel ezek a tesztek már nem függenek a keretrendszerrel, ezért ezeket a fajta teszteket szerver oldalon is lehet használni ugyanazokkal a teszteszközökkel.

8.2 End-to-end tesztelés

A unit teszteken kívül még egy teszt fajta szerepel az Angular ajánlásában, ez pedig az end-to-end teszt, ami leginkább, az egyéb irodalmakban fellelhető, rendszer tesztnek felelhet meg. Az end-to-end tesztek során nem az alkalmazás részeit vizsgáljuk külön-külön, hanem azokat egyben, a legfelső rétegtől a legalsóig. Egy ilyen teszt általában egy funkcionalitásra koncentrál és felhasználói cselekedet egy sorát játssza végig, például bejelentkezés, új parancs hozzáadása, parancslista ellenőrzése. A tesztek a működésüket nem az alkalmazás programozási interfészeinek a segítségével végzik, hanem, mintha emberi személyek lennének, a böngészőn keresztül hajtják végre a műveleteket. Ennek lehetővé tétele két eszköz segítségével történik. Az első, és legfontosabb, a Webdriver [43], ami képes a böngészőket, az általuk kiajánlott interfészeket, programozottan vezérelni, pontosan ugyanúgy, mintha egy ember tenné egér és billentyűzet segítségével. A WebDriver API-n keresztül a böngészőkben felépített DOM elemek tudjuk megkeresni, illetve ezeken műveleteket végezni programozottan, mint például egérgattintás, billentyű leütések, stb. A második eszköz a tesztek elkészítéséhez a Protractor, ami a WebDriver-t egészíti ki az Angular tesztelését megkönnyítő funkciókkal, mint például az alkalmazás állapotátmeneteinek a megvárása két művelet végrehajtása között.

9 Biztonság

Mivel az alkalmazás nyíltan elérhető, webes megjelenítő réteggel rendelkező program, eredendően sok potenciális támadási felülettel rendelkezik. Annak érdekében, hogy ezeket minél jobban ki tudjam küszöbölni, több ajánlást, illetve eszközt használtam fel és ezek mentén próbáltam elérni minél biztonságosabb működést.

9.1 Angular

Elsőként az Angular honlapján ajánlott rövid lista [44] alapján tettem meg az ajánlott óvintézkedéseket, amik tehát a kliens oldalra vonatkoznak.

9.1.1 Cross-Site Scripting (XSS)

Az első támadási lehetőség, amire az Angular felhívja a figyelmet, az a Cross-Site Scripting [45] támadás. Ennek lényege, hogy a támadó a szerver vagy a kliens oldalon keresztül, saját maga által készített, futtatható kódot (például `<script>` tag közötti kód, vagy `` módon megadott kód) tesz elérhetővé egy honlapon. Mikor más felhasználók az oldalra látogatnak, akkor azok böngészője automatikusan lefuttatja ezeket a kódrészleteket, amivel a felhasználók tudta nélkül, számukra potenciálisan veszélyes műveleteket hajtanak végre. Az ajánlás leírja azonban, hogy ezek ellen a támadások ellen az Angular beépítve védi a felhasználót úgy, hogy minden egyes esetben, ahol a DOM dinamikusán változhat, az Angular motor a változó részeket egy előzetes vizsgálatnak veti alá és a lehetséges veszélyeket hordozó részeket törli, vagy escape karakterek segítségével jeleníti meg. Ilyen eset például, amikor a keretrendszer két kapcsos zárójel közti Angular kifejezést értékel ki és ott mondjuk `<script>` tag közötti részeket talál, amiket aztán úgy jelenít meg, hogy a tag-eket törli, a tartalmukat azonban meghagyja.

A leírásban szerepel az is azonban, hogy ezt a védelmi mechanizmust, ha nem is ajánlott, de kikerülhetjük. Az alkalmazás fejlesztésében ezt nekem meg is kellett tennem, ugyanis a Központi oldalon elhelyezett, beépített chat ablakot egy `iframe` segítségével lehet csak megjeleníteni, ennek a forrását pedig dinamikusán kell előállítani, hiszen az éppen aktuálisan bejelentkezett felhasználó chat szobájára kell mutatnia, ami az URL része kell, hogy legyen. A biztonsági intézkedések kikerülése az alábbi kódrészletnek a segítségével lett elérve:

```
this.chatSource = this.domSanitizer.bypassSecurityTrustResourceUrl(
    `https://www.twitch.tv/${data.user.name}/chat`
);
```

Itt az történik, hogy az Angular-os domSanitizer szolgáltatás bypassSecurityTrustResourceUrl metódusának adom át azt az URL-t, ami szükséges az iframe-ben lévő chat betöltéséhez. Ez lesz az a szolgáltatás, ami egy olyan objektumot fog eredményezni, aminek a tartalmát adatkötéskor az Angular nem ellenőrzi. Ezek után a chatSource attribútumban tárolódik el a metódus hívásának az eredménye, ami pedig adatkötés segítségével a sablonban lévő iframe elem „src” attribútumára van kötve.

```
<iframe frameborder="0" scrolling="no" [src]="chatSource">
```

Mint korábban említettem az Angular beépített biztonsági szolgáltatását természetesen nem ajánlott megkerülni, de ebben az esetben én mégis megtehettem két dolog miatt. Az első, hogy a bypassSecurityTrustResourceUrl metódusnak átadott URL string nagy része, legfőképpen a domain, állandó szöveg, nem változóként van megadva, ami a javasolt megoldás erre az esetre. A második óvintézkedés, hogy az URL maradék része, ami a „data.user.name” attribútumból áll elő, a szerver oldalról érkezik, ahol is az érték egy biztonságos Twitch API hívásból, vagy az adatbázisból származik, semmiképpen sem felhasználói input-ból. Ez tehát azt jelenti, hogy mindaddig biztonságos az URL ilyen fajta előállítás, amíg a szerver oldalunk és az adatbázisunk biztosítva van.

9.1.2 Cross-Site Request Forgery (CSRF)

A másik támadási lehetőség, amire az Angular biztonsági kiadvány felhívja a figyelmet, a Cross-Site Request Forgery [46]. Ebben a támadó, a felhasználót egy nem kívánatos műveletre veszi rá egy olyan célpont weboldalon, ahol a felhasználó aktív munkamenettel rendelkezik. Ezt a támadó úgy éri el, hogy egy általa irányított oldalon olyan kódrészleteket rejt el, amik a célpont weboldal valamilyen szolgáltatását használják. Mikor a felhasználó megnyitja a támadó birtokában lévő oldalt, akkor a kérések a célpont weboldal számára a felhasználó nevében, az ő munkamenetén belül mennek el, tehát a támadó a felhasználó jogosultságaival tud műveleteket végrehajtani a célpont oldalon. A támadás elleni védekezéshez a szervernek minden esetben meg kell bizonyosodnia, hogy a hozzá érkező kérések az alkalmazáshoz tartozó weboldaltól érkeznek, nem pedig valahonnan máshonnan. Ennek egyik ajánlott módja, hogy a szerver egy token-t ad át a kliensnek egy CSRF védekezésre szolgáló sütiiben. A kliens

ezek után a sütiben tárolt adatot kiolvassa és minden további kérdésben elküldi a szervernek egy erre a célra szánt fejlécben. A szerver az újabb kérések érkezésekor ellenőrzi, hogy a fejlécben kapott adat megegyezik-e a sütiben kiküldöttel és ez alapján utasítja vissza, vagy teljesíti a kérést. Mint látható tehát, a támadás kivédéséhez nem elég kliens oldali beállításokat végezni, ehhez a szerver oldalt is fel kell készíteni. A kiadvány leírja, hogy az Angular beépítve támogatja a védelem kliens-oldali részét, a szerver oldali részről viszont a fejlesztőknek kell gondoskodnia, ami az alkalmazás esetében is megtörtént, és amit a dolgozat a 9.2.5 fejezetben fejt ki bővebben.

9.2 Express

Az Angular-on, tehát a kliens oldali technológián, kívül, a szerver oldalon használt eszköz, az Express.js keretrendszer is megfogalmazza a maga biztonsági javaslatait [47], amiket szintén felhasználtam az alkalmazás fejlesztése során.

9.2.1 Transport Layer Security (TLS)

Az Express kiadványa által tett egyik ajánlás, hogy a szerver és kliens oldal TLS titkosítási protokoll segítségével kommunikáljon egymással. A Transport Layer Security (TLS) [48] és annak elődje, a Secure Socket Layer (SSL) [49] protokollok használatukkor a hálózati réteg felett titkosítják a két oldal közti üzeneteket, így azoknak a tartalma akkor sem fog egy támadó számára elérhetővé válni, ha azokat esetleg képes lenne elfogni valamilyen közbeékelődéses támadással, vagy hasonlóval.

Az alkalmazás esetében a védekezés java már megvolt oldva bármilyen fejlesztés nélkül, a Heroku által. A Heroku ugyanis az alkalmazások számára kiosztott, alapértelmezett, „herokuapp.com” domain-en biztosítja a HTTPS elérést. Abban az esetben szükséges a felhasználóknak fizetős TLS szolgáltatást igénybe venni, ha saját domain címekkel rendelkeznek, amiket védeni akarnak, de a diplomamunka során erre nem volt szükség. Mégis akadt egy probléma a Heroku által nyújtott titkosítási szolgáltatással, mégpedig az, hogy a konfiguráció a TLS nélküli, egyszerű HTTP kéréseket nem korlátozza, ezért a szerver kódjában volt szükséges ezeknek a letiltása. Erre a célra egy új middleware réteget adtam hozzá az alkalmazáshoz, az „express-enforces-ssl”-t. Ahogy a neve is mondja, a bővítés minden kérést, ami http protokollon érkezik HTTPS helyet, egyszerűen átirányít az oldal védett megfelelőjére. Mivel helyi környezetben a titkosítás nem lett beállítva, ezért a réteg abban az esetben nincs

hozzáadva az alkalmazáshoz, ha az éppen helyi környezeten fut, aminek az ellenőrzése egy környezeti változó segítségével történik.

9.2.2 Helmet

A Helmet egy újabb ajánlott bővítése az Express keretrendszernek, amely nem más, mint 9 kisebb middleware-nek az összefoglaló csomagja. Ezekben közös, hogy mindegyik a HTTP fejléceket manipulálja valamilyen módon: vagy újabbakat ad a szerver által küldöttekhez, vagy töröl belőlük bizonyosokat. Az érdekesebb funkciók közé tartozik például az „X-Frame-Options” fejléc beállítása a válaszokba, ami megakadályozza, hogy az alkalmazást beágyazzák egy iframe-be. Egy másik érdekes és fontos hatása az „X-Powered-By: Express” fejléc törlése a szerver üzenetekből, ami implementációs részleteket fedne fel a szerver oldalról, azáltal, hogy elárulná, hogy a szerver oldal Express technológiával van megvalósítva. Utóbbi olyannyira fontos, hogy a kiadvány külön ki tér rá, hogy ha a Helmet-et nem is szeretnénk használni, legalább mi magunk kapcsoljuk ki a fejlécet.

9.2.3 Munkamenet süti

Az Express ajánlásokat fogalmaz meg a sessiont kezelő middleware réteg helyes beállításához is, vagyis, hogy az általa létrehozott süti ne rejtessen veszélyeket magukban. A weblapok által beállítható süti nem mások, mint egyszerű kulcs-érték párok, illetve a hozzájuk tartozó meta adatok, amiket a böngészők a szerver oldal kérésére tárolnak el, és amiket a kliens minden kérés fejlécében elküld a szervernek.

Az alapértelmezett beállításokkal az első gond az, hogy az Express-hez ajánlott munkamenet kezelők nem egy általános nevet adnak meg a session-t azonosító süti-nek, hanem egy Express specifikust, „connect.sid” névvel. Ezzel ugyanaz a probléma, mint az „X-Powered-By” fejléccel, vagyis, hogy a támadók minden különösebb probléma nélkül ki tudják deríteni, hogy a háttérben egy Node.js Express szerver áll és ehhez tudják igazítani a támadásaikat. Ennek elkerülése érdekében tehát egy általános nevet, a „sessionId”-t adtam meg a süti-nek.

A további beállítások közé tartozik a „cookie.secure” kapcsoló true-ra állítása, aminek az eredményeként a böngészők a süti-t csak abban az esetben küldik el a szervernek, ha HTTPS kommunikáció zajlik közöttük. Ez lényegében azt eredményezi,

hogy az alkalmazás lényegi működéséhez nélkülözhetetlen munkamenetet nem lehet létrehozni nem biztonságos kapcsolaton keresztül.

A „`cookie.httpOnly`” kapcsolóval további védelmet tudunk szerezni a Cross-Site Scripting támadásokkal szemben. A kapcsoló hatására a munkamenet kezelő bővítmény a kiküldött sütit ellátja a „`httpOnly`” direktívával, ami azt eredményezi, hogy az adott süti elérését a böngészők nem teszik lehetővé kliens oldali kód számára. Ezzel tehát akkor is védve lesz a munkamenet azonosító, ha egy támadó valamilyen módon sikeresen tudott saját kódot elhelyezni az alkalmazásunk kódjában.

Az utolsó beállítás a „`cookie.maxAge`” mező beállítása, ami egy lejárat dátumot ad a kiküldött sütinek. Ezzel azt lehet megszabni, hogy a sütit a böngésző mennyi idő leteltével törölje, amivel a süti ellopásának a kockázatát csökkenthetjük, illetve ennek eredményeként, ha a sütit mégis el sikerül lopnia egy támadónak, akkor az csak egy bizonyos ideig használható számára. Az alkalmazás esetében ezt az időt egy napra állítottam be.

9.2.4 Alkalmazás függőségek

Az Express által tett következő biztonsági ajánlás, az alkalmazás függőségeinek a karbantartása. Az npm csomagkezelő eszköz segítségével tudjuk kezelni az alkalmazásunk függőségeit, de az semmilyen megkötést nem tesz rájuk, tehát semmi nem köti meg a kezünket, hogy egy olyan régi verzióját használjuk egy könyvtárnak, ami bizonyítottan biztonsági hibákat tartalmaz. Az ilyen sérült, elavult függőségek megkeresésére és karbantartására az Express két eszközt ajánl, a Node Security Project-et (nsp) [51] és a Snyk-et [52]. Mindkettő elérhető parancssoros eszközként, működésük lényege pedig, hogy összehasonlítják az alkalmazásunk `package.json` fájlját az általuk karbantartott sérülékenységi adatbázisokkal. Az adatbázisokat az eszközök futás közben távolról érik el. Ezekben az adatbázisokban információk vannak az npm központi tárolójába feltöltött modulok minden egyes verziójáról, illetve a bennük talált esetleges biztonsági hibákról, amiket a fejlesztők folyamatosan frissítenek. A parancssoron kívül mindkét eszköz kisebb-nagyobb támogatást ad más eszközökkel való integrációra, például GitHub-bal, vagy a Heroku-val, ezzel folyamatos monitorozó szolgáltatást nyújtva az alkalmazásainkhoz.

Az első eszköz, az nsp parancssori kimenete egy egyszerű, jól átlátható táblázatos forma, amely listába szedi az összes talált hibát, illetve a hozzájuk tartozó

információkat, mint a sérülékenység neve, hogy melyik modulban található és, hogy melyik verzióban lett kijavítva. Ahogy a képen lévő második táblázatban látszik, az eszköz képes megtalálni az alkalmazásunk által közvetlenül hivatkozott függőségek hibáit, de, ahogy az első táblázat mutatja, akár egy hosszú függőség láncolaton is képes kimutatni a közvetett függőségek sérülékenységeit. A táblázat alapján végül eldönthetjük, hogy melyek azok a függőségek, amiket frissíteni szeretnénk, amit kézzel kell megtennünk az npm segítségével.

```
$ nsp check
(+) 6 vulnerabilities found
```

	Incorrect Handling of Non-Boolean Comparisons During Minification
Name	uglify-js
CVSS	8.3 (High)
Installed	2.3.6
Vulnerable	<= 2.4.23
Patched	>= 2.4.24
Path	msc-diploma@0.0.0 > @angular/cli@1.4.5 > postcss-url@5.1.2 > directory-encoder@0.7.2 > handlebars@1.3.0 > uglify-js@2.3.6
More Info	https://nodesecurity.io/advisories/39

	Regular Expression Denial of Service
Name	moment
CVSS	7.5 (High)
Installed	2.19.1
Vulnerable	<2.19.3
Patched	>=2.19.3
Path	msc-diploma@0.0.0 > moment@2.19.1
More Info	https://nodesecurity.io/advisories/532

	Regular Expression Denial of Service
--	--------------------------------------

27. ábra A Node Security Project kimenete

A második eszköz, a Snyk esetében lehetőséget adnak egy interaktívabb megoldás kipróbálására, amiben egy varázslót kell futtatnunk a függőségek ellenőrzéséhez. Itt a varázsló nem egyszerre írja ki az összes talált hibát, hanem sorra veszi őket és mindegyikhez felajánl bizonyos műveleteket, mint a sérülékeny függőség

frissítése, vagy a sérülékenység figyelmen kívül hagyása. Azokban az esetekben, ahol azt választjuk, hogy a függőséget frissíteni szeretnénk, az eszköz maga fogja az npm segítségével elvégezni a szükséges módosításokat a projektünkben.

```
$ snyk wizard
Snyk's wizard will:
  * Enumerate your local dependencies and query Snyk's servers for vulnerabilities
  * Guide you through fixing found vulnerabilities
  * Create a .snyk policy file to guide snyk commands such as `test` and `protect`
  * Remember your dependencies to alert you when new vulnerabilities are disclosed

Querying vulnerabilities database...
Tested 955 dependencies for known vulnerabilities, found 6 vulnerabilities, 9 vulnerable paths.

? Low severity vuln found in moment@2.19.1, introduced via moment@2.19.1
- desc: Regular Expression Denial of Service (ReDoS)
- info: https://snyk.io/vuln/npm:moment:20170905
  Remediation options Upgrade to moment@2.19.3

? Low severity vuln found in moment@2.19.1, introduced via msc-diploma-bot@1.0.0
- desc: Regular Expression Denial of Service (ReDoS)
- info: https://snyk.io/vuln/npm:moment:20170905
- from: msc-diploma-bot@1.0.0 > moment@2.19.1
  Remediation options Skip

? 2 vulnerabilities introduced via pm2@2.7.2
- info: https://snyk.io/package/npm/pm2/2.7.2
  Remediation options
  Re-install pm2@2.7.2 (triggers upgrade to moment@2.19.3)
  Review issues separately
> Set to ignore for 30 days (updates policy)
  Skip
```

28. ábra Snyk eszköz kimenete

Érdekességgként ez a sebezhetőség típus nem csak az Express ajánlása szerint fontos. Az alkalmazás fejlesztése közben maga a GitHub is küldött egy e-mailt számomra, illetve a repository oldalán is kitett egy bejegyzést, mikor kiderült, hogy az alkalmazásban egy olyan modulra van függőség, ami potenciális veszélyeket hordoz. Az látszik, hogy a GitHub csak egy ilyen hibát talált, míg az előző két eszköz többet is, tehát annyira komoly védelemről itt nem beszélhetünk, de mindenképpen érdemes megemlíteni, hogy ez a védekezési forma a GitHub szerint is fontos.

9.2.5 Cross-Site Request Forgery (CSRF)

Ahogy az Angular biztonsági fejezetében szereplő CSRF fejezetben 9.1.2 említettem, ennek a támadásnak a kivédéshez a kliens és a szerver oldalon is szükséges módosítások elvégzése. Az Express is megemlíti, hogy érdemes ezek ellen a támadások ellen védekezni, mégpedig a csrf [53] Express bővítmény segítségével. A bővítmény

hozzáadása után a middleware rétegeink által megkapott „request” paraméterhez ad hozzá egy új metódust, „csrfToken” névvel. A metódus meghívásakor visszaad egy token-t, amit viszont a fejlesztő dolga eljuttatni a kliens oldalra, annak függvényében, hogy ott milyen technológia van. Jelen esetben a kliens oldali Angular-hoz igazodva egy új, saját middleware réteget adtam az alkalmazáshoz, ami a token-t egy „XSRF-TOKEN” nevű süti-be teszi bele minden hívás válaszában. Az Angular HttpClient szolgáltatása, ami a szerver oldalra való kommunikációra szolgál, alapértelmezetten keresi az előbb említett nevű sütit, és ha megtalálja, akkor ezentúl minden kérésbe beleteszi a benne tárolt token-t „X-XSRF-TOKEN” néven. A csrf bővítmény alapértelmezetten több helyen is keresi a kérésekben a token-t, ezek közül az egyik az „X-XSRF-TOKEN” fejléc, tehát a szerver oldalon további módosításokra nem volt szükség.

9.3 Injection

Az Angular és az Express ajánlotta biztonsági intézkedéseken kívül az injection támadások elleni védekezésnek kiemelt figyelmet szenteltem. Ez a támadás, a weboldalak sebezhetőségéről szóló OWASP TOP 10 kiadvány 2013-as [54] és 2017-es [55] verziójának is az első helyén áll, tehát egy nagyon elterjedt, komoly veszélyforrásról van szó. Az injection támadások dióhéjban arról szólnak, hogy a támadó valamilyen nem ellenőrzött adat küld a célpont egy célpont interpreternek egy parancs, vagy lekérdezés részeként. A támadó az általa küldött adat segítségével rá tudja venni az interpretert olyan parancsok végrehajtására, vagy olyan adatok lekérdezésére, amikhez egyébként nem lenne joga. Jelen dolgozat szempontjából ez a támadási forma azért is érdekes, mert az injection támadások egyik előszeretett célpontja az SQL adatbázisok voltak, a jelenlegi alkalmazás viszont, jóval később megjelent NoSQL adatbázisokat használ a működéséhez. Ez azt jelenti, hogy az injection új módjait kell a támadóknak felfedezniük, és ugyanúgy, a fejlesztők arra kényszerülnek, hogy új módokon védekezzenek a támadások ellen. Mivel a NoSQL adatbázisok az utóbbi néhány évben kezdtek el igazán elterjedni (a MongoDB-ben és a Redis-nek is 2009-ben lett kiadva az első verziója) a témában még viszonylag kevés értekezés született, illetve a fellelhető források nem is mindig értenek egymással teljesen egyet a megfelelő védekezés módjában.

9.3.1 MongoDB

Az alkalmazás által használt, sérülékenyebb adatbázis a MongoDB, amelynek támadási lehetőségeiről több iromány és blog bejegyzés is született [56][57][58][59]. A cikkekből két támadási lehetőség körvonalazódik ki, ami MongoDB adatbázis esetén jellemző lehet.

Az első, a MongoDB API specifikus „\$where” operátor használata az alkalmazásunkban, amit az úgynevezett „query filter” dokumentumokban lehet használni. A query filterekkel azt tudjuk meghatározni, hogy egy művelet mely dokumentumokon hajtsdjon végre, ha az képes egyszerre többen is dolgozni, amilyen például az olvasás. Ezek a dokumentumok JavaScript objektumok, amikben a kulcsok azok az attribútumok, amiken feltételeket akarunk megfogalmazni, az értékek pedig a maguk a feltételek, amiket akár a különböző operátorok segítségével is megadhatunk. A query filter objektumok struktúrája:

```
{
  <mező1>: <érték1>,
  <mező2>: { <operátor>: <érték> },
  ...
}
```

A \$where operátor használatával általánosan tudjuk megfogalmazni a lekérésünk feltételeit, hasonlóan a hagyományos SQL where feltételekhez. A probléma ezzel az, hogy a \$where operátornak bármilyen JavaScript kódot értékül lehet adni, így, ha megengedjük, hogy ide felhasználói input kerüljön, akkor egy támadó jogosulatlan adatbázis műveleteket hajthat végre, vagy akár Denial of Service támadást véghez vihet egy végtelen ciklus segítségével. A legegyszerűbb megoldás a támadás ellen az, ha egyáltalán nem használjuk a \$where operátort, vagy csak olyan esetekben, amikor teljesen biztosak lehetünk benne, hogy nem felhasználói input-ból származik az operátornak átadott kifejezés. A \$where ellen szól egyébként még az is, hogy használatakor az adatbázis motor nem tud indexeket használni, így jóval lassabbak lehetnek az így megfogalmazott lekérdezések. A fejlesztett alkalmazásban ennek tükrében sehol nem használtam az operátort, így biztosítva a támadás elleni védelmet.

A második jellemző támadás a query filterek azon használati esetét célozza, amiben egyszerű mező: érték formában akarunk szűréseket végezni, például:

```
const query = {
  username: req.body.username,
  password: req.body.password
};

db.collection('users').findOne(query, function (err, user) {
  console.log(user);
});
```

A fenti példában a felhasználótól kapott név-jelszó páros alapján szeretnénk az adatbázisból egy dokumentumot megtalálni a „user” kollekcióból. A fejlesztői elgondolás itt természetesen az, hogy a kapott értékek string típusúak lesznek, de egy támadót semmi sem gátol meg abban, hogy string helyett objektumokat küldjön a két mezőben, amit, ahogy korábban láthattuk, a MongoDB is elfogad. Ilyen objektum lehet például a „{ \"\$ne\": null}”, ami valószínűleg minden felhasználó esetében igaznak fog kiértékelődni. Megfelelő objektumok megadásával a támadó tehát ráveheti a szervert, hogy olyan műveleteket hajtson végre, amihez nincs joga. Ez ellen a támadás ellen az egyik javasolt védekezési mód a „mongo-sanitize” npm modul használata. A modul egy egyszerű függvényből áll, ami a paraméterben kapott objektumokból kitörli azokat a mezőket, amik dollár jellel kezdődnek, ami a MongoDB operátorok jellemzője. Az alkalmazásomat tehát úgy alakítottam át, hogy, a sanitize függvény segítségével, minden felhasználói inputot átalakítok biztonságos formára az adatbázis műveletek előtt.

A fenti esetekből látszik, hogy az adatbázis eszközök nem adnak a kezünkbe semmit, amivel védekezni tudunk a támadások ellen, mint például a hagyományos SQL esetén a parametrizált lekérdezések. Ezért különösen fontos a tanulmányok azon állítása, hogy az injection támadások ellen a legjobb védekezés a kódbázis manuális ellenőrzése, lehetőleg több fejlesztő által.

9.3.2 Redis

A Redis, kulcs-érték tároló révén lényegesen egyszerűbb műveletekkel és API-val rendelkezik, mint a MongoDB. Injection jellegű támadása a Redis fejlesztőinek állítása szerint nem lehetséges az adatbázis motor használatával annak felépítése miatt. Redis esetében támadási lehetőséget elméletileg egyedül az arra csatlakozó kliensekben használt API-k jelenthetnek. A tipikus támadási lehetőség itt az olyan metódusok kihasználása, amik elfogadnak paraméterként egy elemet, vagy egy tömböt is. Ha a metódus azzal a szándékkal van használva, hogy egy darab értéket tároljanak vele,

akkor egy támadó a kérések megfelelő módon való összeállításával ráveheti a szerveret, hogy az egy érték helyett egy egész tömböt tároljon az adatbázisban. A tömb tartalmazhat a helyes működést előidéző adat mellett támadó jellegű adatokat is, például XSS kódot [60]. Az ilyen esetek ellen a például úgy védekezhetünk, hogy ellenőrizzük a kapott adatok valóban string típusúak-e, például a JavaScript beépített „typeof” operátorával.

10 Összefoglalás

A munkám során tehát megismerhettem több olyan eszközt és technológiát, amelyek, annak ellenére, hogy az utóbbi évtizedben jelentek meg, hatalmas jelentőséggel bírnak a modern szoftverfejlesztés világában. A megszerzett tudás segítségével sikeresen megalkottam egy webes felülettel rendelkező, komplex rendszert. Ennek a rendszernek mondhatni minden egyes rétege, a megjelenítéstől az adattárolásig ezeknek a modern eszközöknek a segítségével lett kialakítva. Elmondható továbbá, hogy a megalkotott programot körülvevő projektstruktúra úgy lett kialakítva, hogy gyorsan és könnyedén be lehessen vonni a fejlesztésbe újabb embereket az esetleges további fejlesztésekhez. Az eddigieken kívül még az alkalmazásról bebizonyosodott, hogy könnyedén telepíthető újabb környezetekbe a megfelelő beállítások elvégzése után. Megismertem és alkalmaztam továbbá több tesztelési módszert, illetve mélyebb ismeretekre tudtam szert tenni a webes biztonság terén az alkalmazás biztonságosabbá tétele során.

A jövőbeli fejlesztések során az alkalmazásba elsőként újabb funkciókat lehet beépíteni. A chat botok rengeteg lehetőséget rejtenek magukban, mint például üzenetek automatikus szűrése, statisztikák gyűjtése, stb. Az alkalmazás biztonsági óvintézkedéseit is tovább lehetne bővíteni, például az OWASP TOP 10 lista segítségével, hiszen egy webes alkalmazás biztonsága sosem lehet tökéletes, azt folyamatosan fejlesztenünk kell. Végül, mivel az alkalmazás kliens és szerver oldala közt főként REST segítségével történik a kommunikáció, ezek lazán csatolt mivolta miatt könnyedén ki lehetne alakítani a webes kliens mellé mobil klienseket is, akár több platformra.

Összességében elmondható tehát, hogy rengeteg új tudással gazdagodtam a megismert eszközök, technológiák révén, illetve némi gyakorlatra is sikerült szert tennem ezekkel az alkalmazás fejlesztése közben.

Irodalomjegyzék

- [1] Twitch, <https://www.twitch.tv> (2017 december)
- [2] Internet Engineering Task Force: *Request for Comments, Internet Relay Chat Protocol*, <https://tools.ietf.org/html/rfc1459> (2017 május)
- [3] Typescript, <https://www.typescriptlang.org> (2017 május)
- [4] Mozilla Developer Network: *JavaScript*, <https://developer.mozilla.org/bm/docs/Web/JavaScript> (2017 december)
- [5] Mozilla Developer Network: *EcmaScript*, https://developer.mozilla.org/en-US/docs/Web/JavaScript/Language_Resources (2017 december)
- [6] Microsoft Visual Studio, <https://www.visualstudio.com/vs> (2017 december)
- [7] WebStorm, <https://www.jetbrains.com/webstorm> (2017 december)
- [8] Eclipse, <https://www.eclipse.org> (2017 december)
- [9] Visual Studio Code, <https://code.visualstudio.com> (2017 december)
- [10] git, <https://git-scm.com> (2017 december)
- [11] Electron, <https://electronjs.org> (2017 december)
- [12] Angular, <https://angular.io> (2017 május)
- [13] Material Design, <https://material.io> (2017 november)
- [14] GitHub: *Angular Material 2*, <https://github.com/angular/material2> (2017 november)
- [15] World Wide Web Consortium: *CSS Flexible Box Layout Module*, <https://www.w3.org/TR/css-flexbox-1> (2017 november)
- [16] GitHub: *Angular Flex Layout*, <https://github.com/angular/flex-layout> (2017 november)
- [17] Angular CLI, <https://cli.angular.io> (2017 november)
- [18] Node.js, <https://nodejs.org> (2017 november)
- [19] Less, <http://lesscss.org> (2017 december)
- [20] Sass, <http://sass-lang.com> (2017 december)
- [21] Karma, <https://karma-runner.github.io/1.0/index.html> (2017 december)
- [22] Protractor, <http://www.protractortest.org/#/> (2017 december)

- [23] *Webpack*, <https://webpack.github.io> (2017 november)
- [24] *npm*, <https://www.npmjs.com> (2017 november)
- [25] *Maven*, <https://maven.apache.org> (2017 december)
- [26] *Express*, <https://expressjs.com> (2017 november)
- [27] *MongoDB*, <https://www.mongodb.com> (2017 november)
- [28] *Mongoose*, <http://mongoosejs.com> (2017 november)
- [29] *Redis*, <https://redis.io> (2017 november)
- [30] Internet Engineering Task Force: *Request for Comments, The OAuth 2.0 Authorization Framework*, <https://tools.ietf.org/html/rfc6749> (2017 május)
- [31] *GitHub*, <https://github.com> (2017 november)
- [32] *Travis CI*, <https://travis-ci.org> (2017 november)
- [33] *Heroku*, <https://www.heroku.com> (2017 november)
- [34] Angular API: *Resolve*, <https://angular.io/api/router/Resolve> (2017 november)
- [35] Angular API: *CanActivate*, <https://angular.io/api/router/CanActivate> (2017 november)
- [36] GitHub: *Winston*, <https://github.com/winstonjs/winston> (2017 november)
- [37] Internet Engineering Task Force: *Request for Comments, The WebSocket Protocol*, <https://tools.ietf.org/html/rfc6455> (2017 november)
- [38] *Socket.io*, <https://socket.io/> (2017 november)
- [39] Wikipedia: *ISO 8601*, https://en.wikipedia.org/wiki/ISO_8601 (2017 november)
- [40] Angular API: *DatePipe*, <https://angular.io/api/common/DatePipe> (2017 november)
- [41] GitHub: *ngx-charts*, <https://github.com/swimlane/ngx-charts> (2017 november)
- [42] Angular: *Testing*, <https://angular.io/guide/testing> (2017 december)
- [43] *WebDriver*, <http://www.seleniumhq.org/projects/webdriver> (2017 december)
- [44] Angular: *Security*, <https://angular.io/guide/security> (2017 december)
- [45] Open Web Application Security Project: *Cross-Site Scripting*, [https://www.owasp.org/index.php/Cross-site_Scripting_\(XSS\)](https://www.owasp.org/index.php/Cross-site_Scripting_(XSS)) (2017 december)

- [46] Open Web Application Security Project: *Cross-Site Request Forgery*, [https://www.owasp.org/index.php/Cross-Site_Request_Forgery_\(CSRF\)](https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF)) (2017 december)
- [47] Express: *Security Best Practices*, <https://expressjs.com/en/advanced/best-practice-security.html> (2017 december)
- [48] Wikipedia: *Transport Layer Security*, https://en.wikipedia.org/wiki/Transport_Layer_Security (2017 december)
- [49] Wikipedia: *Secure Sockets Layer*, https://en.wikipedia.org/wiki/Transport_Layer_Security#SSL_1.0.2C_2.0_and_3.0 (2017 december)
- [50] GitHub: *Helmet*, <https://github.com/helmetjs/helmet> (2017 december)
- [51] *Node Security Project*, <https://nodesecurity.io> (2017 december)
- [52] *Snyk*, <https://snyk.io> (2017 december)
- [53] GitHub: *csurf*, <https://github.com/expressjs/csurf> (2017 december)
- [54] Open Web Application Security Project: *OWASP TOP 10 - 2013*, https://www.owasp.org/images/f/f8/OWASP_Top_10_-_2013.pdf (2017 december)
- [55] Open Web Application Security Project: *OWASP TOP 10 - 2017*, https://www.owasp.org/images/7/72/OWASP_Top_10-2017_%28en%29.pdf.pdf (2017 december)
- [56] Open Web Application Security Project: *Testing for NoSQL injection*, https://www.owasp.org/index.php/Testing_for_NoSQL_injection (2017 december)
- [57] WebSecurify blog: *Hacking NodeJS and MongoDB*, <https://blog.websecurify.com/2014/08/hacking-nodejs-and-mongodb.html> (2017 december)
- [58] WebSecurify blog: *Attacking NodeJS and MongoDB – Part Two*, <https://blog.websecurify.com/2014/08/attacks-nodejs-and-mongodb-part-to.html> (2017 december)
- [59] Zanon blog: *NoSQL Injection in MongoDB*, <https://zanon.io/posts/nosql-injection-in-mongodb> (2017 december)
- [60] Patrick Spiegel, Medium blog: *NoSQL Injection > Redis*, <https://medium.com/@PatrickSpiegel/https-medium-com-patrickspiegel-nosql-injection-redis-25b332d09e58> (2017 december)

Függelék