

SZAKDOLGOZAT

Görög Balázs

Debrecen

2022

Debreceni Egyetem
Informatikai Kar
Adattudomány és Vizualizáció Tanszék
E-Sport bajnokság-szervező weboldal Java alapokon

Témavezető: Tiba Attila
Beosztása: Tanársegéd

Készítette: Görög Balázs
Szak Megnevezés:
Programtervező Informatikus BSc

Debrecen
2022

Tartalomjegyzék

1.Bevezetés	5
1.1 Bevezetés a PostgreSQL technológiához	6
1.2 Bevezetés a Spring technológiához	6
1.3 Bevezetés a ReactJS technológiához	6
2. Architektúrális áttekintés	7
2.1 REST Áttekintés	7
2.2 Relációs adatbázis áttekintés	7
2.4 React áttekintés	8
3. Az adatbázis	9
3.1 Első normálforma	9
3.2 Második Normálforma	10
3.3 Harmadik Normálforma	11
3.4 Boyce-Codd Normálforma	11
3.5 Az adatbázis elérése	12
3.5.1 Az entitások	13
3.5.2 A repository-k	15
3.5.3 Csatlakozás az adatbázishoz	16
3.6 Az adatbázis elérhetőségének gyorsítása	17
4. A Logikai réteg	19
4.1 Az osztályok elrendezése	19
4.2 Service réteg	21
4.3 A Controller réteg	23
4.3.1 Annotációk	24
4.3.2 A design	28

4.4 Aszinkron adatfeldolgozás	30
4.4.1 A külön szál	31
4.5 Aszinkron értesítések.....	36
4.6 A Logolás	39
4.7 A biztonság kialakítása.....	40
4.8 A backend előkészítése a frontendre	42
5. A kliensoldali réteg.....	44
6. Az alkalmazás működés közben.....	47
6.1 Próba-bajnokság	51
7. Összefoglalás	54
8. Irodalomjegyzék	56
9. Függelék	58
10. Köszönetnyilvánítás	58

1.Bevezetés

Napjainkban egyre jobban előtérbe kerülnek a különböző webes alkalmazások, erre több ok van, mint például a nagy mennyiségben elérhető webfejlesztő programozók, de a fő okok inkább a technikaiak. Ilyen technikai oknak fel lehet sorolni például azt, hogy a webes alkalmazások, oldalak esetén nagyon egyszerű eljuttatni a végfelhasználóhoz a termék új változatait, vagy még azt, hogy HTML, CSS, és JavaScript szabványok jól dokumentáltak, egyszerűen használhatóak, így egy felhasználói felület kialakítása egyszerű, gyors, és hatékony lehet.

A projekt célja egy olyan API és ahhoz tartozó egyszerű frontend kialakítása volt, ami demonstrálja, hogy a Java nyelv viszonylagos kora és eredeti tervezési irányzata ellenére [1] is képes modern, reszponzív alkalmazások megvalósítására. Másik célja természetesen az, hogy egy egyszerűen használható, könnyű súlyú felületet hozzon létre, amelyen keresztül az egyik legnépszerűbb e-sport kisebb bajnokságait le lehet vezetni, a győzteseket, veszteseket meg lehet egyértelműen állapítani és a verseny kezdőpozícióit könnyen ki lehet alakítani, osztani.

A fő oka a projekt elkészítésének az volt, hogy a létező megoldások nem eléggé alkalmasak, a kisebb versenyek kialakítására, sőt, időnként még az adatkezelési szokásaik is hagynak kíváncsiságot maguk után.

A projekt szerkezete a következő: Az állapotosságot, az adatok perzisztenciáját egy PostgreSQL adatbázis valósítja meg, aminek eléréséről, módosításáról a Spring Data JPA gondoskodik, amit a web felé a Spring kontrollerei nyitnak meg, amely adatot egy ReactJS SPA jelenít meg a felhasználó felé.

A program biztonságosságának elérése, a bejelentkezés megvalósítása a Spring Security-n keresztül valósult meg, ami egy modern, biztonságos bejelentkezési procedúrát tesz lehetővé.

További fontos szempont volt még a GDPR szabályozások betartása, aminek fontos része az, hogy a felhasználó könnyen, egyszerűen, és gyorsan el tudja feledtetni a személyes adatait, ami jelen esetben teljesen automatizált, így ténylegesen olyan gyors és egyszerű amennyire lehetséges.

1.1 Bevezetés a PostgreSQL technológiához

A PostgreSQL egy ingyenes, nyílt forráskódú, objektum-relációs adatbázis kezelő rendszer, amely az utóbbi években nagy népszerűségnek örvend. A modern relációs adatbázis kezelő rendszerekhez hasonlóan támogatja az ACID tranzakciókat, de több nem alapvető tulajdonsággal is rendelkezik, mint például a tömb adattípus, vagy az öröklődés [2] támogatása, ami alapról a tisztán objektum orientált adatbázisok része lenne. Ezen előnyök ismeretében nem nehéz felismerni, hogy az olyan hatalmas vállalatok, mint például a Microsoft [3] miért kezdték el hatalmas, petabyte méretű adatbázisok kezelésére használni.

1.2 Bevezetés a Spring technológiához

A Spring Keretrendszer egy nyílt forráskódú alkalmazás keretrendszer a Java nyelvhez. Főbb tulajdonsága az az Inversion of Control magasszintű támogatása, a nem blokkoló architektúra és a Convention over Configuration elve.

A projektben kihasznált főbb előnyei nem mások voltak, mint: az adatbázis csatlakozók, jelen esetben a JDBC, és annak a kiterjesztése, a Spring Data JPA, a Modell-Nézet-Vezérlő architektúra támogatása, a Spring Security nyújtotta biztonsági előnyök, mint például az API levédésének egyszerű implementációja és legfontosabban a Spring Boot keretrendszer-kiterjesztés, ami segített nagy mennyiségű, triviális konfigurációs kód megírásának elkerülésében.

1.3 Bevezetés a ReactJS technológiához

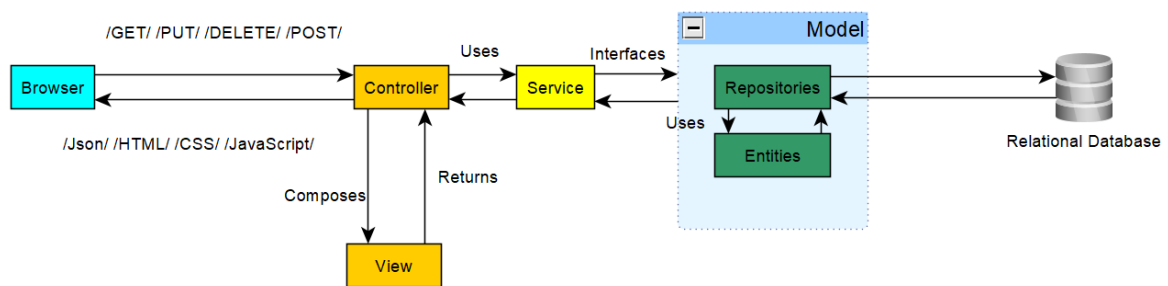
A ReactJS (rövidebben: React) egy, a Facebook által kezdeményezett nyílt-forráskódú front-end JavaScript/ECMAScript könyvtár, aminek a fő célja nem más, mint az újrahasznosítható felhasználói felület komponensek létrehozatala deklaratív módon.

A React eredetileg csak az objektum-orientált, osztály-alapú komponenseket támogatta, de a React 16.8.0-ás verziója óta támogatja a funkcionális komponenseket az úgynevezett hook-okon keresztül, amelyek segítségével minden olyan funkcionalitást el lehet függvényekből érni, amiket eddig csak az osztályokban lehetett.

Projekt front-end oldala is funkcionális komponenseket használ, mert úgy tapasztaltuk, hogy ezekkel sokkal könnyebb dolgozni, egyszerűbb használni és kevesebb hibát eredményeznek. Ezeknek a tulajdonságoknak köszönhetően a front-end gyorsabban készült el, mint amire eredetileg számítottunk, amivel az egész projekt felgyorsult.

2. Architektúrális áttekintés

A projekt felépítése alapjaiban az MVC (Model-View-Controller) architektúrán alapszik, amit a Spring Keretrendszer valósít meg. A Model-t a Spring Repository-k és Entity-k valósítják meg, a relációs adatbázissal való kommunikáció útján. A Model-t a különböző Service-ek és a Service-eken keresztül a Controller-ek dolgozzák fel. A Controller-ek feladata a kliensekkel való kommunikálás a HTTP protokollon keresztül, aminek különböző View-okat szolgáltatnak ki, mint például JSON adat vagy JavaScript fájlok.



1. ábra: A projekt felépítése

2.1 REST Áttekintés

A REST rövidítés jelentése nem más, mint Representational State Transfer, ami mára egy nagyon népszerű szabvánnyá vált. Főbb előnyei az állapotmentesség, az egyszerű, kliens-szerver architektúra és ez egyszerű, szabványos adatrepresentáció [4].

Jelen projektben azért esett a REST-re a választás, mivel ez kiváló olyan adatok átvitelére, amit JSON vagy hasonló formátumban jól lehet ábrázolni, mint például egy relációs adatbázis táblájának adatai.

Az állapotmentességet ebben az esetben a HTTP protokollon keresztül valósítottuk meg, mivel ez a protokoll alapvetően nem tárol állapotot [5].

2.2 Relációs adatbázis áttekintés

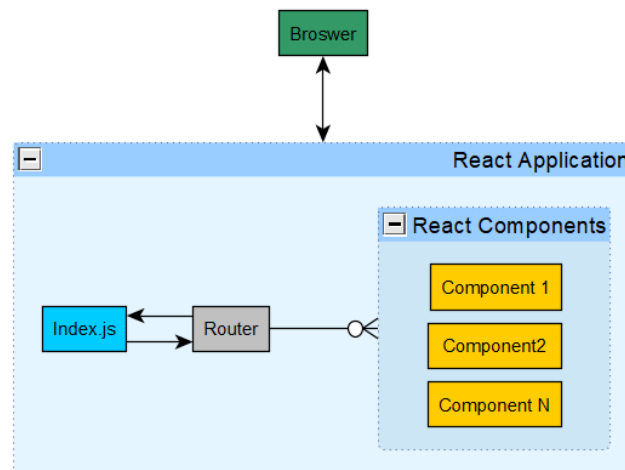
A relációs adatbázisok, mint ahogy a nevük sejteti, az adatok relációs modelljén alapszanak. Táblákban és az azokban lévő adatok közti kapcsolatok alapján tárolják a szükséges információt.

Jelen esetben azért erre az adatbázis fajtára esett a választás, mert ez általánosan a legjobb, abban az esetben, mikor a fejlesztő előre tudja a tárolni kívánt adatok fajtáját, mennyiségét, az azok közti kapcsolatot és az elrendezését. Ebben a projektben ezek a feltételek teljesülnek, így egyértelmű választás volt egy relációs adatbázis.

2.4 React áttekintés

Manapság egyre ritkább, hogy alap JavaScriptben (vagy másik nevén EcmaScript) fejlesszünk weboldalakat, ennek több oka van. Ennél a frontend-nél az ok egyszerű volt, mégpedig az, hogy React-tal könnyű Single Page Application-öket létrehozni, ami a szerveroldali részt jelentősen megkönnyíti, de további előnye volt még a könnyű állapotkezelés is.

A React-hoz tartozó, React Router nevű könyvtár segítségével oldottunk meg a megfelelő állapotú dokumentum megjelenítését. Használata és a működése, egyszerű: A React alkalmazás mindig csak azt a React komponenst jeleníti meg, aminek a hozzá tartozó, meghatározott útja, illik az éppen aktuális URL-re, így minimalizálni lehet a szükséges HTTP végpontok kialakítását. A kiszolgáló szerver szempontjából a React alkalmazás egy statikus erőforrás, ami a kiszolgálást is jelentősen egyszerűvé tette.



2. ábra: A React Router működése

3. Az adatbázis

A relációs adatbázis, ami jelen implementációban a PostgreSQL alatt valósult meg, talán az egyik legfontosabb része a projektnek, hisz, ha annak a kialakítása nem optimális, rossz, akkor minden más rész működését és fejlesztését jelentősen nehezíti, ebből következően egy jól tervezett, kialakított relációs adatbázis-séma nagyban megkönnyítheti a fejlesztés későbbi lépéseit.

Az adatbázis létrehozásakor az első kérdés, amivel foglalkoztunk, az az volt, hogy használjuk-e a PostgreSQL által biztosított tömb adattípust. Több előny és hátrány megfontolása után egy döntő információra leltünk, miszerint a PostgreSQL nem támogatja jelenleg a külső kulcsok tömbként való tárolását, ezzel az ACID elvekből a C-t, a konzisztenciát nem képes biztosítani. Mivel, ha nem külső kulcsként vannak eltárolva a tömb elemei, így, ha esetleg törlésre kerül az a sor, amire a kulcs utal, akkor a kulcs nem törlődik tovább és egy nem létező elemre utal, ami könnyen nagyon nagy problémává tud válni.

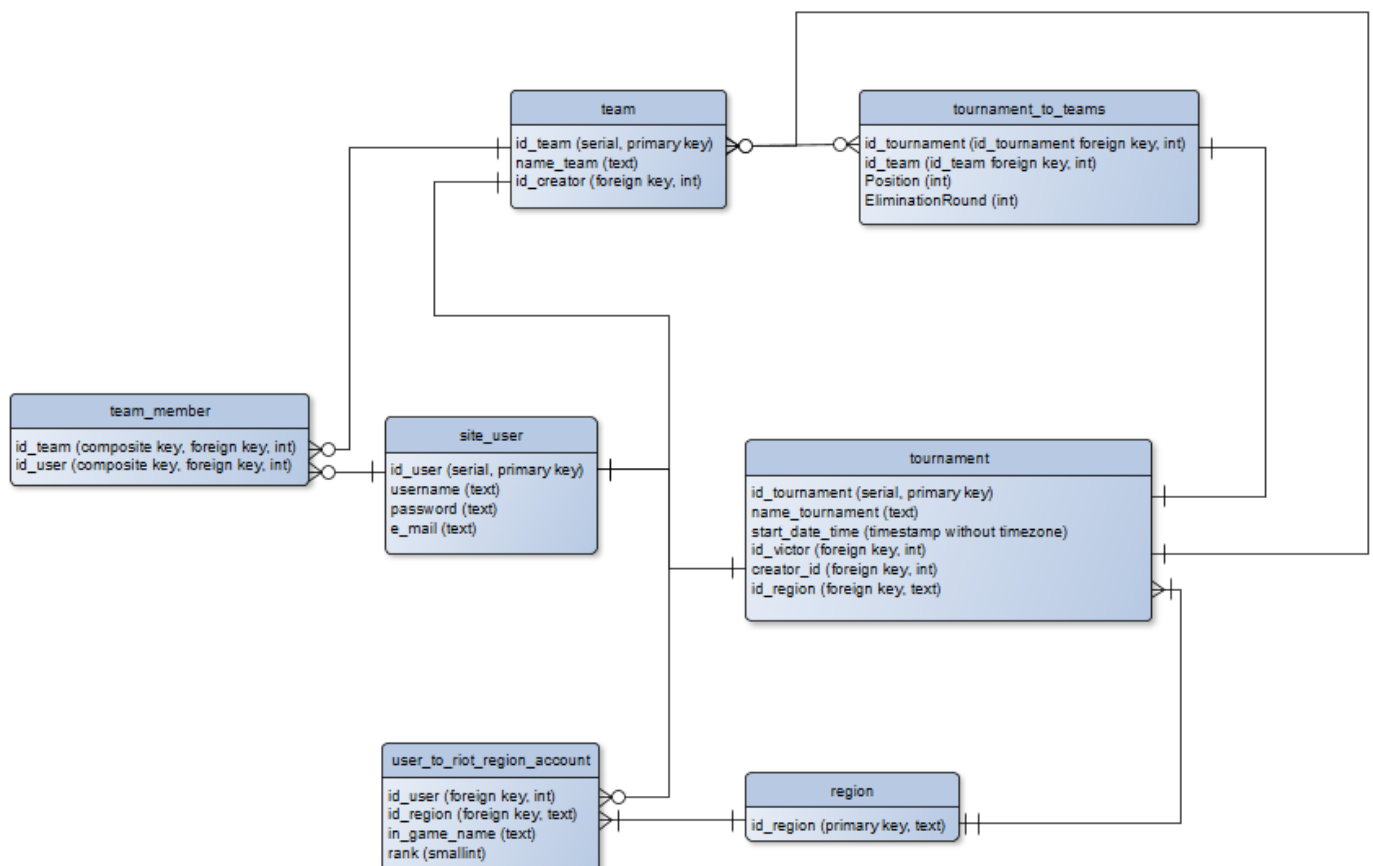
Az adatbázis-séma tervezésekor szempont volt, hogy annyi normálformát teljesítsen az adatbázis, amit még ajánlott, emberileg könnyen olvasható és a számítási nehézsége se túl magas. Ebből az okból, az adatbázis tervezésekor a megcélzott normálforma a Boyce-Codd Normálforma volt. Ennek több oka is volt, többek között az is, hogy a tanulmányaink és a szakmai gyakorlatunk alatt tapasztaltak szerint az iparban gyakran nem terveznek több normálformára adatbázisokat. Ennek egyik legeggyértelműbb oka az emberi olvashatóság szükséglete, valamint az iparban feldolgozandó hatalmas mennyiségű adatnál nagyon fontos a számítási bonyolultság.

3.1 Első normálforma

Legelőször az első normálforma kielégítéséről kellett gondoskodni, ami szerencsére eleve teljesül, hisz az első normálforma definíciója a következő:

- Egy reláció akkor és csak akkor van első normálformában, ha minden attribútuma egyszerű, nem összetett adat.

Láthatjuk, hogy ez már pusztán azért teljesül, mert nem került kihasználásra a PostgreSQL által biztosított tömb adattípust, így minden attribútum egyszerű, elemi adatokat tartalmaz.



3. ábra: Az adatbázis-séma

3.2 Második Normálforma

Egy reláció akkor van második normálformában, ha:

- Első normálformában van.
- Minden nem elsődleges kulcs attribútuma teljesen funkcionálisan függ az elsődleges kulcstól.

Ennek a jelentése már közel sem annyira egyértelmű, mint az első normálforma esetében volt, de egy példával ez is könnyedén bemutatatható. Az előző részben röviden be lett bizonyítva, hogy az első normálforma követelményei teljesülnek, ezért azt ebben a részben már nem szükséges tárgyalni.

Jelen esetben, figyeljük meg a 3. ábrán a „team” nevezetű táblát. Ez a tábla, a második normálformát abban a példa esetben sértené, ha az „id_creator” attribútuma mellett a tábla része lenne még egy „name_creator” attribútum. Mivel adott felhasználót az ID egyértelműen

azonosít, így beláthatjuk, hogy adott „id_creator” attribútumhoz mindig ugyanolyan értékű „name_creator” attribútum tartozna.

Láthatjuk, hogy a fenti sémában nincs olyan attribútum, ami sértené a fenti definíciót, így tudható, hogy a reláció nem sérti a második normálformát.

3.3 Harmadik Normálforma

Egy reláció akkor és csak akkor van harmadik normálformában, ha:

- Második normálformában van.
- Minden nem elsődleges attribútuma nem tranzitív módon függ a minden kulcstól.

Bizonyos értelmezésben az adatbázis sémánk sérti ezt a normálformát, hisz a „user_to_riot_region_account” táblán belül ott vannak az „in_game_name” és „rank” attribútumok, amik között bizonyos értelmezésben függés áll fenn, így sértve a harmadik normálformát. Jelen értelmezésében az adatnak, az „in_game_name” mindössze a játészó felek beazonosításának megkönnyítését szolgálja, hisz a „rank” attribútum a felhasználó adott régióra vonatkozó képességeit szolgáltatott kifejezni, hisz a régiók között ismertek vannak különbségek.

Mivel itt erősen határeset az, hogy sérti-e a séma normálformát, az adatbázis számítási komplexitását jelentősen növelné az átdolgozása. Továbbá a fellépő problémák esetleges konzisztencia hibák esetén minimálisak lennének és még a felhasználó nem is képes megváltoztatni az „in_game_name” attribútumot függetlenül a „rank” attribútumtól, ezért úgy döntöttünk, hogy így hagyjuk az adatbázis kialakítását.

3.4 Boyce-Codd Normálforma

A Boyce-Codd normálforma a harmadik normálforma egyfajta kiterjesztése, néha a három és feledik normálformaként is utalnak rá, ami a definícióján is érződik. Egy reláció akkor és csak abban az esetben van Boyce-Codd normálformában, ha:

- Harmadik normálformában van.
- Ha bármely nem triviális $L \rightarrow B$ függés esetén L a superkulcs. (Az $L \rightarrow B$ jelentése az, hogy B funkcionálisan függ L -től)

Láthatjuk, hogy ez definíció nagyon hasonló a harmadik normálformához és hogy a harmadik normálformához leírtakugyan úgy alkalmazhatóak itt is, hisz minden más tábla esetén teljesül a BCNF normálforma, kivéve „user_to_riot_region_account” tábla bizonyos értelmezéseiben.

3.5 Az adatbázis elérése

Az adatbázis elérését alapvetően a JDBC driver (Java Database Connectivity driver) teszi lehetővé. A JDBC fő feladatai a következők [6]:

- Az adatforrással (jelen esetben egy PostgreSQL adatbázissal) való kapcsolat teremtése.
- Lekérdezések és frissítési utasítások elküldése az adatforrás.
- A lekérdezések eredmények kezelése.

Láthatjuk, hogy ezek mind felettebb hasznos dolgok, de talán a leghasznosabb része az nem más, mint az úgynevezett PreparedStatement-ek. Ezek olyasfajta előre fordított SQL utasítások, amelyek helyes használatával könnyen elkerülhetők az SQL befecskendezési támadások jelentős része.

Röviden, az SQL befecskendezés a biztonsági támadások egy olyan módja, amely a felhasználói inputot kihasználva engedély nélkül SQL utasításokat hajt végre a rendszeren, amely gyakran rongáló hatással van, vagy olyan adatokhoz férnek hozzá, amihez nem szabadna. Egy példával egyszerűen demonstrálható:

Tegyük fel, hogy a felhasználótól elkérjük a nevét, és az átadott szöveget semmilyen szinten nem szűrjük le. Ebben az esetben egy rosszakarató felhasználó bármiféle akadály nélkül beírhatja, hogy a neve nem más, mint „Robert’); DROP TABLE users;--”, ebben az esetben az adatbázis több szintaktikailag jó SQL parancsot kap, és megpróbálja mindet végrehajtani, amik közül az egyik nem lesz más, mint „DROP TABLE users;” ami természetesen semmilyen körülmények között nem jó, így az ilyesfajta támadásokat nagyon fontos elkerülni.

Szerencsére, ezzel nem kell kézzel foglalkozni, mert a Spring Data JPA belső működése szerint alapvetően PreparedStatement-eket használ, pár kivétellel persze.

A Spring Data JPA lényege az, hogy az teljesen elabsztraktálja az adatelérési réteget az adott projektben. Alapvetően, a Spring Data JPA egy újabb absztrakciós réteget biztosít tetszőleges JPA implementációk felett, tovább egyszerűsítve az adatelérést. A rendszer főbb részei az

Entity-k és az azokhoz tartozó Repository-k, amik együtt alkotják az MVC architektúra Modell részét.

3.5.1 Az entitások

```
@Entity
@Table(name = "site_user", schema = "public")
public class SiteUser {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "id_user")
    private Long id;

    @Column(name = "username", nullable = false)
    private String username;

    @JsonProperty(access = JsonProperty.Access.WRITE_ONLY)
    @Column(name = "password", nullable = false)
    private String password;

    @JsonProperty(access = JsonProperty.Access.WRITE_ONLY)
    @Column(name = "e_mail", nullable = false)
    private String eMail;

    @OneToMany(fetch = FetchType.LAZY)
    @JoinColumn(name = "id_user", referencedColumnName = "id_user")
    private Set<RegionalAccount> regionalAccounts;

    @ManyToMany(mappedBy = "teamMembers", fetch = FetchType.LAZY)
    @JsonIgnore
    private Set<Team> userTeams;
```

4. ábra: Egy példa egy entity-re

A 4. ábrán látható entitás és hozzá hasonló többi entitás arra szolgál a kódban, hogy a Java kódban könnyen kezelhető formátumban tárolja el a Repository által lekérdezett adatokat, vagy azokat az adatokat, amiket az adatbázisba szeretnénk írni. Sajnos ezeknek nem tökéletes a fejlesztői környezetek által való támogatottsága, mint a 4. ábrán is látható, néha jó, korrekt dolgokat is hibaként kezel a Java linter.

Az Entitás kódjában láthatunk többféle és fajta annotációt, ezek a közül a fontosabbak:

- @Id
- @Column
- @OneToMany
- @Entity
- @JsonProperty
- @JsonIgnore

Az első, @Id annotáció jelentése egyszerű: A megjelölt változó (ami az adatbázisban egy attribútum) része az elsődleges kulcsnak. Ez kicsit bonyolódhat, abban az esetben mikor összetett kulcsra van szükségünk, amely esetben több megoldás és lehetséges, mint például az @IdClass vagy az @EmbeddedId annotációk.

A második, @Column annotáció sem sokkal bonyolultabb, jelentése mindössze annyi, hogy az adatbázis melyik attribútumának értékét tartja a változó, pár plusz beállítással, mint például, hogy lehet-e NULL az érték, vagy, hogy egyedi kell-e legyen az értéke.

A harmadik, @OneToMany annotáció a táblák értékei között kapcsolat kifejezésére hivatott, ennek a segítségével adott entitáshoz tartozó entitásokat úgy tudunk elérni, mintha a Java objektum részei lennének. A FetchType.Lazy annyit jelent mindössze, hogy a gyermek objektumok betöltése addig nem történik meg, amíg nem próbálunk hozzáférni, ezzel gyakran processzor-időt és memória helyet lehet spórolni.

Az @Entity annotáció pusztán arra szolgál, hogy jelezze a rendszer felé, hogy a jelölt osztály egy entitás és repository osztályokban használható.

A @JsonProperty annotáció és annak az „access = JsonProperty.Access.WRITE_ONLY” beállítása biztonsági célokat szolgálnak, mégpedig azt, hogy a rendszer soha ne adja vissza a hash-elt jelszót kivéve, ha a backend közvetlen kéri el, és azt, hogy véletlen se szegje meg a rendszer az adatvédelmi elveket és szivárogtassa ki a felhasználók e-mailjeit jogosulatlan személyeknek.

A @JsonIgnore viszont már fontosabb célokat szolgál, hisz, ha a kapcsolatok mind két oldala tartalmazza a másikat, akkor bármely olyan esetben, mikor egész objektumot kívánunk átadni, végtelen rekurzió keletkezik, ami természetesen nagyon nem kívánatos és mindenképp

elkerülendő. Kihasználva azt, hogy a JPA belül FasterXML Jackson JSON parsert használ kikerülhetjük a végtelen rekurziót azzal, hogy mikor JSON-t próbál létrehozni, nem engedjük, hogy hozzáférjen a változóban lévő adatokhoz.

3.5.2 A repository-k

```
@Repository
public interface SiteUserRepository extends JpaRepository<SiteUser, Long> {

    Optional<SiteUser> findUserById(Long id);

    Optional<SiteUser> findSiteUserByUsername(String username);

    boolean existsByUsername(String username);
}
```

5. ábra: Egy repository

A repository osztályoknak viszonylag egyértelmű a feladata, mégpedig az, hogy absztraktálják és kezeljék az adatbázis adott táblájával végzett műveleteket, mint például az írást, olvasást, módosítást és törlést. Láthatjuk, hogy ebben az osztályban egyáltalán nincs SQL kód.

SQL helyett, a `@Query()` annotáció segítségével tudunk saját JPQL lekérdezéseket írni, de mégis mi az a JPQL? A JPQL rövidítés jelentése nem más, mint a: Java Persistence Query Language.

A JPQL egy olyan lekérdező nyelv, amely arra használható, hogy a Java perzisztens osztályain kereséseket definiáljunk, függetlenül attól, hogy milyen módon vannak azok a rendszerben tárolva [7]. Ezt figyelembe véve, könnyű látni, hogy egy igen hasznos eszköz tud lenni, mikor még nem tudjuk a pontos tárolási mechanizmusát az adatoknak, vagy az megváltozhat a jövőben, viszont mi ennek ellenére nem használtunk közvetlen JPQL-t.

A Spring Data JPA képes a Repository-n belüli metódusok neve alapján fordítási időben létrehozni a szükséges lekérdezéseket, ezzel megspórolva nagy mennyiségű SQL/JPQL írását, jelentősen gyorsítva a fejlesztést. Ezt természetesen kihasználtuk, hisz láttuk, hogy nincs értelme újra feltalálni azt, amit már más jobban megcsinált előttünk.

Természetesen, a `JpaRepository` szuperosztály szolgáltat több gyakran használt metódust alapból, mint például a `findAll()` vagy a `save()`, így ezeket nem volt szükséges kiírni, sőt,

kifejezetten nem ajánlott, amennyiben máshogy megoldható, hisz ezek a gyári metódusok jobban voltak tesztelve, mint a kézzel újra implementáltak.

Az entitásokkal ellentétben, ez az osztály mindössze egyetlen annotációval van ellátva, mégpedig az `@Repository`-val, ennek a jelentése kifejezetten egyszerű, arra szolgál, hogy jelezze, hogy az annotált osztály egy Repository, vagyis egy eszköz, melyen keresztül az adatok tárolása, lekérése és keresése lehetséges. Természetesen ez azt is jelzi, hogy erre az interfészre automatikusan generálni kell egy implementációt. [8].

3.5.3 Csatlakozás az adatbázishoz

Fontos lehet még megjegyezni azt, ahogyan a kapcsolat létrejön a futó Java kód és az adatbázis között. Alapvetően, a PostgreSQL figyel egy megadott TCP/IP portot, ez általában konvenció és az alapbeállítások miatt az 5432-es port, amin keresztül a tetszőleges nyelvben íródott kód tud kommunikálni az adatbázis szerverrel egy egységes interfészen keresztül.

Spring Boot alatt a kapcsolat kialakításának legegyszerűbb módja az, hogy a Spring application.properties fájljában beállítjuk a szükséges adatokat, mint például jelszó, felhasználónév, vagy URL, és engedjük, hogy a kapcsolat automatikusan létrejöjjön a Spring által.

```
spring.datasource.url= ${POSTGRES_URL}
spring.datasource.username=${POSTGRES_USERNAME}
spring.datasource.password=${POSTGRES_PASSWORD}
```

6. ábra: Kapcsolat konfigurálása.

Látható, hogy ezek az érzékeny adatok nem lettek beleírva a fájlba közvetlenül, hanem környezeti változóként vannak kezelve, így elkerülve a lehetséges biztonsági problémákat abban az esetben, ha valaki jogosulatlanul jutna a fájl tartalmához, továbbá ezzel jelentősen könnyebb a rendszer hordozhatóságát megoldani, hisz ebben az esetben nem szükséges kódot átírni, hogy tetszőleges, másik fizikai vagy virtuális gépen lévő adatbázishoz kapcsolódjon.

3.6 Az adatbázis elérhetőségének gyorsítása.

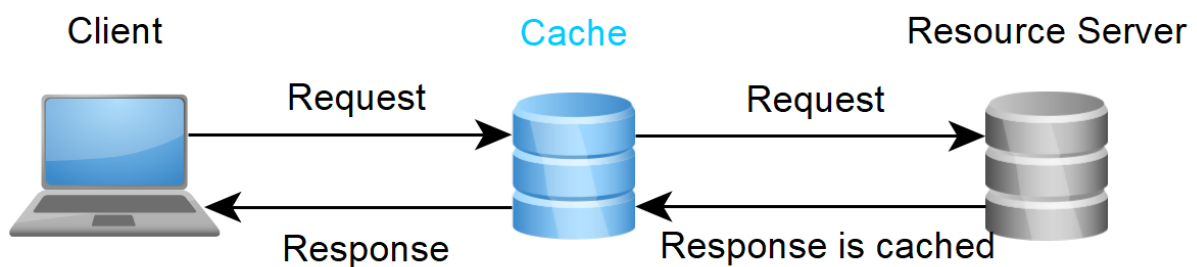
Mivel a PostgreSQL nem egy memóriában tárolt adatbázis, ezért bármilyen adat eléréséhez az adatbázisnak a merevlemezről (vagy tartós állapotú meghajtóról) kell az adatokat kiolvasnia, aminek az elérése nagyságrendekkel lehet lassabb, mint a memóriából kinyert adatoké, az olvasási sebességről nem is beszélve, ami szintén ugyan úgy nagyságrendekkel lehet lassabb.

Mivel a legtöbb rendszerben fontos a gyors elérhetőség, itt is igyekeztünk egy megoldást készíteni arra az esetre, mikor több felhasználó akarja egyszerre elérni és tegyük fel, hogy nagyobb mennyiségű adatot szeretnének lekérni, például meg szeretnék tudni az összes verseny nevét.

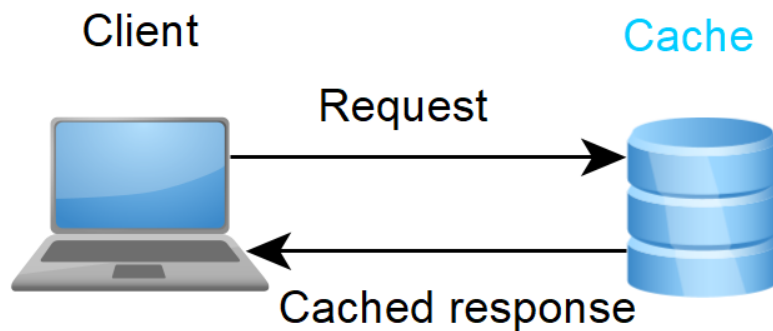
Ezek az adatbázis hívások és SQL lekérések nem lassúak, ha csak egyszer kell lefutniuk, viszont, ha például már tízszer, akkor már kifejezetten sok időbe kerülnek.

Erre a problémára általában a cache-elés kifejezetten jó megoldás, ezért mi is ezt választottuk. Lényegében arról van szó, hogy a háttértárt használó adatbázis mellé fenntartunk egy memóriában tárolót is, amiben, ha nem található meg a keresett adat, akkor azt elkérjük a háttértárban tároló adatbázistól. A végeredményt eltároljuk a memóriában tároló adatbázisban, az eredményt pedig tovább küldjük a kliens felé, így legközelebb mikor szükség van az adatokra akkor gyorsan, a memóriából kiolvasva lehet elérni azokat.

Persze lépnek fel emiatt új problémák is, mint például a cache-ek érvénytelenítése, vagy a cache-ben található adatok frissen tartása, hisz, ha elavult adatot szolgálunk ki a felhasználónak, az rosszabb mintha lassan szolgáltunk volna ki jó adatot.



7. ábra: Működés, ha a cache-ben nincs benne a kívánt információ



8. ábra Működés, ha a cache-ben fellelhető a szükséges információ

A fenti ábrákon látható, hogy az elképzelés nem különösebben bonyolult, sőt, egyszerű. Viszont egy jó cache megvalósítása nulláról már jóval nehezebb lenne, de szerencsére a Spring Boot itt is segítségünkre volt.

A cache kialakításának az első lépése a `@EnableCaching` konfiguráció-annotáció használata volt, ennek annyi volt a feladata mindössze, hogy jelezze a Spring felé, hogy készítse elő a rendszert a cache-eléshez.

```
@Override
@Cacheable(value = "tournaments", key = "#id")
public Tournament getById(Long id) {
    return tournamentRepository.findById(id)
        .orElseThrow(() -> new ResourceNotFoundException("Tournament not found."));
}

@Override
@CacheEvict(value = "tournaments", key = "#tournament.id")
public void validateAndDelete(Tournament tournament, SiteUser siteUser) throws UnauthorizedException {
    if (tournament.getCreatorId().equals(siteUser.getId())) {
        tournamentRepository.deleteById(tournament.getId());
    }
    throw new UnauthorizedException();
}

@Override
@CachePut(value = "tournaments", key = "#tournament.id")
public Tournament validateAndSaveNewName(Tournament tournament, SiteUser siteUser, String newName)
    throws UnauthorizedException {
    if (siteUser.getId().equals(tournament.getCreatorId())) {
        tournament.setTournamentName(newName);
        return tournamentRepository.save(tournament);
    }
    throw new UnauthorizedException();
}
```

9. ábra: Egy cache-elt service

A képen néhány cache-elt metódus látható, amelyek mind egy interfész metódusait implementálják, ezért mind el van látva a java `@Override` annotációjával, meg a cache-eléshez szükséges annotációkkal:

- `@Cacheable`
- `@CacheEvict`
- `@CachePut`

Az első, a `@Cacheable` azt adja meg, hogy az annotált metódus eredményét el kell cache-elni a „tournaments” nevű cache-ben, aminek a kulcsa a metódus egyetlen argumentuma, az ID.

A második, `@CacheEvict` annotáció azt jelzi, hogy a metódus lefutása után törölni kell a „tournaments” nevű cache-ből azt az elemet, aminek a kulcsa megegyezik a kulcs mezőben megadottal. Ennél az annotációnál továbbá beállítható még az is, hogy a cache-t teljesen kiürítse, ami néhány metódus hívásnál szükséges lehet.

A harmadik annotáció. A `@CachePut` pedig, mint ahogy a neve is sejteti, a cache frissítésére szolgál, szimplán annyi történik ebben az esetben, hogy a cache adott kulcsú elemét frissíteni kell a friss információval.

A cache nem magát az adatbázis adatait tárolja el a gyorsítótárban, hanem a Service-ek hívásának az eredményét, de igazából ezek megegyeznek hatásukban, mivel a Service réteg lényegében egy absztrakciós réteg az adatbázis fölött.

4. A Logikai réteg

A rétegek elválasztásának több oka is van, de az elsődleges az, hogy az adatelérési logika belső működése nem lényeges a logikai réteg számára, vagyis a logikai réteget nem érdekli, hogy hogyan és honnan jönnek elő a szükséges adatok, csak azt, hogy jöjjenek.

Természetesen voltak olvashatósági érvek is, hisz a logikai réteg nagyon gyorsan nehezen átláthatóvá és olvashatóvá válhat, ha nincsenek elkülönítve a rétegek megfelelően.

4.1 Az osztályok elrendezése

A projektben fellelhető osztályok logikus, érthető elrendezése is egy szempont volt, ezért a lehető legjobban próbáltuk kihasználni a Java package rendszer előnyeit.



10. ábra: Példa a package-elrendezésre

A fenti ábrán jól látszik, hogy az úgynevezett Common-Closure Principle jól alkalmazható Java csomagokra. Az elv lényege az, hogy az olyan csomagokat, amelyek feltehetőleg együtt fognak változni egy közös csomagba csomagoljuk, vagyis minden csomagnak a lehető legkevesebb oka legyen a változásra [9].

A csomagok elrendezésének a célja az is volt, hogy minden levélelem csomag a lehető legkisebb felelősséget ölelje fel és a lehető legkevesebb oka legyen a változásra. Jelen példában, a „Persistence” nevű csomag csak is az adatbázissal való kommunikációval foglalkozik, üzleti logikát nem tartalmaz.

Az előzőhöz hasonlóan, a Service osztály csak a perzisztancia réteg által szolgáltatott adatok elérésével, azoknak továbbjuttatásával, feldolgozásával foglalkozik, míg ezzel szemben a kontroller réteg feladata a többi réteg adatainak kommunikálása a távoli klienssel.

Látható, hogy az ilyesfajta elrendezés szükséges, hisz a projekt nagyon gyorsan átláthatatlanná vált volna, mivel több, mint negyven különböző Java osztályt tartalmaz és több mint húsz csomagot.

Természetesen van jelentős összefüggés a rétegek között, például a Service réteg a Persistence rétegre épült, míg a kontroller réteg függ a Service rétegtől.

4.2 Service réteg

A service réteg feladata az üzleti logika definiálása, annak megvalósítása. Létezésének az egyik legfontosabb oka, hogy egy újabb absztrakciós réteget hozzon létre a Controller és a Persistence réteg között.

Az absztrakciós réteg hasznosságát egy egyszerű példával könnyű demonstrálni:

```
@Override
public SiteUser getCurrentlyLoggedInSiteUser() {
    Authentication authentication = SecurityContextHolder.getContext().getAuthentication();
    return siteUserRepository.findSiteUserByUsername(authentication.getName())
        .orElseThrow(() -> new ResourceNotFoundException("User not found"));
}
```

11. ábra: Példa egy service-beli metódusra

A 11. ábrán látható kód talán a legegyszerűbb service-metódus, amit fel tudtunk hozni példának, de jól demonstrálható rajta a réteg szükségessége.

A működése nem bonyolult, mindössze annyi csinál, hogy elkéri a globális autentikációs állapottól az adott kontextusban aktív felhasználó felhasználónevét (magyarán szólva azt a felhasználót, aki a kérést végezte), és az egyedi felhasználó név alapján lekéri minden más adatát az adatbázistól, továbbá felkészül arra az esetre, ha nem létezik olyan nevű felhasználó, amely esetben egy kivételt dob, akármennyire is legyen kicsi az esély egy ilyen hibára.

A kódot megfigyelve, könnyen elképzelhető, hogy ez a metódus sok, akár tucatnyi helyen is fel van használva, szóval mi van, abban az esetben, ha meg szeretnénk változtatni, például azt, hogy milyen üzenet van a kivételben? Ha nem lenne service réteg, akkor az egy hosszú és lassú folyamat lenne, amiben könnyen keletkezhetnek esetleges hibák, amiket szintén idő lenne megkeresni, kijavítani.

Természetesen vannak hátrányai is a rétegnek, mint például az, hogy eggyel több metódushívás szükséges a feladat elvégzéséhez, ami egy kevés processzoridőbe kerül, de annak a pár utasításnak a végrehajtási ideje elhanyagolható, a logika végrehajtási bonyolultságához képest gyakorlatilag nullához közelít.

Másik felhozható hátránya tovább az, hogy az amúgy is nagyon sok osztályt tartalmazó Java nyelvű kódot további osztályokkal toldja meg, ami nem feltétlen kívánatos, részben emiatt az

elv miatt van több mint negyven osztály és több mint húsz csomag a projektben, hisz minden hozzáadott service két teljes Java osztályt ad a projekthez és egy plusz csomagot.

Talán legfontosabb előnye a Service rétegnek az, hogy segít összegyűjteni a függőségeket, mint például a Repository osztályokat, így azokat nem kell közvetlenül injektálni a Controller osztályokba. Az injektálás részletes tárgyalására később fog sor kerülni.

A 9. ábrán megfigyelhető, hogy a cache kialakításában is nagy segítség volt a service-architektúra, hisz így a service-eken keresztül, absztrakt módon volt lehetséges eltárolni az alacsonyabb rétegek eredményeit.

A 10. ábrán megfigyelhető, hogy a Service-osztályok nem csupán a semmiből jönnek létre, hanem mind egy-egy interfésznek az implementációja.

```
public interface SiteUserService {  
  
    SiteUser getCurrentlyLoggedInSiteUser();  
  
    SiteUser getArbitraryUser(Long id);  
  
    void modifyNameAndSave(SiteUser siteUser, String name);  
  
    boolean registerUser(SiteUserRegisterer siteUserRegisterer);  
  
    boolean changePassword(SiteUser siteUser, String oldPassword, String newPassword);  
  
    boolean deleteAccount(SiteUser siteUser, String password);  
}
```

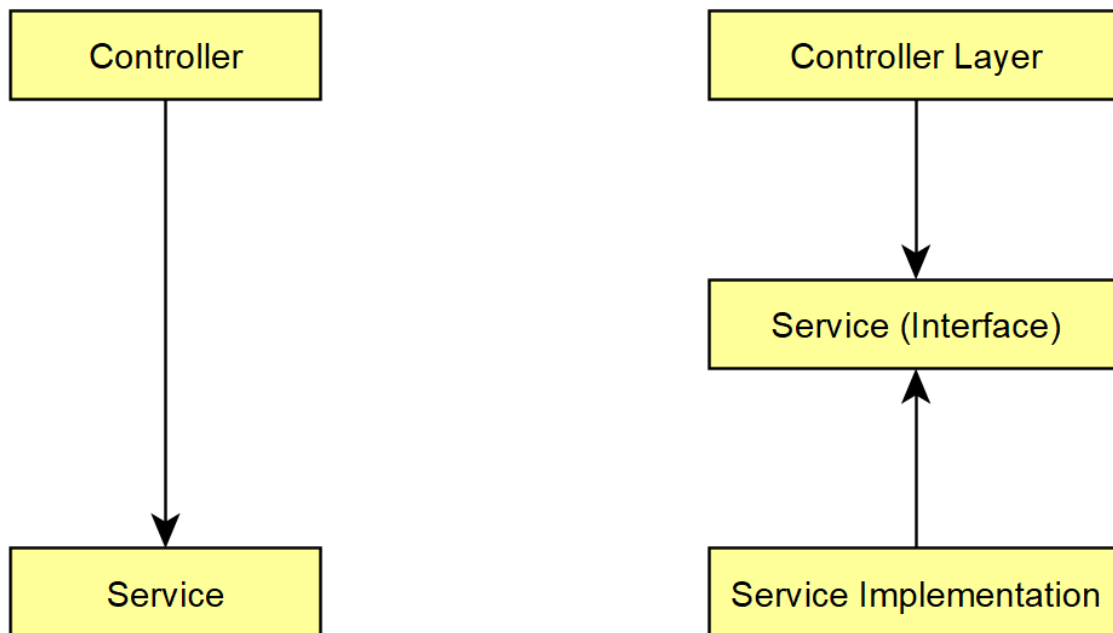
12. ábra: Példa egy Service interfészre

Ennek több oka is van, de legegyszerűbben és talán legjobban a SOLID elvek közül a legutolsó fejezi ki, Függőség Megfordítás elv.

A Függőség Megfordítás elv kimondja, hogy a magas szintű modulok ne függjenek az alacsony szintű moduloktól, hanem absztrakcióktól, mint például egy interfész. Pontosan ez miért is jó? Röviden, rugalmasságot és stabilitást nyújt a projektnek szerkezeti szinten, hisz kicserélni egy interfész implementációját sokkal könnyebb és gyorsabb, mint teljesen új osztályt írni, de ha esetleges mock-oló tesztelésre van szükség, az is sokkal könnyebb interfészeken keresztül.

Említésre került korábban, hogy az interfészeknek vannak implementációik, amikre példák megfigyelhetők a 9. és 11. ábrákon. Lényegében, ezek annyit tesznek, hogy tényleges metódus-

testet adnak az interfészben definiált metódus-fejekhez, amit az ábrákon látható @Override annotáció jól jelez.



12. ábra: Hagyományos és függőség invertált architektúra

A 12. ábrán jól látható, miért nevezik függőség megfordítása elvnek, az is jól látható, hogy a megvalósítása és elve se bonyolult. A neve is abból ered, hogy az az osztály, ami eredetileg a függőség volt, függ egy interfésztől, és tőle közvetlen nem függ semmi más.

4.3 A Controller réteg

A java-oldalon fellelhető egyik legfelsőbb réteg nem más, mint a Controller osztályok által alkotott réteg, ezek olyan osztályok, melyek feladata a kliensekkel való kommunikáció levezetése, a helyes üzleti logika meghívása, a létrehozott adat helyes formátumban való átadása, ilyen formátum például a JSON, de lehetne akár XML vagy CSV is.

```

@RestController
public class TournamentController {

    private final TournamentToTeamsService tournamentToTeamsService;
    private final TeamService teamService;
    private final SiteUserService siteUserService;
    private final TournamentService tournamentService;

    @Autowired
    public TournamentController(TournamentToTeamsService tournamentToTeamsService,
                                TeamService teamService,
                                SiteUserService siteUserService,
                                TournamentService tournamentService) {
        this.tournamentToTeamsService = tournamentToTeamsService;
        this.teamService = teamService;
        this.siteUserService = siteUserService;
        this.tournamentService = tournamentService;
    }

    @PostMapping(value = "/api/tournament/create", consumes = "application/json", produces = "application/json")
    public ResponseEntity<String> createNewTournament(@RequestBody Tournament tournament) {
        try {
            if (!tournamentService.validate(tournament)) {
                return new ResponseEntity<>(
                    body: "\"Error, no field can be empty, null, or compromised of only whitespaces.\",",
                    HttpStatus.BAD_REQUEST);
            }

            SiteUser siteUser = siteUserService.getCurrentlyLoggedInSiteUser();
            tournamentService.setCreatorAndSave(tournament, siteUser);

            return new ResponseEntity<>( body: "\"Saved tournament.\",", HttpStatus.OK);
        } catch (DataIntegrityViolationException e) {
            return new ResponseEntity<>( body: "\"Tournament name already in use.\",", HttpStatus.BAD_REQUEST);
        } catch (ResourceNotFoundException e) {
            return new ResponseEntity<>( body: "\"Error: " + e.getMessage() + "\",", HttpStatus.NOT_FOUND);
        }
    }
}

```

13. ábra: Példa egy kontrollerre

4.3.1 Annotációk

A 13. ábrán megfigyelhető egy példa egy kontrollerre, amin látható, hogy itt már viszont több dolog is történik, csak azért is, mert itt már több, bonyolultabb annotáció is szerepel, ezek bármiféle fontossági sorrend nélkül:

- @RestController
- @PostMapping
- @RequestBody
- @Autowired

A Spring által biztosított @RestController annotáció igazából egy rövidítés két annotációra, a @Controller és a @ResponseBody annotációkra. Az @Controller jelentése eléggé egyszerű,

mindössze annyit tesz, hogy jelzi a Spring felé, hogy ez egy komponens és mégpedig egy specializált, kontroller komponens. A `@ResponseBody` igazából annyit tesz, hogy az annotációnak köszönhetően akármit is adnak vissza az osztályban található metódusok, azokat JSON formátumúvá alakítja, ami furcsa lehet első ránézésre, mert az első példában csak is egyetlen String-értéket adunk vissza, ami szerencsére viszont egy szabályos JSON dokumentum [10], így a kliensoldali JavaScript általánosan le tudja kezelni, ugyanúgy, mint az adatbázis adatait tartalmazó JSON dokumentumokat.

A második annotáció is igazából egy rövidítés, mert a `@PostMapping` valójában a `@RequestMapping` egy specializált változata, ami annyit fejez ki, hogy ez a metódus HTTP POST metódust használ, de vannak még specializált annotációk a GET, HEAD, PUT, PATCH és DELETE metódusokra is, lefedve ezzel a fontosabb HTTP metódusokat.

A `@PostMapping` argumentumai között több dolgot is meg lehet figyelni, mint például a „value”, „consumes”, és „produces” mezőket, ezek közülük a „value” a legfontosabb.

Az annotáció „value” mezője megadja, hogy az alapnak beállított URL-t felhasználva, milyen URL-t figyeljen a szerver az adott fajta kérésért, ami, ha teljesül meghívja metódust. Például, az ábrán látható metódus abban az esetben kerül meghívásra, ha a kliens egy HTTP POST kérést küld a <http://pelda.com/api/tournament/create> címre és természetesen akkor és csak akkor, ha kérésben lévő adatok helyes formátumban vannak. Az ábrán látható metódus, azért HTTP POST kérést vár el, mert itt új erőforrást kerül létrehozásra és az ilyesfajta kérésekhez, a szabvány értelmében, a POST metódus az elvárt [11].

A `@PostMapping` további argumentumai, a „consumes” és „produces”, egyszerűen azt jelölik, hogy a metódus JSON formátumú adatot fogad el és azt is ad vissza, igazából ez a két érték pusztán beállítja a szükséges HTTP header értékeket, mint például az „accept”.

Fontos még az `@RequestBody` annotáció is, hisz ezzel könnyen lehet jelezni a Spring felé, hogy milyen kialakításban várja el az adatokat a kérés body-részeiben. Tegyük fel, hogy van egy egyszerű osztályunk, aminek két tagja van csak: First, és Second. Ebben az esetben a Spring olyan JSON-t vár el, ami egy objektum, két mezővel, First és Second. Ha minden megfelel, a Spring automatikusan deszerializálja a JSON dokumentumot, nekünk pedig csak a létrejött objektummal kell foglalkoznunk, aminek köszönhetően sok triviális kód írása elkerülhető.

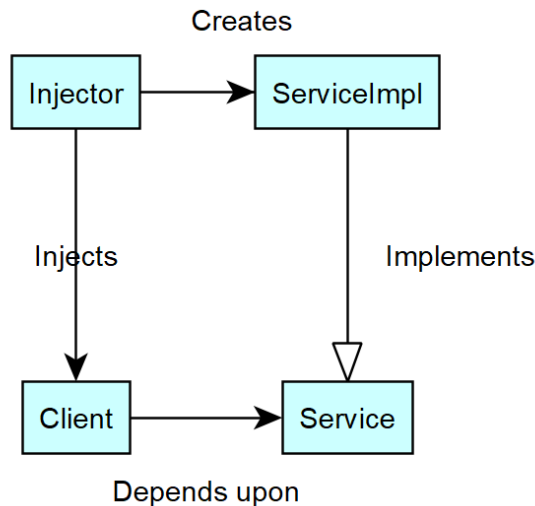
A projektben vannak olyan esetek is, amikor nagyon nehézkes lett volna külön osztályt létrehozni a deszerializációra, ezért csak simán egy `Map<String, Object>` objektumba deszerializálódik a dokumentum, ami után, ámbár kevésbé kényelmesen, de kulcs-érték páronként elérhetőek az adatok, ahol a kulcs a JSON mező neve, értéke pedig természetesen a JSON mező értéke.

Az `@Autowired` annotáció kitárgyalása előtt érdemes lehet megemlíteni az `@PathVariable` annotációt. Ez az annotáció, a `@RequestBody`-hoz hasonlóan arra szolgál, hogy adatokat, argumentumokat adjunk át a Controller osztály metódusainak. Jelen esetben, ez az annotáció arra szolgál, hogy az URL-ből nyerjünk ki adatokat. Sajnos ez az annotációt csak akkor érdemes használni, hogyha az átadni kívánt adat csak egyetlen, atomi részből áll, mint például egy szóból álló szöveg, vagy egy szám. Az alábbi példa alapján, az „`/api/users/12/accounts`” végződésű URL hívásból, a 12-es szám fog bekerülni, mint `userId` változó.

```
@GetMapping(value = "/api/users/{userId}/accounts", produces = "application/json")
public ResponseEntity<Object> getUsersAccounts(@PathVariable Long userId) {
```

14. ábra: A `@PathVariable`

Az utolsó, talán legfontosabb annotáció pedig az `@Autowired` annotáció volt. Ennek a szerepe igen fontos és a működése sem triviális, hisz ez az annotáció valósítja meg a függőség-befecskendezést. A függőség-befecskendezés egy olyan módszer, amely segítségével adott osztálynak (továbbiakban kliens) a függőségeit átadjuk. A függőség injektálás módszer lényege az, hogy az objektumok használatát elválasztjuk a létrehozásuktól, effektíve függetlenné téve a klienst attól, ahogyan létrejönnek az általa használt függőségek [12]. Természetesen ehhez szükség van egy injektáló eszközre is, amiben a Spring nagy segítségre volt.



15. ábra: Függőség-befecskendezés

A Spring Keretrendszer fordítási időben megkeresi az összes úgynevezett „Bean” -t, ami lényegében egy olyan singleton objektum, amit a Spring IoC (Inversion of Control) Container hoz létre és kezel, ezt többféle képpen is lehet jelezni a Container felé, akár @Bean (vagy az abból származó @Controller és @Service) osztály-annotációval, vagy XML konfigurációval, bár az utóbbi használata manapság már nem ajánlott. Az Inversion of Control egy elv, arra, hogy definiáljuk egy objektum függőségeit anélkül, hogy azokat létrehoznánk. Láthatjuk, hogy a függőség befecskendezés módszere ennek csupán egy megvalósítása.

A Spring függőség befecskendezés megvalósítása sem túl bonyolult. Lényegében az Inversion of Control Container valahonnan kiolvassa a szükséges konfigurációs metaadatokat (például XML fájlból, Java kódból vagy annotációkból) és ezek alapján létrehozza a szükséges Bean-eket, amiket oda fecskendez be, ahol az megengedett, például egy @Autowired annotációval rendelkező konstruktorba, vagy annotált Setter-be, de ez nem a legjobb megoldás, általánosan a konstruktoros injektálást ajánlott használni.

Összességében, a függőség befecskendezés előnye az, hogy sok ismétlődő programrészlet megírását el lehet vele kerülni. Továbbá jelentősen csökkenti az osztályok közötti csatolást, mivel ennek köszönhetően az osztályok nem konkrét implementációktól, hanem interfészeketől függenek, ami hosszútávon könnyebben fenntartható kódbázist eredményez. Ezért ennek a projektnek az elkészítése közben is nagy előszeretettel használtuk.

4.3.2 A design

A 13. ábrán több, fontosabb kitárgyalható rész is van, ezek bármiféle fontossági sorrend nélkül:

- Az kivételek.
- A visszaadott objektum fajtája.
- A HTTP státusz kódok.
- Az /api/ URL-út prefixum.

Elsősorban talán a kivételekről érdemes hosszában tárgyalni. A 13. ábra példájában látható két kivétel, a `ResourceNotFoundException` és a `DataIntegrityViolationException`.

A `ResourceNotFoundException` a projekt kódjában, úgymond „kézzel” kerül dobásra, ez a kivétel a `RuntimeException`-ből származtatott le, ami azt jelenti, hogy nem kötelező elkapni egy Try-Catch blokkban, de mégis jó ötlet. A `ResourceNotFoundException` jelentése általánosan az, hogy ahogy a neve is kifejezi, a keresett erőforrás nem található, legyen az fájl, vagy az adatbázisban fellelhető információ.

```
} catch (UnauthorizedException e) {  
    return new ResponseEntity<> ( body: "\"Forbidden: You are not the creator of the tournament.\"\"",  
        HttpStatus.FORBIDDEN);  
}
```

16. ábra: Az `UnauthorizedException`

Az `UnauthorizedException`, a `ResourceNotFoundException`-höz hasonlóan a `RuntimeException` egy leszármaztatott osztálya, ezért nem kötelező elkapni, de erősen ajánlott. Viszont, a `ResourceNotFoundException`-nel ellentétben, ez a kivétel nem a Java API vagy valamely könyvtár része, hanem a projektben létrehozott, specializált kivétel. Mivel az ehhez hasonló nevű `IllegalAccessException` a reflexió műveletekre vonatkozott, nem a tényleges biztonsági problémákra, ezért kénytelenek voltunk saját kivételt definiálni, ami szerencsére nem volt túl bonyolult.

```

public class UnauthorizedException extends RuntimeException{

    public UnauthorizedException(String errorMsg){
        super(errorMsg);
    }

    public UnauthorizedException(){
        super();
    }

}

```

17. ábra: Az egyszerű, atomi kivétel definiálása

Látható, hogy egy egyszerű kivételt nem túl nehéz definiálni, pusztán bátran kell használni a java által biztosított „super” levédett szót, ami jelen esetben mindössze meghívja a szuperosztály (a RuntimeException) adott konstruktorát [13].

Az utolsó a 13. ábrán megfigyelhető kivétel a DataIntegrityViolationException. Ennek a szép hosszú nevű kivételnek a szerepe is fontos, hisz ez az a kivétel, amit a perzisztencia réteg dob, abban az esetben, ha adott műveletet nem lehet végrehajtani az a perzisztencia ellátón, legyen az akármilyen is. Jelen esetben, a PostgreSQL csatlakozó ezt akkor dobja, hogyha a művelet megsértene bizonyos Constraint-eket, ami a példa esetében akkor keletkezik, ha az adott írási művelet olyan bajnokságot próbál eltárolni az adatbázisban, aminek a neve egyezik egy másik, már létező bajnoksággal, annak ellenére, hogy az az attribútum nem kulcsa a táblának. Ezt a PostgreSQL oldalon természetesen a UNIQUE constraint-el van megvalósítva. Lényegében, ez a kivétel segít elkerülni az inkonzisztenciákat a frontend, backend, cache és adatbázis között.

Megfigyelhető, hogy a Controller metódusai nem csak pusztán az elkért objektumokat, adják vissza, hanem azokat egy ResponseEntity<T> objektumba csomagolva továbbítják a kliens felé, mivel a ResponseEntity<T> egy parametrikusan polimorf osztály, ezért közel bármilyen Java-osztályt elfogad, mint paraméter, még az Object ősosztályt is, ezzel jelentősen könnyítve a visszatérési értékek kialakítását.

A ResponseEntity<T> osztály fő célja, hogy a szükséges adatok mellé megadja a szükséges HTTP válasz kódot. A HTTP protokoll válaszkódjai, mint ahogy az elnevezésük is sejteti, azt a célt szolgálják, hogy megadják a kérés sikerességét, vagyis státuszát. A státusz kódokat 5

csoportba soroljuk a szabványnak megfelelően [14], de ez a projekt csak néhányat használ explicit módon:

- 200 (OK),
- 202 (Elfogadva),
- 400 (Rossz kérés),
- 403 (Tiltott),
- 404 (Nem talált),
- 500 (Szerverhiba)

A HTTP státuskódok helyes használata jelentősen könnyíti a Frontend-en való munkát, hisz ezeknek köszönhetően nem kell kézzel megnézni a válaszok tartalmát, csak a válaszkódot az esetek többségében, elkerülve ezzel a fölösleges, lassú adatfeldolgozást a kliens oldalon, ami természetesen nem tenne jót a Frontend teljesítményének.

Fontos lehet megemlíteni még egy, majdnem az összes API-endpointot érintő témát, mégpedig az endpointokhoz tartozó prefixumot, specifikusan az `/api/` prefixumot, ami a frontendhez tartozó endpointokon kívül minden endpointra vonatkozik.

A lényege röviden az, hogy minden nem frontendes endpointot ellátunk egy `/api/` prefixummal, mivel ezzel könnyen, és minimális munkával és odafigyeléssel el lehet kerülni az endpointok ütközését, ami nehezen debugolható problémává válhat. A frontend-en való útválasztást a `React-router-dom` valósítja meg, ami teljesen független a Spring-es útválasztástól, ezért fontos volt elkerülni az útvonal ütközéseket a projekten belül.

Legvégül. A HTTP protokollra egyszerű okból esett a választás, hisz az egy állapotmentes, elterjedt, könnyen használható, dokumentum átvitelre jól alkalmazható protokoll. A frontend oldalon is nagyon jól van támogatva, mint például a böngésző által végzett implicit GET hívások, vagy a `Fetch API`-val, Javascriptből könnyen végezhető hívásokkal, továbbá bármilyen REST API-t is az esetek túlnyomó többségében is HTTP-vel szokás megvalósítani, nem más állapotmentes protokollon, így egyértelmű volt a választás.

4.4 Aszinkron adatfeldolgozás

A bajnokságok megfelelő kezdőpozícióinak kialakításához sajnos nem volt elegendő a véletlenszerű sorsolás vagy a betűrendi sorrendbe rendezés, hanem valami olyasféle elrendezést

volt szükség kialakítani, ami figyelembe veszi a csapatok nyerési esélyeit, és azok alapján számolja ki a kezdő állapotot.

Ehhez a problémához jött hatalmas segítségre a Riot Games által nyújtott nyilvános API, ami mindössze egy rövid ingyenes regisztráció után egy REST interfészen keresztül képes szolgáltatni rengeteg információt a regisztrált felhasználókról, mint például aktuálisan elért rang, játékbeli szint, lejátszott meccsek száma, vagy a felhasználói ID.

Ezek közül természetesen a rang a megszerzése volt a cél, hisz a játékban lévő kilenc fő rang csoport alapján könnyen megbecsülhetőek a felhasználó csapatának nyerési esélyei.

Természetesen ez a probléma azért létezett, mert szerettük volna elkerülni az olyan szituációkat, ahol egy bajnokságon belül a két legjobb csapat az első fordulóban találkozik, ezzel könnyen unalmassá téve a verseny további részét, hisz annak az esélye, hogy más csapat nyerjen azután jelentősen leesik, ezáltal a győztes kiléte gyakorlatilag az első fordulóban eldől, minden más csapat már csak a második és harmadik helyért küzdhet.

Természetesen voltak problémák is a Riot API-val, mint például az, hogy két API hívásra volt szükség adott játékos rangjának megszerzéséhez, ami azért volt problémás, mert a Riot API másodpercre és két percre levette is limitálva volt, ezért, ha túl sok kérést végeztünk rövid időn egy API kulcs segítségével, akkor a további kéréseket elutasította a rendszer és várni kellett. Természetesen erre is sikerült találni egy megoldást, ami képes volt elkerülni a limitálást, anélkül, hogy a fő végrehajtási szál blokkolná hosszú időre.

Az első probléma, amit meg kellett oldani az a fő szál blokkolásának elkerülése volt. Két irányból lehetett megközelíteni: azonos szálon lévő aszinkron HTTP kérések végzése, mint például a Javascript által biztosított Fetch API, vagy külön szálon végzett HTTP kérések és adatbázis írások.

Végül a kettő egyfajta kombinációjára esett a választás, mégpedig arra, hogy az egész logika egy külön szálon hajtódik végre, de azon a szálon belül aszinkron végződnek a http kérések, vagyis a rendszer nem várja meg a válaszokat mielőtt újabb kéréseket végezne a szolgáltató felé.

4.4.1 A külön szál

A Spring szolgáltató egy beépített aszinkron végrehajtási megoldást, az `@Async` és `@EnableAsync` annotációkon keresztül, de ezek hosszú tesztelés és fejlesztés után

inkonzisztensnek bizonyultak, sőt, bizonyos esetekben nem is bizonyultak aszinkronnak, ami elfogadhatatlan volt. Ennek hatására egy eggyel alacsonyabb szintű megoldást kellett választanunk, ami még mindig jóval magasabb szintű volt, mint például a C és C++ (vagy alap Java) nyelvek többszálú végrehajtás megvalósításai.

A rendszeren belül arra a megoldásra esett a választás, hogy a külön szálon való végrehajtást a hívónak legyen dolga elintézni, nem a service-nek, ezért a konfiguráció és a service is ezt figyelembe véve lettek kialakítva.

```
@Bean
public TaskExecutor taskExecutor(){
    ThreadPoolTaskExecutor executor = new ThreadPoolTaskExecutor();

    executor.setThreadNamePrefix("Async-");
    executor.setCorePoolSize(Runtime.getRuntime().availableProcessors() - 1);
    executor.setMaxPoolSize(Runtime.getRuntime().availableProcessors() - 1);
    executor.setQueueCapacity(100);
    executor.afterPropertiesSet();
    return executor;
}
```

18. ábra: A TaskExecutor konfigurálása

```
executor.execute(() -> rankingService.updateRank(tournament));
```

19. ábra: A TaskExecutor használata

A TaskExecutor interfészt szintén a Spring biztosítja, lényege mindössze az, hogy elabsztraktálja a Java Runnable-ök használatát [15].

A 19. ábrán látható módon pusztán egy lambda kifejezés használatával könnyen lehet külön szálon futtatni tetszőleges kódot, ami jelen esetben csak egyetlen metódus hívása, de a lambda szintaxisnak köszönhetően lehetne kód blokk is. Természetesen a TaskExecutor is függőség befecskendezés útján kerül a Service osztályba bele, aminek az előnyeit már nem szükséges tárgyalni.

A TaskExecutor így külön szálon képes végrehajtani az adott metódust, nem blokkolva a fő végrehajtási szálát, aminek köszönhetően az ezt használó kontroller vissza tud adni egy 202-es (Elfogadva) kódot, ami azt jelenti, hogy a kérés el lett fogadva, tovább lett adva feldolgozásra, ahelyett, hogy akár egy teljes percre hagyná várni a klienst.

Természetesen voltak hátrányai is a több szálon való munkának, például az, hogy a JPA lusta betöltési stratégia nem működik nem fő szálról, ezért az ilyen esetekben a mohó betöltési stratégiát volt szükséges alkalmazni. A külön szálon való dolgozás egyik legnagyobb problémája gyakran nem más, mint hogy a szálaknak meg kell várnia egymás eredményeit, azt, hogy ne módosítsák ugyanazt az adatot egyszerre, vagyis a szinkronizáció kérdése.

Ezt elkerülendő, az adatbázis táblájában, amiben aktuálisan dolgozunk, érdemes valahogy jelezni azt. Jelenleg ezt pusztán egy egyszerű boolean változó beállításával van megvalósítva, ami, ha igaz, a backend nem próbálja meg módosítani az adott táblát, mivel tudja, hogy azt éppen frissíti valamely más szál.

Ezeket elintézve három fontosabb probléma maradt:

- Az egy szálon való aszinkron HTTP kérések létrehozása.
- A RIOT API korlátozások betartása.
- A válaszok deszerializálása használható formátumba.

Mivel a válaszok hosszú, JSON dokumentumok formájában érkeznek, ezért azoknak deszerializálására elegendő ugyan azt a FasterXML-deszerializálót használni, mint a kliens által küldött kérések értelmezéséhez, mindössze annyit pluszt tartalmaz, hogy a könnyű olvashatóság érdekében egy Java Enum osztályt is felhasználunk.

```
public enum RankEnum {  
    IRON((short)0),  
    BRONZE((short)1),  
    SILVER((short)2),  
    GOLD((short)3),  
    PLATINUM((short)4),  
    DIAMOND((short)5),  
    MASTER((short)6),  
    GRANDMASTER((short)7),  
    CHALLENGER((short)8);  
  
    private final short value;  
  
    private RankEnum(short value) { this.value = value; }  
  
    public short getValue() { return value; }  
}
```

20. ábra: Az Enum

```

tournament.setUpdating(true);
tournamentRepository.save(tournament);

String url = riotBaseUrlBuilder(BASE_URL, tournament.getRegionId());

WebClient webClient = WebClient
    .builder()
    .baseUrl(url)
    .defaultHeader(header: "X-Riot-Token", RIOT_TOKEN)
    .build();

List<RegionalAccount> accounts = new ArrayList<>();

for (TournamentToTeams tournamentToTeams : tournament.getTeams()) {
    for (SiteUser user : tournamentToTeams.getTeam().getTeamMembers()) {
        accounts.add(user.getRegionalAccountByRegion(tournament.getRegionId())
            .orElseThrow(() -> new ResourceNotFoundException("Regional Account not found.")));
    }
}

List<List<RegionalAccount>> partitions =
    new ArrayList<>(); //splitting it into easily manageable chunks to avoid rate limiting
for (int i = 0; i < accounts.size(); i += 20) {
    partitions.add(accounts.subList(i, Math.min(i + 20, accounts.size())));
}

```

21. ábra: RIOT API használata, kivonat

```

try {
    List<IdDeserializer> ids = new ArrayList<>();
    for (int i = 0; i < partitions.size(); ) {
        if (REQUEST_COUNTER <= 100 || REQUEST_COUNTER + partitions.get(i).size() < 100) {
            ids.addAll(Objects.requireNonNull(Flux.fromIterable(partitions.get(i)).Flux<RegionalAccount>
                .flatMap(account -> webClient.get().uri(String.format(SUMMONER_URL, account.getInGameName()))
                .accept(MediaType.APPLICATION_JSON)
                .retrieve()
                .bodyToMono(IdDeserializer.class)) Flux<IdDeserializer>
                .collectList() Mono<List<IdDeserializer>>
                .block()));
            REQUEST_COUNTER += partitions.get(i).size();
            Thread.sleep(ONE_SECOND);
            i++;
        } else {
            Thread.sleep(TWO_MINUTES);
            REQUEST_COUNTER = 0;
        }
    }

    List<List<RankDeserializer>> rankings = new ArrayList<>();

    List<List<IdDeserializer>> idPartitions =
        new ArrayList<>(); //splitting it into easily manageable chunks to avoid rate limiting
    for (int i = 0; i < ids.size(); i += 20) {
        idPartitions.add(ids.subList(i, Math.min(i + 20, ids.size())));
    }

    for (int i = 0; i < idPartitions.size(); ) {
        if (REQUEST_COUNTER <= 100 || REQUEST_COUNTER + partitions.get(i).size() < 100) {
            rankings = Flux.fromIterable(idPartitions.get(i)).Flux<IdDeserializer>
                .flatMap(idDeserializer -> webClient.get().uri(String.format(RANK_URL, idDeserializer.getId()))
                .accept(MediaType.APPLICATION_JSON)
                .retrieve()
                .bodyToMono(new ParameterizedTypeReference<List<RankDeserializer>>() {
                }) Flux<List<RankDeserializer>>
                .collectList() Mono<List<List<RankDeserializer>>>
                .block());
            REQUEST_COUNTER += idPartitions.get(i).size();
            Thread.sleep(ONE_SECOND);
        }
    }
}

```

22. ábra: Riot API használata, kivonat

A 21. és 22. ábrákon megfigyelhető, hogy a korábban látott problémák egyértelmű, jól elkülönült lépésekben kerülnek megoldásra. Először az adatbázis frissítés történik meg, annak érdekében, hogy mint említve volt, elkerüljük, hogy egyszerre több szál módosítsa ugyanazt a rekordot.

Másodjára az alap URL összerakása történik meg, ami sajnos nem volt teljesen triviális, mivel az API valamiért kicsit különböző formátumot használ néhány régió API végpontjainál, ami természetesen nem könnyített a feladatban.

Ezt követően mindössze összeállításra kerül egy lista a regionális felhasználókról azt adott régióban, amit azonnal fel is oszt az alkalmazás egy listákat tartalmazó listákba, figyelve arra, hogy csak akkorák legyenek a blokkok, amelyeket még nem blokkol le a RIOT API, ha egy másodpercen belül történik.

A példában utoljára látható lépés pedig az, hogy blokkonként aszinkron HTTP hívásokat küldünk a RIOT API felé, ami után kicsivel több, mint egy másodpercet várunk (pontosan 1010 ezred másodpercet, csak egy kis, minimális plusz időt, hogy biztosan el legyen kerülve a blokkolás).

Meg lehet figyelni, hogy az aszinkron kliens használata hasonlít a funkcionális programozáshoz, amiben persze van logika, hisz a funkcionális dolgok nagyon könnyen párhuzamosíthatóak, ami egy aszinkron HTTP kliensnél igen is fontos. Pontosabban, a kliens használata hasonlít a Java Stream-ekhez, amik hasonlóan párhuzamosak, és jól demonstrálják, hogy a programozása paradigmák eszközei, amiket érdemes olyan problémákon alkalmazni, amikre legjobban illenek.

A kliens több dolgot is csinál, de mélyebben belemenni nem érdemes a legtöbbbe: összerakja az egyedi URL-t minden felhasználóhoz, megadja a deszerializálót, figyeli az elküldött kérések számát és ha túl sok volt, abban az esetben megvárja a két percet, amit az API elvár.

Sajnos ez még csak a felhasználók egyedi ID attribútumának megszerzéséhez volt elég, a következő lépése az algoritmusnak ugyan úgy blokkokra ossza az ID-eket mint korábban a neveket, ezek alapján is URL-eket rak össze, amiket felhasználva elkéri a felhasználó rangját, amit az enum osztály segítségével gyorsan feldolgoz, egész számmá alakít és beleírja az adatbázisba, ami után feloldja a zárást az rekord felett. Természetesen ennek a résznek is figyelnie kell a hívások mennyiségére, ami ugyan úgy került megoldásra, mint az előbbi résznél.

4.5 Aszinkron értesítések

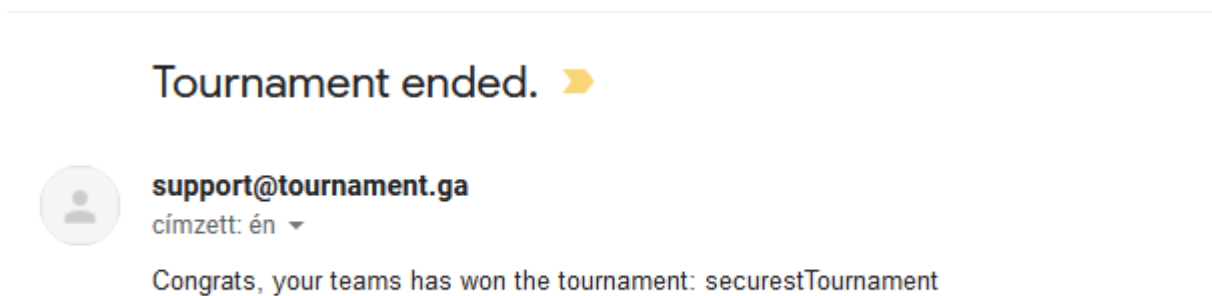
A projekt elkészítésénél egyik cél az volt, hogy a program képes legyen értesíteni a felhasználókat adott bajnokságok kimeneteléről, amit rengeteg sok féleképpen meg lehetett volna oldani, de ebben az esetben az e-mailre esett a választás, mivel ez egy egyszerű és elterjedt technológia.

Mivel saját SMTP szerver kifejlesztése nagyon nagy projekt lett volna, ezért a projekt keretén belül úgy döntöttünk, hogy megpróbáljuk használni az egyik ismertebb SMTP szolgáltatót, amelyek közül a Mailgun-ra esett a választás.

A Mailgun használata egyszerű volt, mindössze egy API-t kellett használni, amin keresztül tetszőleges e-mail címre tudunk tetszőleges, általunk birtokolt domainről emailt küldeni.

A domain megszerzése már valamennyivel bonyolultabb volt, de sikerült egy évre megszerezni ingyenesen a „tournament.ga” című, domaint, aminek felhasználásával képessé kellett volna válnunk az e-mailek elküldésének megkezdésére, bár előbb szükséges volt nem kevés konfigurációra. A domain ingyenes megszerzéséhez természetesen nagyban hozzájárult az, hogy a cím kialakítása miatt nem rendelkezik jelentős kereslettel.

Először is, a Cloudflare szolgáltatásait be kellett állítani, ahol lényegében szükséges volt bebizonyítani azt, hogy ténylegesen birtokoljuk a domaint, ami után a Mailgun által biztosított DNS rekordokat elhelyezve, majd a Mailgunt eszerint bekonfigurálva képessé váltunk végre az email-ek elküldésére.



23. ábra: Példa az elküldött értesítésekre.

```

public static void notifyUsers(Tournament tournament) throws UnirestException {

    List<Team> teams= new ArrayList<>();

    Logger logger = LoggerFactory.getLogger(NotificationServiceImpl.class);

    for (TournamentToTeams tournamentToTeams : tournament.getTeams()){
        teams.add(tournamentToTeams.getTeam());
    }

    for (Team team : teams) {
        String message;
        if (team.getId().equals(tournament.getVictorId())){
            message = "Congratulations, your teams has won the tournament: " + tournament.getTournamentName();
        }
        else
        {
            message = "The tournament: " + tournament.getTournamentName() + " you partook in has ended. GGWP!";
        }

        for (SiteUser currUser : team.getTeamMembers()) {

            HttpResponse<String>
                request = Unirest.post( url: "https://api.eu.mailqun.net/v3/mq.tournament.qa/messages")
                    .basicAuth( username: "api", API_KEY)
                    .queryString( name: "from", value: "notifier@tournament.ga")
                    .queryString( name: "to", currUser.getMail())
                    .queryString( name: "subject", value: "Tournament ended.")
                    .queryString( name: "text", message)
                    .asString();
            logger.info(request.getBody());

        }
    }
}

```

24. ábra: A szolgáltatás használata

Látható, hogy a szolgáltatás használata, és az azt kiszolgáló logika nem bonyolult.

Mindössze két egyszerű lépésből áll: a releváns csapatok összegyűjtése az adatbázisból, és az API hívásából.

Az adatbázisból való lekérés, mint látható, triviális ezért azt nem is szükséges tovább tárgyalni, viszont az API használata már kevésbé volt egyszerű. Az API használatához természetesen szükségünk volt, egy, a szolgáltatótól szerzett API kulcsra, amit miután megszereztünk, csakis környezeti változóként elérve használtunk, mert közvetlenül a kódba írva nem lenne biztonságos tárolni.

A lista megszerzése után, az API-t meghívjuk egy http kliensen keresztül, aminek a válaszát természetesen logoljuk, amiről később fog egy pár szó esni.

Sajnos az E-mail küldés az implementáció után problémákba ütközött, mivel egy Mailgun frissítés után hiba keletkezett a felhasználói felületben, ami lehetetlenné tette a havi maximum elküldött e-mailek limitálását, ami miatt az E-mail küldő része a weboldalnak jelenleg nem elérhető. Ennek a résznek az írásának az időpontjában a Mailgun még nem javította ki a hibát sajnos.

4.6 A Logolás

Minden komoly projektnek előbb-utóbb foglalkoznia kell a logolás kérdésével, ami alól ez a projekt sem volt kivétel.

A logolás fontos, mivel segítségével az esetleges hibák, anomáliák a működésben sokkal könnyebben felderíthetőek, ha azoknak van írott nyoma. Java platformon van ennek rengeteg sok megvalósítása, amelyek között lehet válogatni, ezért szerencsére nem kellett kézzel megvalósítani. A logolás megvalósítására példák: Logback, Apache Commons Logging, Tinylog, Log4J. Sajnos az SLF4J nem egy logoló keretrendszer, hanem csak egy interfész, ami tud használni sok logoló implementációt.

```
logger.info("RANK UPDATE STARTING: " + System.currentTimeMillis());
```

25. ábra: Példa a Logger használatára

Mivel a hírhedt Log4j biztonsági rés nem sokkal a projekt fejlesztésének elkezdése előtt derült ki nem sokkal [16], ezért az átfogó logolás kérdését egy kevés ideig napolni kellett, addig, amíg nem jött létre egy átfogó javító-csomag, amely minden Log4J biztonsági hibát javított, mivel sajnos sok más logoló könyvtár a Log4j-t használta valamilyen szinten.

A korábbi biztonsági rések ellenére azért esett a Log4J-re a választás, mert a hiba kiderülése után számtalan biztonsági kutató kezdte el átnézni a Log4J-t bármilyen, akár legapróbb biztonsági probléma után.

Ezek után a cél egyszerű volt, mindössze be kellett állítani a projekt által használt függőség-kezelő eszközt, amely jelen esetben a Maven volt, hogy csak is azt a specifikus verzióját töltsse le az artifactoryból a keretrendszernek, amiben már biztosan tudjuk, hogy javítva van a hiba, ezzel elkerülve a számtalan, súlyos biztonsági hibát, ami a Log4J nagyon sok 2.x verzióját szennyezte.

```
<dependency>
  <groupId>org.apache.logging.log4j</groupId>
  <artifactId>log4j-core</artifactId>
  <version>2.17.2</version>
</dependency>
```

26. ábra: A Maven konfigurálása

4.7 A biztonság kialakítása

Korábban szó esett róla, de részletesebben nem került tárgyalásra a biztonság témaköre. A projekt keretein belül, a biztonságot és a belépést a Spring Security használatával valósítottuk meg.

A Spring Security egy hozzáférés-ellenőrző és autentikációs keretrendszer [17]. Legnagyobb előnyei a könnyű használat, minimális konfiguráció szükséglete és a magas szintű személyre szabhatóság. A projekt keretein belül két fontos felelősséggel rendelkezik a Spring Security: A felhasználók regisztrálása és hitelesítése, és az erőforrás elérések szabályozása.

A felhasználók regisztrálása hasonlóan egyszerűen történik meg, pusztán egy külön erre kialakított API végponton keresztül, amiben, ha minden rendben van, az adatokkal titkosításra kerül a jelszó és az ahhoz tartozó hash. Minden további információt beleírunk az adatbázis megfelelő rekordjába.

A jelszavak titkosításához a Spring Security által nyújtott bcrypt titkosítási eljárásra esett a választás, mivel ez egy kifejezetten modern, biztonságos eljárás, főleg, mivel eleve képes „sót” használni a jelszavak titkosításánál, ezzel további védelmet nyújtva bizonyos típusú támadások ellen.

A felhasználó hitelesítéséhez három fontos dolgot volt szükség beállítani: a belépési végpontot, a végpont által használt formátumot, és a felhasználói információt szolgáltató egyedi UserDetailsService-osztályt.

Az egyszerűbb része természetesen a UserDetailsService beállítása volt. Pusztán arra volt szükség, hogy az adatbázisból lekértük az adott azonosítóval rendelkező felhasználó adatait, és azokat megfelelő formátumban visszaadtuk a Spring Security belső rendszerének.


```

@Override
public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {

    SiteUser siteUser = siteUserRepository.findSiteUserByUsername(username)
        .orElseThrow(() -> new ResourceNotFoundException("User not found"));

    boolean enabled = true;
    boolean accountNonExpired = true;
    boolean credentialsNonExpired = true;
    boolean accountNonLocked = true;

    return new User(siteUser.getUsername(), siteUser.getPassword(), enabled, accountNonExpired,
        credentialsNonExpired, accountNonLocked, new ArrayList<>());
}

```

27. ábra: A rövid userDetailsService

További beállításokat a SpringSecurityConfig osztályon keresztül voltunk képesek beállítani, ami egy Spring konfigurációs Bean. Ezen az osztályon keresztül nagyon sok dolgot be tudtunk állítani, mint például a Spring Security AuthenticationProvider-ét, amin keresztül lényegében megadtuk a használni kívánt UserDetailsService-t és a jelszó titkosítót, amiket így már képes használni a Spring.

```

@Override
protected void configure(HttpSecurity http) throws Exception {
    http.csrf().disable().authorizeRequests().ExpressionUrlAuthorizationConfigurer<...>.ExpressionInterceptUrlRegistry

        .antMatchers(
            HttpMethod.GET,
            ...antPatterns: "/index*", "/static/**", "/*.js", "/*.json", "/*.ico", "/register", "/api/users/register"
        ) ExpressionUrlAuthorizationConfigurer<...>.AuthorizedUrl
        .permitAll() ExpressionUrlAuthorizationConfigurer<...>.ExpressionInterceptUrlRegistry
        .anyRequest().authenticated()
            .and() HttpSecurity
        .formLogin().loginPage("/login") FormLoginConfigurer<HttpSecurity>
        .loginProcessingUrl("/api/login-process")
        .defaultSuccessUrl("/self-user").permitAll();
}

```

28. ábra: A konfiguráció

A 27. ábrán több dolog is beállításra kerül, amik sorrendben a következők:

- Egy sor erőforrás, amit mindenki elérhet.
- Azokon az erőforrásokon kívül mindent csak hitelesítés után lehessen elérni.
- A belépési adatok elvárt formátuma.
- A belépéshez szükséges URL-ek.

- A belépés utáni jogok.

Az első dolog, ami beállításra kerül, az `antMatchers()` hívásán keresztül, azok az erőforrások, amelyeket minden akár be nem lépett felhasználó is elérhet, ilyenek például a statikus erőforrások, vagy a regisztrációhoz szükséges frontend és backend API végpontok, hisz ha csak már eleve belépett felhasználó lenne képes új felhasználót regisztrálni, annak nem lenne túl sok haszna.

Ezt követően, az `anyRequest().authenticated()` metódus-lánc adja meg, hogy minden más erőforrás eléréséhez már hitelesítve kell legyen a kliens. Az ezt követő `and()` metódus hívás pusztán arra szolgál, hogy a konfiguráció különböző lépéseit összeláncoljuk.

A `formLogin()` metódus-hívás adja meg az elvárt formátumát a belépési adatoknak, ami a HTML form-okhoz hasonlóan kulcs-érték párokban küldi el.

A `loginPage()`, `loginProcessingUrl()`, és `defaultSuccessUrl()` metódusok nagyon hasonló dolgokat csinálnak: A `loginPage()` megadja, hogy milyen URL-t adjon vissza a Spring Security mikor valakitől belépést vár el, ez jelen esetben egy React oldal, amiről majd a következő fejezetben kerülnek bővebben tárgyalásra.

A `loginProcessingUrl()` megadja, hogy mely URL-re kell küldeni a form-ban tárolt adatokat, ennél az oknál fogva rendelkezik az `/api/` prefix-el. A `defaultSuccessUrl()` megadja, hogy belépés esetén melyik URL-re legyen irányítva a felhasználó, abban az esetben, ha belépés előtt nem próbált valamely más oldalt elérni, amely esetben arra az oldalra kerül irányításra.

4.8 A backend előkészítése a frontendre

Korábban is meg volt említve, de a backendnek feladata a frontend kiszolgálása, amihez a backend és frontend között megfelelő kapcsolatot volt szükséges kialakítani. Mivel a JavaScript és a Java a nevüket meghazudtolva teljesen különböző technológiák, ezért ez a feladat bonyolultabbnak bizonyult, mint amire az első felmérés alapján számítottunk.

Először arra volt szükség, hogy az NPM által fordított, optimalizált React fájlok egy olyan helyen legyenek, ahonnan könnyen elérhetőek és kitakaríthatóak újrafordítás esetén. Ebben volt hatalmas segítség a Maven Frontend Plugin, ami egy olyan Maven plugin, ami segít abban, hogy NPM fordításokat futtassunk akár közvetlen a Maven-t használó integrált fejlesztői környezetből. Külön előnye, hogy a plugin által használt NodeJS és NPM verziók lehetnek

különbözőek a globális verzióktól, ami kifejezetten előnyös, ha több projekten szükséges egyszerre dolgozni. (28. ábra)

Az így létrejött fájlokat természetesen érdemes még a megfelelő mappába másolni, ami jelen esetben A Java nyelv által használt Target mappa volt. Ez a mappa tartalmazza a fordított osztályokat, erőforrásokat, ezért természetes volt, hogy a React fájljai is ide fognak kerülni.

A fájlok automatikus másolását is egy hasznos pluginon keresztül valósítottuk meg, a Maven Resources Pluginon keresztül. Ezt a plugint jelent esetben arra használtuk, hogy a Package életciklus-fázis alatt futtassuk le a fájlok másolását a target mappa megfelelő részmappájába.

```
<plugin>
  <groupId>com.github.eirslett</groupId>
  <artifactId>frontend-maven-plugin</artifactId>
  <version>1.6</version>
  <configuration>
    <nodeVersion>v16.13.0</nodeVersion>
    <npmVersion>8.1.0</npmVersion>
    <workingDirectory>src/main/webapp</workingDirectory>
    <installDirectory>${project.build.directory}</installDirectory>
  </configuration>
  <executions>
    <execution>
      <id>install node and npm</id>
      <goals>
        <goal>install-node-and-npm</goal>
      </goals>
    </execution>
    <execution>
      <id>npm install</id>
      <goals>
        <goal>npm</goal>
      </goals>
    </execution>
    <execution>
      <id>npm run build</id>
      <goals>
        <goal>npm</goal>
      </goals>
      <configuration>
        <arguments>run build</arguments>
      </configuration>
    </execution>
  </executions>
</plugin>
```

29. ábra: Az NPM-Maven integráció

Természetesen még volt egy lépés hátra a frontend hozzákapcsolásához, hisz a fent ábrázolt rendszer pusztán elérhetővé teszi a szerver részére a fájlokat, nem adja meg azt, hogy azokkal mit kell kezdeni. Emiatt ki kellett alakítani egy specializált kontrollert, aminek a feladata csak és csakis az volt, hogy a frontend fájljait szolgálja ki megfelelő URL-lel rendelkező GET kérések számára. Amennyiben ugyan olyan módon kívántuk volna megoldani az irányítást, mint a többi kontrollernél, akkor rendkívül gyorsan több tucatnyi URL-t is kellett volna írni, ami fenntarthatósági szempontból nagyon nem lett volna előnyös. Szerencsére a controller-annotációk elfogadtak reguláris kifejezéseket is, amikkel nagyon jól lehetett mintákat illeszteni.

```
@Controller
public class ReactAppController {

    @RequestMapping(value = { "", "/{x:[\\w\\-]+}", "/{x:^(?!api$).*$/**/{y:[\\w\\-]+}" })
    public String getIndex(HttpServletRequest request) {
        return "/index.html";
    }
}
```

30. ábra: Kontroller és reguláris kifejezések.

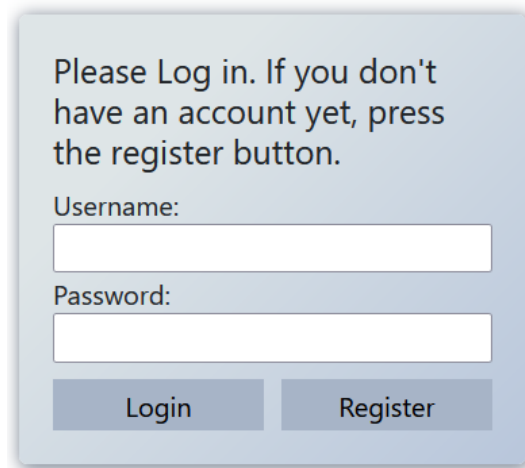
5. A kliensoldali réteg

A kliensoldali, vagyis frontend réteg, valamelyest alacsonyabb fontosságú volt az API mellett, de a megvalósítása nem volt egyszerűbb, mint a szerveroldalé. Miután a szerver elküldi a szükséges HTML fájlt, a kliensoldalon lévő React-router dönti el, hogy melyik oldalt mutassa éppen:

```
<BrowserRouter>
  <Routes>
    <Route path='/login*' element={<Login/>}/>
    <Route path='/register' element={<Register/>}/>
    <Route path='/headertest' element={<NavHeader/>}/>
    <Route path='/self-user' element={<SelfUser/>}/>
    <Route path='' element={<SelfUser/>}/>
    <Route path='/new-team' element={<CreateTeam/>}/>
    <Route path='/self-teams' element={<SelfTeams/>}/>
    <Route path='/one-team/:id' element={<OneTeam/>}/>
    <Route path='/teams' element={<AllTeams/>}/>
    <Route path='/create-tournament' element={<CreateTournament/>}/>
    <Route path='/all-tournaments' element={<AllTournaments/>}/>
    <Route path='/one-tournament/:id' element={<OneTournament/>}/>
  </Routes>
</BrowserRouter>
```

31. ábra: A React-router

Ahogy a 29. ábrán is látható, a React-router a böngésző által küldött URL alapján dönti el a megjelenítendő komponenst, ezzel lényegében teljesen kliensoldalivá vált a navigáció. Egyik, a React-router által visszaadott komponens a login komponens, ami különleges szereppel rendelkezik, hisz a Spring Security ettől várja a FormData formátumú adatokat az adott API végpontra.



Please Log in. If you don't have an account yet, press the register button.

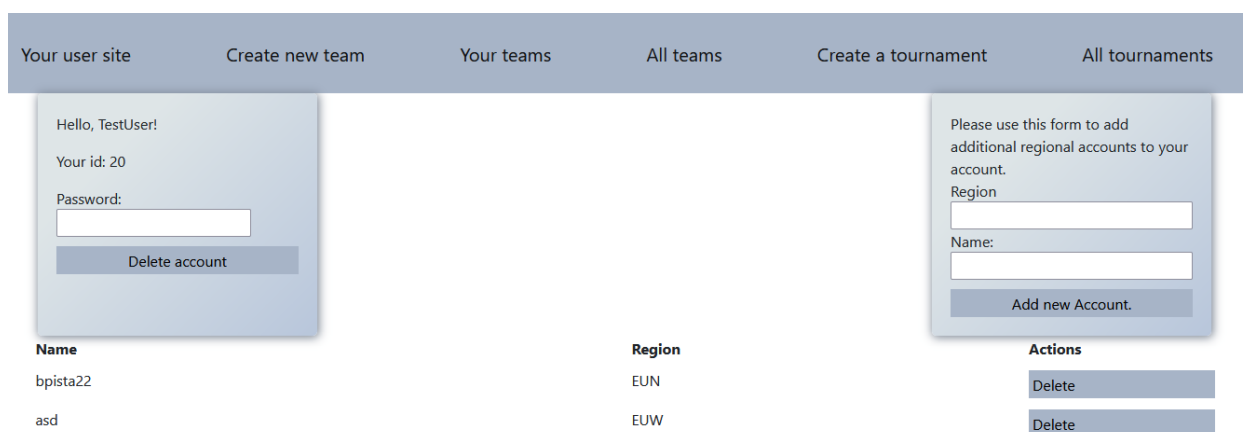
Username:

Password:

Login Register

32. ábra: A belépési képernyő

Természetesen, a React nagy előnye az komponensek újrahasználatossága [18], amit a projekt igyekezett kihasználni. Emiatt, a projekt keretein belül törekedtünk a konzisztens esztétikai megjelenés megvalósítására.



Navigation: Your user site | Create new team | Your teams | All teams | Create a tournament | All tournaments

Hello, TestUser!

Your id: 20

Password:

Delete account

Please use this form to add additional regional accounts to your account.

Region

Name:

Add new Account.

Name	Region	Actions
bpista22	EUN	Delete
asd	EUW	Delete

33. ábra: Példa képernyő

Mivel az alap React statikus fájllokból, erőforrásokból épül fel, vagyis nem szerveroldalon kerül összeállításra az adott HTML dokumentum, ezért azokat az adatokat valahogy el kellett kérni a szervertől miután az kiszolgált a React dokumentumokat. Érdeemes megjegyezni, hogy léteznek

szerveroldali renderelést támogató React-ból leszármaztatott könyvtárak, mint például a Next.js, de a projekt tervezésénél arra döntésre jutottunk, hogy erre jelen esetben nincs szükség.

```
fetch('/api/teams/remove',{
  method:'DELETE',
  headers: {
    'Accept': 'application/json',
    'Content-Type': 'application/json'
  },
  body: JSON.stringify(jsonBody),
}).then(response =>{
  response.json().then( json =>{
    if(response.ok){
      let updatedTeams = [...teams].filter(i => i.id !== id);
      let teamsTemp = teams;
      teamsTemp.regionalAccounts = updatedTeams;
      toastMessage.current = json;
      setTeams(updatedTeams);
    }
  })
})
```

34. ábra: A fetch API

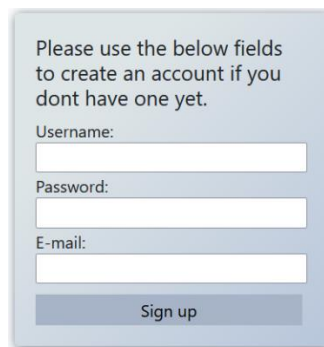
A 32. ábrán látható az adatok megszerzésének módja a frontenden, ami a széles körben használt és ismert Fetch API. A Fetch API egy interfészt biztosít az erőforrások hálózaton való eléréséhez. Előnye, hogy ez az API kifejezetten rugalmas és hatékony, ráadásul minden modern böngésző alaptól támogatja a használatát [19].

A példán látható HTTP hívás egy DELETE kérés, vagyis arra szolgál, hogy erőforrásokat eltávolítsunk. A képen látható, hogy az API képes minden a HTTP hívásokhoz szükséges részt kialakítani, és használni, mint például a „header” és „body” részek.

6. Az alkalmazás működés közben

Az alkalmazás kliensoldali része alapvetően a fejléc segítségével valósítja meg a navigálhatóságot. Az oldalak, képernyők jelentős részét el lehet érni ezen fejléc használatával közvetlenül. Bizonyos oldalak, mint például egy adott csapat oldala, csak közvetett úton, a fejlécről elérhető oldalakon található hivatkozásokon keresztül érhető el.

Az oldal felkeresése esetén, az első képernyő amire átirányítja a rendszer a böngészőt, az a 32. ábrán látható belépési űrlap. Amennyiben a felhasználó rendelkezik egy fiókkal, belépés után alapesetben a 33. ábrán látható oldalra kerül átirányításra, amennyiben nem, akkor viszont regisztráció szükséges. A regisztrációs oldalt 32. ábrán látható módon, egy átirányító gombon keresztül lehet elérni. A regisztrációhoz szükséges természetesen egy e-mail-cím, egy tetszőleges felhasználónév és egy jelszó, ami, ha sikeres az adatbázisban létrejön az új felhasználói fiók (35. ábra).



Please use the below fields
to create an account if you
dont have one yet.

Username:

Password:

E-mail:

Sign up

35. ábra: A regisztrációs-űrlap

Amikor a felhasználó sikeresen elnavigál a 33. ábrán látható oldalra, az ott lévő gombokkal, az oldal utasításainak segítségével akár ki is törölheti a felhasználói fiókját, hozzácsatolhatja a League of Legends fiókjait az alkalmazásban lévő fiókjához és ki is törölheti azokat tetszése szerint.

A navigációs fejlécről sorban az első elérhető oldal az úgynevezett „Create new Team” oldal, amely segítségével a felhasználó egy új csapatot tud létrehozni (36. ábra).

You can use the field below to create a new team.

Team name:

Sign up

36. ábra: Új csapat létrehozása

A fejlécről elérhető következő képernyő az úgynevezett „Your teams” képernyő, amin keresztül az éppen belépett felhasználó elérheti az összes, a fiókjához tartozó csapatot. Innen akár ki is tud lépni a csapatokból vagy elérheti az azokhoz tartozó információs oldalakat (37. ábra).

Your user site	Create new team	Your teams	All teams	Create a tournament	All tournaments
Id	Name	CreatorId	Actions		
57	pageTestTeam3	20	Enter	Leave	
1	testTeam	1	Enter	Leave	
58	algoTest1	20	Enter	Leave	
59	algoTest2	20	Enter	Leave	
60	algoTest3	20	Enter	Leave	
61	algoTest4	20	Enter	Leave	
62	algoTest5	20	Enter	Leave	
63	algoTest6	20	Enter	Leave	
64	algoTest7	20	Enter	Leave	
65	algoTest8	20	Enter	Leave	

37. ábra: A fiókhoz tartozó csapatok

A csapatokhoz tartozó információs oldalakon keresztül a belépett felhasználó, amennyiben a csapat alapítója, képes megváltoztatni a csapat nevét, egy verseny egyedi azonosítója beírása után képes csatlakozni az adott versenyhez, vagy akár ki is tudja vonni a csapatot azokból a versenyekből, amikhez már csatlakozott a csapat. Erről a képernyőről továbbá a csapat bármely tagja átirányíthatja a böngészőjét adott versenyek oldalára.

TeamId: 1
TeamName: testTeam
CreatorId: 1
New name:
Rename team

Please use this form to join tournaments.
Id:
Join

ID
1
2
20

Username
newestName
test2
TestUser

ID	Name	StartDateTime	Leave	Navigate
2	test2plus	2023-02-01 00:00	Leave	Navigate
1	Testt1	2022-02-01 00:00	Leave	Navigate

38. ábra: Adott csapat.

A következő, fejlécből közvetlen elérhető, oldal az úgynevezett „All teams” oldal. Ez az oldal nagyban hasonlít a korábban látott „Your Teams” oldalhoz, funkcionalistában is hasonló. A fő különbség a két oldal között az, hogy ezen képernyő használatával, csatlakozni lehet csapatokhoz, de azokból kilépni nem (38.ábra).

A fejlécről közvetlen elérhető következő képernyő már a bajnokságokhoz tartozik. Ez az oldal az úgynevezett „Create a Tournament” oldal. Ezen oldalon keresztül a felhasználó létre tud hozni tetszőleges mennyiségű saját bajnokságot, amikhez szükséges, hogy adjon meg egy címet vagy nevet, egy Riot régiót, és egy időpontot (39. ábra).

Please use this form to add additional regional accounts to your account.
Name:
Region:
2022-04-12 06:04

< April 2022 >

Su	Mo	Tu	We	Th	Fr	Sa
27	28	29	30	31	1	2
3	4	5	6	7	8	9
10	11	12	13	14	15	16
17	18	19	20	21	22	23
24	25	26	27	28	29	30
1	2	3	4	5	6	7

06:04

39. ábra: Egy bajnokság létrehozása

A navigációs fejlécből elérhető utolsó képernyő az úgynevezett „All-tournaments” képernyő. Ez a képernyő megjelenésben és funkcionalitásban is hasonlít az „All-teams” oldalhoz, mindkettőnek az az egyszerű fő feladata van, hogy emberileg olvasható formátumban jelenítsenek meg nagymennyiségű tömbös adatot és az, hogy az alapján való navigációt lehetővé tegyék. Az „All-tournaments” oldal fő feladata az, hogy a „Navigate” feliratú átirányító gomb segítségével segítsen abban, hogy a felhasználó eljusson az általa kívánt verseny oldalára (40. ábra).

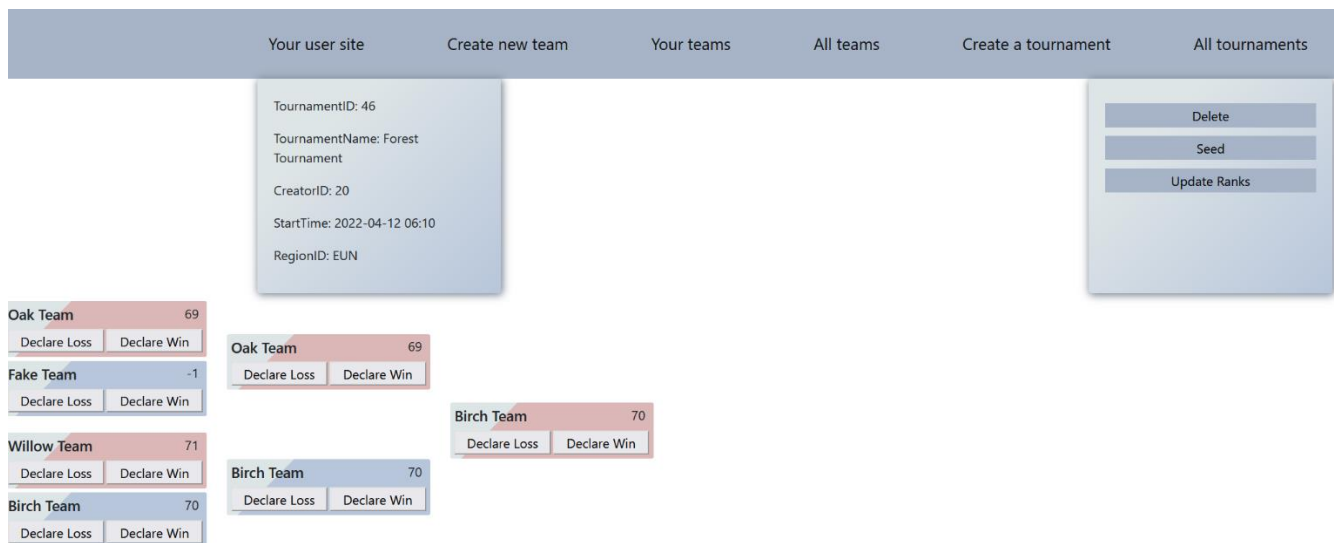
Your user site Create new team Your teams All teams Create a tournament All tournaments						
ID	Name	StartTime	VictorId	RegionId	CreatorId	Navigate
1	Teszt1	2022-02-01 00:00	2	EUW	1	Navigate
2	teszt2plus	2023-02-01 00:00	2	LAN	2	Navigate
4	seederTournament	2023-02-01 00:00		EUW	1	Navigate
8	securerTournament	2022-02-03 10:15	21	LAN	20	Navigate
10	valami	2020-08-19 10:00		EUN	20	Navigate
44	algoTestTournament	2022-02-23 00:00		EUN	20	Navigate

40. ábra: Az összes bajnokság képernyője

Az egyetlen oldal, ami nem került még bemutatásra, az az úgynevezett „One-tournament” oldal. Ezen oldalon keresztül történik meg egy-egy verseny lebonyolítása, annak eredményeinek nyomon követése továbbá még alapvető információkkal is szolgál.

Amennyiben a belépett felhasználó hozta létre a versenyt, amit az oldal ábrázol, képes azt törölni, a csapatokban lévő felhasználók készségszintjét megpróbálhatja elkérni a Riot API-tól és az elért készségszintek alapján még a verseny kezdőpozícióit is kioszthatja automatikusan. Ezeket a funkciókat sorban a „Delete”, „Update ranks” és „Seed” gombokkal lehet elérni.

Továbbá, minden csapat alatt a „Declare Loss” és „Declare Win” gombokkal lehet az adott csapatot győztessé vagy kiesetté nyilvánítani.



41. ábra: Egy bajnokság

6.1 Próba-bajnokság

Természetesen élesben is ki volt próbálva a rendszer. Az éles próbához igyekeztünk minél több csapatot kiállítani, mivel egy League of Legends csapat legalább 5 játékosból kell álljon, ezért csak 3 csapatot tudtunk összeállítani, ami még mindig 15 személy volt, akiknek sikeresen megszervezni egy-egy közös játszmát nem volt feltétlen könnyű feladat, még a rendszer segítségével sem.

Mivel a három nem egész hatványa a kettőnek, ezért sajnos az egyik csapatnak az első fordulóban nem volt játszmája, így azt úgy tekintette a rendszer, hogy automatikusan megnyerték, az ellenfelük helyére egy automatikusan vesztesnek tekintett csapatot helyezett be:



42. ábra: A kiinduló állapot

Szerencsére a meccseket nem volt szükséges közvetlenül egymás után lejátszani, ezért azok időbeli megszervezése nem volt túlságosan bonyolult. A következő két játszmat sikerült különösebb emberi probléma nélkül lejátszani, bár az éles próba alatt sikerült találni egy hibát az algoritmusban, ami a következő fordulokat kiszámolja, amit sikerült rövidesen javítani:

```
public static List<TournamentToTeams> calcNextRound2(List<TournamentToTeams> previousRound, Integer roundNumber) {
    List<TournamentToTeams> nextRound = new ArrayList<>();

    previousRound.sort(Comparator.comparing(TournamentToTeams::getPosition));
    if (previousRound.size() == 1) {
        return new ArrayList<>();
    }

    for (int i = 0; i < previousRound.size() - 1; i = i + 2) {
        if (previousRound.get(i).getEliminationRound() == null || (previousRound.get(i).getEliminationRound() >=
            roundNumber)) { //if the current team is not yet eliminated
            if (previousRound.get(i + 1).getEliminationRound() != null &&
                previousRound.get(i + 1).getEliminationRound() <= roundNumber) {
                nextRound.add(previousRound.get(i));
            }
        } else {
            nextRound.add(previousRound.get(i + 1));
        }
    }

    for (int i = 0; i < nextRound.size(); i++) {
        nextRound.get(i).setPosition(i + 1);
    }

    return nextRound;
}
```

43. ábra: A javított algoritmus

Az első és második játszma lejátszása között kijavításra került az adott fordulót kiszámoló algoritmus hibája, ami után rövidesen lejátszásra került az utolsó játszma, a döntő, aminek a végeredménye alább látható:



44. ábra: A végeredmény

7. Összefoglalás

A projekt során több, fontosabb megállapításra került sor. Voltak pozitív és negatív dolgok egyaránt.

Az első az, hogy a modern Java nyelv könnyen, és hatékonyan használható modern webalkalmazások készítésére, anélkül, hogy feláldoznánk a különböző JavaScript keretrendszerek által nyújtott előnyöket, mint például a Single Page Application (Egyoldalas Alkalmazás) lehetősége. Több, a modern Java által nyújtott eszköz is nagy hasznunkra vált, mint például a modularitás és az annotációk. Persze voltak hátrányai is a Java-nak, mint például a nagy mennyiségű osztály, ami létrejött, amik nem feltétlenül voltak jó hatással a rendszer átláthatóságára. Egyik kisebb hátrány továbbá még az inkonzisztens memória használat volt, amit a Java szemétyűjtése okozott, de szerencsére nem okozott problémát.

A hátrányai ellenére a Java nyelv kifejezetten hasznosnak, könnyen használhatónak bizonyult és a projekt végére egyértelműen kijelenthető, hogy jó választás volt a rendszerhez.

Másik fontosabb következtetés, amit megállapítottunk, az volt, hogy a Spring keretrendszer által nyújtott biztonsági, adatbázis-elérési, és webes kontroller szolgáltatások hatalmas segítségre lehetnek bármely Java alapú webes, vagy akár asztali, alkalmazás elkészítésében. Mivel az ismétlődő programrészek elkészítésének szükségletét minimalizálják, az alacsony szintű, gyakran más által már jobban megírt részek megírásának szükségletét meg eltörli, hisz például a Spring által biztosított kontrollerek nagyon jól működtek, bár ennek sajnos voltak hátrányai is. További hatalmas előnye volt a modern Spring-nek az annotációk használata, amiknek köszönhetően nem volt szükséges az XML-alapú konfigurációs fájlok használata, amelyek hiánya hatalmas mértékben könnyítette a projekt olvashatóságán.

Egyik ilyen hátrány volt a ReactJS megfelelő hozzákötése a Spring bizonyos részeihez, névlegesen a Spring Security-hez, nem volt egyszerű és a dokumentációja sok kívánni valót hagyott maga után. A Spring erősen ösztönzi a felhasználót arra, hogy minél több osztályba szedje szét a funkcionalitásokat, ezzel bár olvashatóbbá téve az osztályokat, a számukat is jelentősen megnöveli, ezzel hozzájárva a Java amúgy is nem elhanyagolható problémájához.

Természetesen, a Spring hátrányai ellenére is egy nagyon jó keretrendszernek bizonyult, rengeteg munkát spórolt meg a fejlesztés során, így bátran kijelenthető, hogy jó választás volt a projekthez.

Sajnos a projekt elkészülése során egy komolyabb, negatív következtetés is megállapításra került, mégpedig a külső fél által fenttartott API-k és könyvtárak használatának kockázatairól, problémáiról.

Legelőször a RIOT API-val való problémák keletkeztek, amik a projekt eredeti tervezetének megváltoztatását igényelték többek között. Mivel a „Tournament API” nem elérhető személyes használatra [20], ezért a projekt minden más részét jelentősen szükséges volt áttervezni.

Ezek után a Mailgun problémái léptek fel, ami miatt a projekt egy nem létfontosságú részét volt szükséges ideiglenesen nem működőképesnek tekinteni. Bár maga a korlátozás-funkció hiánya nem akadályozza meg az e-mailek elküldését, de a hiánya miatt egy esetleges rosszkaratú támadó hatalmas mennyiségű üzenet elküldésére veheti rá a szerveroldalt.

Természetesen, a hírhedt Log4J biztonsági hiba is komoly fejtörést váltott ki arról, hogy jó ötlet volt-e olyan nagyméretű külső könyvtárakat használni, amik funkcionalitását egy akár sokkal egyszerűbb, kisebb függőség is meg tudná oldani, aminek remélhetőleg kevesebb esélye van biztonsági réseket létrehozni.

Ezen problémák után azt a következtetést vontuk le, hogy bármely külső API, vagy külső könyvtár, használata előtt komolyan meg kell fontolni, hogy az adott projektnek ténylegesen szüksége van-e az adott API-ra vagy esetleg a külső könyvtár által nyújtott funkcionalitást máshogy nem valósítható meg.

Összességében, a projekt főbb funkcionalitásai elkészültek a kitűzött céloknak megfelelően. A létrejött problémák csak másodlagos és harmadlagos funkcionalitásokat befolyásoltak és ezek ellenére is elkészültek, ámbár az eredeti tervezettől eltérően. A projekt elkészítése során több hasznos következtetéssel lettünk gazdagabbak a Java és React alapú fullstack applikációkról, a Java ökoszisztémával kapcsolatban, valamint a külső erőforrások használatáról.

8. Irodalomjegyzék

- [1] Britannica, The Editors of Encyclopaedia. "Java". Encyclopedia Britannica, <https://www.britannica.com/technology/Java-computer-programming-language>
(Utolsó hozzáférés: 2022 március 31.)
- [2] Schönig, H. (2020). *Mastering PostgreSQL 13*. Birmingham, UK: Packt Publishing.
- [3] Giordano C. (2019). *Architecting Petabyte-Scale Analytics by Scaling out Postgres on Azure with the Citus Extension*,
<https://techcommunity.microsoft.com/t5/azure-database-for-postgresql/architecting-petabyte-scale-analytics-by-scaling-out-postgres-on/ba-p/969685>
(Utolsó hozzáférés. 2022 március 31.)
- [4] Fielding, R.T. (2000). *Architectural Styles and the Design of Network-based Software Architectures*, Irvine, CA: University of California
- [5] Berners-Lee, T., Fielding, R.T., Nielsen, H. F. (1996). *Hypertext Transfer Protocol – HTTP/1.0*, RFC Editor, RFC 1945, <https://datatracker.ietf.org/doc/html/rfc1945>
(Utolsó hozzáférés: 2022 április 19)
- [6] Iyer, V., Perry, E.H., Wright, B., Pfaeffle, T. és munkatársaik (2010). *Oracle Database, JDBC Developer's Guide and Reference*, Oracle
- [7] Bauer, C., King, G., Gregory, G. (2015). *Java Persistence with Hibernate 2nd Edition*, Shelter Island, NY, Manning Publications
- [8] Tuduse, C., Bauer, C., King, G., Gregory G. (2022) *Java Persistence with Spring Data and Hibernate*, Shelter Island, NY, Manning Publications
- [9] Martin, R.C., (2000). *Clean Architecture: A Craftsman's Guide to Software Structure and Design*, London, UK: Pearson Education
- [10] Bray, T. (2017), *The JavaScript Object Notation (JSON) Data Interchange Format*, RFC Editor, RFC 8259, <https://datatracker.ietf.org/doc/html/rfc8259>
(Utolsó hozzáférés: 2022 április 19)

- [11] Fieldling, R.T, és munkatársai, (1999). *Hypertext Transfer Protocol – HTTP/1.1*, RFC Editor, RFC 2616, <https://datatracker.ietf.org/doc/html/rfc2616>
(Utolsó hozzáférés: 2022 április 19)
- [12] Fowler M., (2004), *Inversion of Control Containers and the Dependency Injection pattern*, <https://martinfowler.com/articles/injection.html>
(Utolsó hozzáférés: 2022 április 1)
- [13] Gosling, J., Joy, B., Steele, G., Bracha, G., Buckley, A. (2015). *The Java® Language Specification Java SE 8 Edition*, Oracle
- [14] Fieldling, R.T., Reschke, J., (2014). *Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content*, RFC Editor, RFC 723, <https://datatracker.ietf.org/doc/html/rfc7231>
(Utolsó hozzáférés: 2022 április 19)
- [15] Johnson, R. és munkatársai, (2011), *Spring Reference Documentation*,
<https://docs.spring.io/spring-framework/docs/3.0.x/spring-framework-reference/pdf/spring-framework-reference.pdf>
(Utolsó hozzáférés. 2022 április 12.)
- [16] Palmer, D., (2021). *Security Warning: New Zero-Day in the Log4j Java Library is Already Being Exploited*, ZDNet
<https://www.zdnet.com/article/security-warning-new-zero-day-in-the-log4j-java-library-is-already-being-exploited/>
(Utolsó hozzáférés: 2022 április 12.)
- [17] Spilcă, L. (2022). *Spring Security in Action*, Shelter Island, NY, Manning Publications
- [18] Stefanov, S., (2016). *React Up & Running Building Web Applications*, Sebastopol, CA, O'Reilly Media
- [19] Mozilla Developer Network Webes Dokumentáció,
https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API
(Utolsó hozzáférés: 2022 április 1)
- [20] Riot Developer Portal,
<https://developer.riotgames.com/docs/portal>
(Utolsó hozzáférés: 2022 április 1)

9. Függelék

A projekt elérhető az alábbi tárolóban: <https://github.com/Errelian/Szakdolgozat>

10. Köszönetnyilvánítás

Ezúton szeretnék köszönetet mondani témavezetőmnek, Tiba Attila tanár úrnak a segítségéért, jó tanácsaiért, amelyek nagyban hozzájárultak szakdolgozatom elkészüléséhez.

Továbbá szeretnék köszönetet mondani a családomnak, akiknek a türelme és támogatása nélkülözhetetlen volt.