

DEBRECENI EGYETEM  
INFORMATIKAI KAR

SZAKDOLGOZAT

Fullstack labdarúgó tippjáték készítés Spring Boot és React keretrendszerekkel

Témavezető:  
Major Sándor Roland  
Tanársegéd

Készítette:  
Szarvas Péter  
Programtervező informatikus

DEBRECEN

2025

# Tartalomjegyzék

1. Bevezetés .....	3
2. Felhasznált eszközök .....	5
2.1 Git és GitHub .....	5
2.2 Docker és Google Cloud Platform.....	6
2.3 MongoDB NoSQL adatbázisrendszer.....	7
2.4 Java és Spring Boot keretrendszer .....	9
2.5 Spring Dotenv és Lombok .....	10
2.6 TypeScript és React .....	11
2.7 Tailwind CSS, Axios, Zustand, Formik, Yup és React Router .....	11
2.8 LLM használata unit tesztekhez és Javadoc dokumentációhoz.....	12
2.9 Postman és Bruno a funkcionális teszteléshez.....	12
3. Kivitelezési folyamatok .....	13
3.1 Az adatbázis és az adatok tárolásának kivitelezése külső API-ból.....	13
3.2 A felhasználókezelés kivitelezése .....	18
3.3 A REST végpontok és szervizeik létrehozása.....	22
3.4 A frontend elkészítése .....	27
3.5 Az elkészült projekt publikálása Google Cloud Platformon keresztül .....	30
3.6 Használati útmutató .....	33
4. Összefoglalás .....	34
5. Irodalomjegyzék .....	37
6. Köszönetnyilvánítás .....	40

1. ábra – docker-compose.yaml .....	6
2. ábra – A Google Cloud konzol ablaka, azon belül a virtuális gépek menüpont .....	7
3. ábra – Az adatbázis diagramja .....	8
4. ábra - A Mongo Express felülete .....	9
5. ábra - Aombok annotációinak használata a Competition modellnél .....	10
6. ábra - A WebClient alap konfigurációja, az API által elvárt egyedi token fejléccel.....	14
7. ábra - Az ütemező osztály, a havi és 15 percenkénti adatgyűjtéssel.....	15
8. ábra - A DatabaseSequence modell és a SequenceGeneratorService osztály.....	16
9. ábra - A játékosok elmentésére szolgáló metódus .....	17
10. ábra - Az egyedi szekvencia használata a MatchScoreBet modellben .....	17
11. ábra - A WebClient alkalmazása az adatok lekérésére.....	18
12. ábra - A WebSecurityConfig osztály securityFilterChain metódusa, amely a HTTP kérések jogosultsági szabályait definiálja.....	19
13. ábra - A JwtAuthenticationFilter osztály doFilterInternal metódusa.....	20
14. ábra - Az AuthService osztály register metódusa .....	22
15. ábra - Az autentikációs osztály regisztrációs végpontjának metódusa .....	23
16. ábra - A RegisterRequestDTO Jakarta Validation validációval .....	23
17. A FootballController osztály fogadással kapcsolatos végpontjai .....	24
19. ábra - A FootballService osztály cancelMatchScoreBet metódusának részlete (a try-catch blokkot nem tartalmazza) .....	25
20. ábra - A havi ranglétráért felelős metódusai a LeaderboardService osztálynak .....	26
21. ábra - A Login.tsx, ami megjeleníti a LoginForm.tsx-et .....	27
22. ábra - Főoldal a bejelentkezett felhasználóknak.....	28
23. ábra - A táblázat adatokat tartalmazó részének kódja.....	29
24. ábra - A profil oldal.....	29
25. ábra - A létrehozott virtuális gép technikai adatai .....	30
26. ábra - A böngészőalapú SSH kliens.....	31
27. ábra - Az nginx.conf fájl a NANO eszközzel megjelenítve .....	32

## 1. Bevezetés

Az informatika és a játéktervezés rohamos fejlődése az elmúlt évtizedekben alapjaiban változtatta meg a szórakoztatóipart, lehetőséget teremtve a klasszikus játékok digitalizálására és kreatív újragondolására. A sportfogadás és a tippelős játékok sem maradtak ki ebből a folyamatból, hiszen a modern technológiák – például adatbázisok, backend és frontend keretrendszerek – segítségével új szintre emelhetők ezek az élmények. Számos weboldal létezik ma már, amely sporteseményekhez kapcsolódó tippelési lehetőségeket kínál (pl. TippmixPRO [1]), ahol a felhasználók valós időben követhetik az eredményeket és versenghetnek egymással.

Szakdolgozatom témájaként egy focimeccs-eredmény tippelős weboldal megvalósítását választottam, amely nem csupán a hagyományos tippjátékok digitális adaptációja, hanem egyedi funkciókkal bővített, interaktív élményt nyújtó platform. A projekt egy olyan online játékot hoz létre, amely ötvözi a focimeccsek kiszámíthatatlan izgalmát és a stratégiai gondolkodást, miközben a modern webfejlesztési technológiák erejét kihasználva közösségi és kompetitív elemekkel gazdagítja a játékot.

A „GoalRush” nevű játékom legfontosabb újítása a pontozási és ranglista-rendszer. A felhasználók a valós focimeccsek eredményeire tippelhetnek, és pontokat szerezhetnek az alábbiak szerint: egy tökéletesen eltalált eredmény (gólszám és nyertes csapat) 3 pontot ér, a nyertes csapat helyes megtippelése (vagy döntetlennél a döntetlen ténye, de eltérő gólszámmal) 1 pontot ad, minden egyéb esetben pedig 0 pont jár. A játék ingyenes, a pontok csupán a versengést és a ranglétrákon való előrehaladást szolgálják. A rendszer heti és havi ranglétrákat kínál, így a játékosok különböző időtávokon mérhetik össze tudásukat és szerencséjüket. A felhasználók regisztrációval saját profilt hozhatnak létre, amelyen követhetik teljesítményüket és eredményeiket.

A projekt relevanciája több szempontból is kiemelkedő. Egyrészt a focirajongók széles közönségét célozza meg, miközben a tippelés stratégiai mélysége gondolkodásra és elemzésre ösztönzi a játékosokat – mindezt ingyen, kockázat nélkül tehetik meg, szemben a legtöbb sportfogadási oldallal, ahol pénz vagy digitális kredit szükséges egy tipp leadásához. A fiatalabb generáció számára, akik a videójátékok dinamizmusához szoktak, a ranglétrák és a közösségi versengés izgalmasabbá teszi az élményt. Másrészt fejlesztési szempontból is jelentős, hiszen bemutatja, hogyan lehet valós idejű adatokat integrálni egy interaktív játékba modern technológiák – például a MongoDB adatbázis [2], a Spring Boot backend [3] és a React frontend [4] – felhasználásával. Harmadrészt a projekt demonstrálja ezeknek az eszközöknek az alkalmasságát skálázható, felhasználóbarát webalkalmazások készítésére.

A témaválasztásom személyes érdeklődésemből fakad: gyermekkorom óta rajongok a fociért, és mindig is kíváncsi voltam, hogyan lehet digitális platformokon keresztül új élményeket teremteni. A konkrét ötletet a témavezetőmmel közösen pontosítottuk, így döntöttem a Java alapú Spring Boot keretrendszer, a MongoDB NoSQL adatbázis és a React frontend kombinációja mellett. A MongoDB rugalmassága lehetővé tette a felhasználói adatok és meccseredmények hatékony kezelését, a Spring Boot stabil backendet biztosított, a React pedig modern, reszponzív felületet kínált a játékosok számára.

A jövőben szeretném továbbfejleszteni a projektet. Terveim között szerepel egy valós idejű értesítési rendszer bevezetése, amely figyelmezteti a játékosokat a közelgő meccsekre, valamint egy baráti liga funkció, ahol a felhasználók saját csoportokat hozhatnak létre. Emellett a pontozási rendszert is bővíteném, például bónuszpontokkal a különösen nehéz meccsek helyes megtippeléséért. A felhasználói felületet is tovább csiszolnám, mivel a fejlesztés során a backendre helyeztem a nagyobb hangsúlyt, így a frontend terén látok még lehetőséget a finomításra. Hosszabb távon az adatszerzést is optimalizálnám, például a külső API-k helyett saját webscraper megoldással, hogy még nagyobb kontrollom legyen az adatok felett. A fejlesztési folyamat során felmerült kihívások megoldása sok tanulást és kreativitást igényelt, amit a jövőben tovább kamatoztatnék a projekt tökéletesítésére.

## 2. Felhasznált eszközök

### 2.1 Git és GitHub

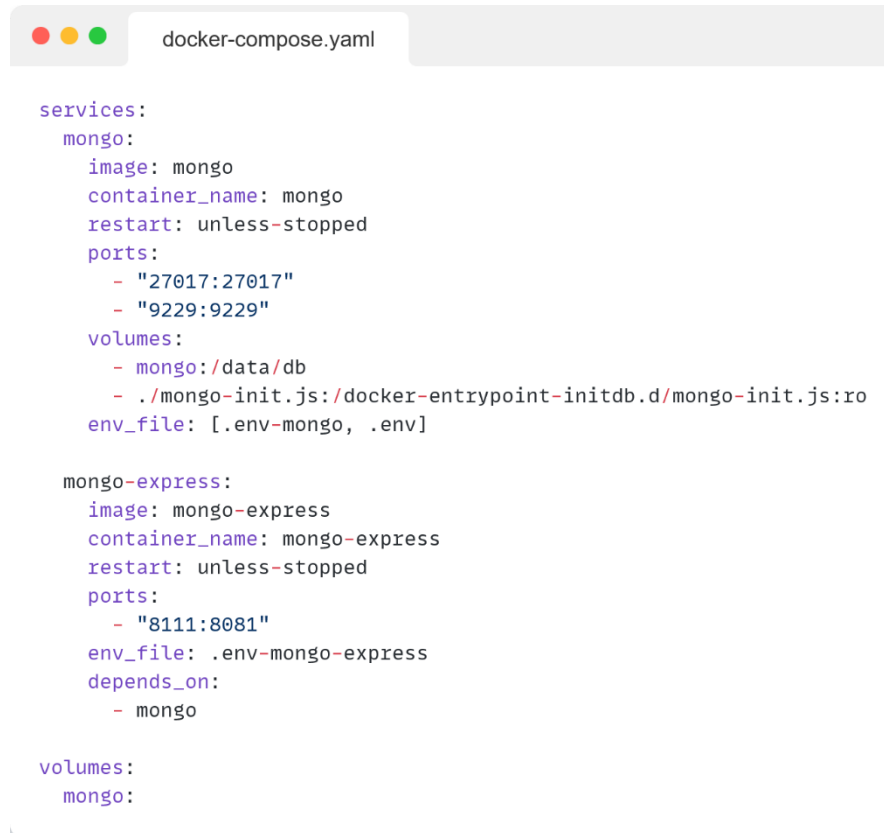
A projekt verziókezeléséhez a Git [5] rendszert választottam, amely egy nyílt forráskódú, Linus Torvalds által 2005-ben létrehozott elosztott verziókezelő rendszer (Version Control System, VCS). A Git eredetileg a Linux kernel fejlesztéséhez készült, hogy hatékonyan támogassa a kódváltozások nyomon követését és a verziók kezelését. A rendszer lehetővé teszi a fejlesztők számára, hogy rugalmasan és megbízhatóan kezeljék a projekt különböző állapotait, akár egyedül, akár csapatban dolgoznak.

A verziókezelő rendszerek használata mára elengedhetlenné vált a szoftverfejlesztési projektekben. A Git számos előnnyel jár: lehetővé teszi a korábbi verziókhoz való visszatérést hibák esetén, támogatja a párhuzamos fejlesztési ágak (branch-ek) kezelését, amelyeket később egyszerűen össze lehet fűsülni (merge). Emellett alkalmas kiadások (release-ek) kezelésére, legyen szó alfa-, béta- vagy végleges verziókról, illetve főbb verziószámú frissítésekről vagy kisebb hibajavító csomagokról.

A projekt forráskódja a GitHub [6] platformon érhető el (<https://github.com/Sz4rv4s/SZAKDOLGOZAT>). A GitHub egy Microsoft által üzemeltetett, felhőalapú szolgáltatás, amely nemcsak a Git-tárhelyet biztosítja, hanem számos további funkciót is kínál, például hibajegyek (issue-k) kezelését, kollaborációs eszközöket, valamint CI/CD (Continuous Integration/Continuous Deployment) folyamatok támogatását. Ezek az eszközök jelentősen megkönnyítették a projekt fejlesztési és karbantartási folyamatait.

## 2.2 Docker és Google Cloud Platform

A fejlesztés során a Docker [7] konténerizációs platformot használtam a MongoDB adatbázis és az ahhoz kapcsolódó, Express.js alapú webes adminisztrációs felület futtatására. A Docker választása azért volt előnyös, mert a projektet több különböző számítógépen fejlesztettem, így szükségem volt egy konzisztens, könnyen telepíthető és módosítható környezetre mind a fejlesztési, mind a tesztelési fázisokban. A konténerizáció lehetővé tette, hogy az alkalmazás függőségei és konfigurációi egységesek maradjanak, függetlenül a futtató környezettől.



```
services:
  mongo:
    image: mongo
    container_name: mongo
    restart: unless-stopped
    ports:
      - "27017:27017"
      - "9229:9229"
    volumes:
      - mongo:/data/db
      - ./mongo-init.js:/docker-entrypoint-initdb.d/mongo-init.js:ro
    env_file: [.env-mongo, .env]

  mongo-express:
    image: mongo-express
    container_name: mongo-express
    restart: unless-stopped
    ports:
      - "8111:8081"
    env_file: .env-mongo-express
    depends_on:
      - mongo

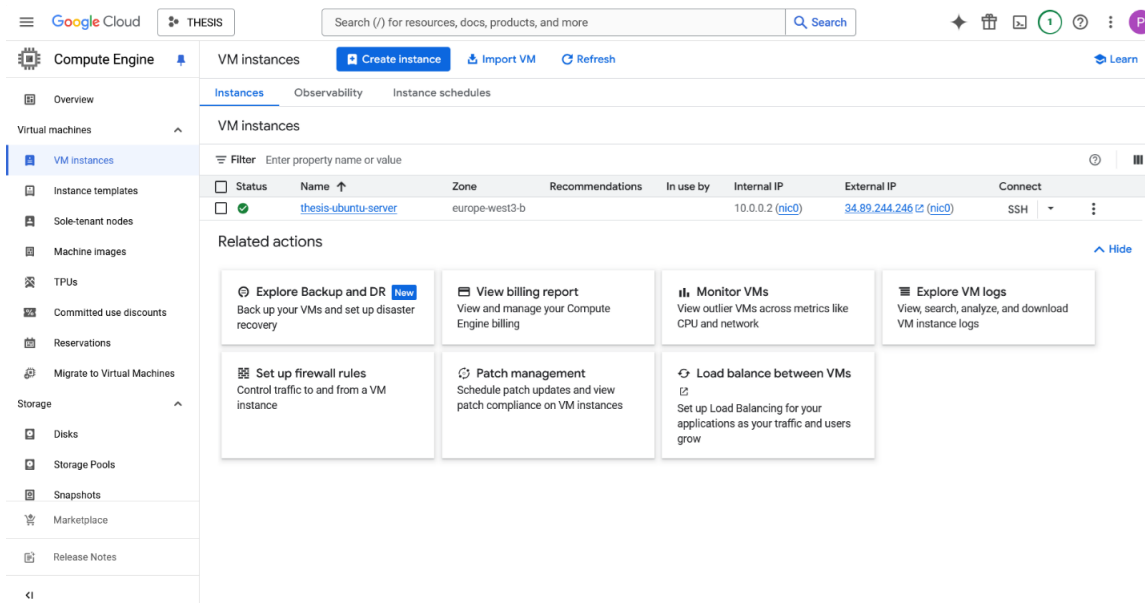
volumes:
  mongo:
```

*1. ábra – docker-compose.yml*

A Docker Compose eszközt alkalmaztam a konténerek kezelésére. Ez egy YAML formátumú leírófájl segítségével definiálja a futtatandó szervizeket, azok portjait, indulási sorrendjét, függőségeit, valamint támogatja az egészségügyi ellenőrzések (health check-ek) beállítását. A Docker Compose használata jelentősen leegyszerűsítette a többkonténeres környezet kezelését és a fejlesztési folyamatok automatizálását.

A projekt éles környezetben való futtatásához a Google Cloud Platform (GCP) [8] szolgáltatásait vettem igénybe. A GCP ingyenes próbaidőszakának kihasználásával egy Ubuntu 24.04 LTS operációs rendszert futtató virtuális gépet hoztam létre, amelyen a backend és a frontend alkalmazások futnak. Az éles környezetet egy NGINX fordított proxyval, NO-IP dinamikus DNS-sel és Let's Encrypt SSL-tanúsítvánnyal egészítettem ki, biztosítva a biztonságos és stabil

hozzáferést. A GCP rugalmassága és skálázhatósága lehetővé tette, hogy a projektet hatékonyan és költségkímélő módon üzemeltessem.

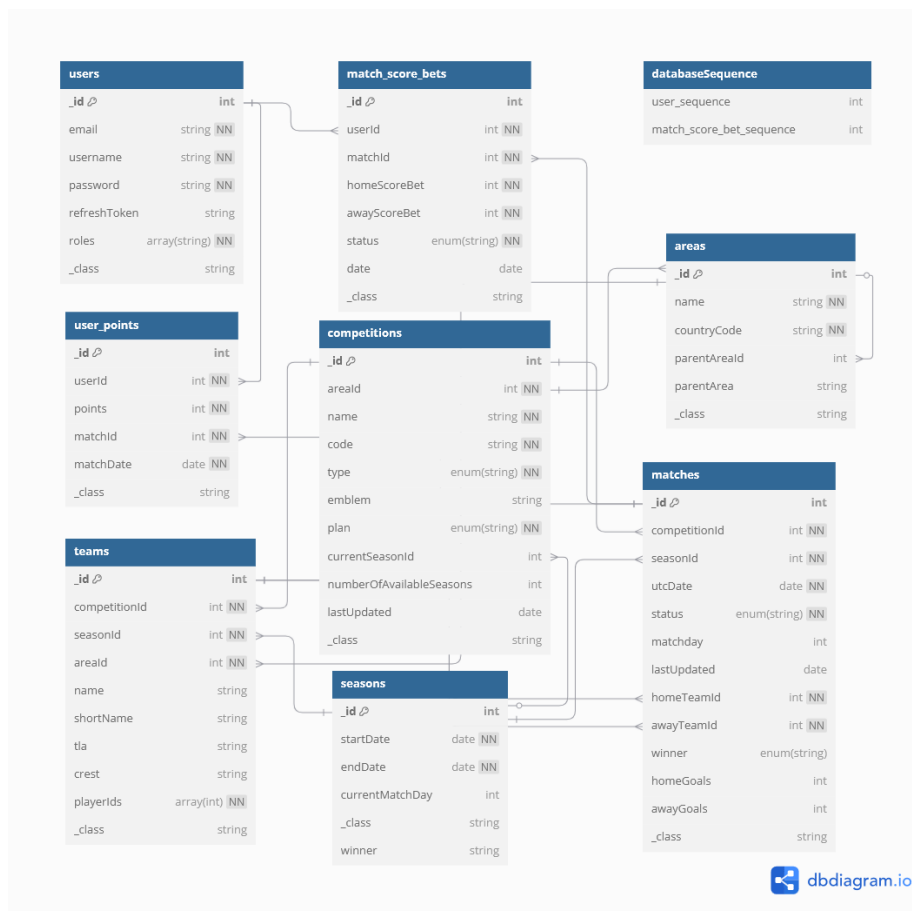


2. ábra – A Google Cloud konzol ablaka, azon belül a virtuális gépek menüpont

## 2.3 MongoDB NoSQL adatbázisrendszer

A projekt adatkezeléséhez a MongoDB NoSQL adatbázisrendszert [2] választottam. A MongoDB egy nyílt forráskódú, dokumentumorientált adatbázis, amely JSON-szerű dokumentumokban tárolja az adatokat, lehetővé téve a séma nélküli, rugalmas adatmodellezést. Ez a rugalmasság előnyös volt a projekt során, mivel a felhasználói profilok, tippek és meccseredmények eltérő struktúrájúak lehettek. Annak ellenére, hogy a MongoDB nem igényel előre definiált sémát, készítettem egy init scriptet, amely sémadeklarációkat és validációkat hozott létre a konzisztencia érdekében. Ezáltal biztosítottam, hogy az adatok nagyjából azonos formátumúak legyenek, megkönnyítve az adatfeldolgozást és a lekérdezéseket.





3. ábra – Az adatbázis diagramja

A MongoDB-t egy Express.js alapú adminisztrációs felülettel is kiegészítettem, amely a Node.js környezetben futott. Az Express.js [9] egy könnyű és rugalmas webes keretrendszer, amelyet az adatbázis kezelésére és egyszerűbb műveletek (pl. adatok manuális bevitele, ellenőrzése) elvégzésére használtam a fejlesztési fázisban. Ez a felület lehetővé tette, hogy gyorsan teszteljem az adatbázis működését és az adatstruktúrákat, mielőtt a Spring Boot backend teljes mértékben átvette volna az adatkezelést.

A MongoDB nagy teljesítménye és horizontális skálázhatósága ideálissá tette a projekt számára. A felhasználók adatait (pl. név, pontszámok), a tippeket és a mérkőzések eredményeit tároltam benne, amelyeket a backend hatékonyan kezelt. A MongoDB-t Docker konténerben futtattam, így a telepítés és a konfiguráció egységes maradt a különböző környezetekben.

## Viewing Database: footballdb

Collections					Collection Name	+ Create collection
View	Export	[JSON]	Import	areas	Del	
View	Export	[JSON]	Import	competitions	Del	
View	Export	[JSON]	Import	databaseSequence	Del	
View	Export	[JSON]	Import	match_score_bets	Del	
View	Export	[JSON]	Import	matches	Del	
View	Export	[JSON]	Import	players	Del	
View	Export	[JSON]	Import	seasons	Del	
View	Export	[JSON]	Import	teams	Del	
View	Export	[JSON]	Import	user_points	Del	
View	Export	[JSON]	Import	users	Del	

4. ábra - A Mongo Express felülete

## 2.4 Java és Spring Boot keretrendszer

A backend fejlesztéséhez a Java programozási nyelvet [10] és a Spring Boot keretrendszert használtam. A projekt a Java 21-es verziójával készült, amely 2023-ban jelent meg mint hosszú távú támogatású (LTS) kiadás [11]. A Java 21 számos újdonságot hozott, például a virtuális szálak (Virtual Threads) teljes körű támogatását a Project Loom részeként, ami jelentősen javítja a párhuzamos műveletek hatékonyságát. Emellett a rekordok (Records) és a mintaegyeztetés (Pattern Matching) továbbfejlesztései egyszerűbbé és olvashatóbbá tették a kódot, különösen az adatkezelő osztályok esetében. A Java 21 választása a projekt jövőbiztosságát és a modern funkciók kihasználását célozta.

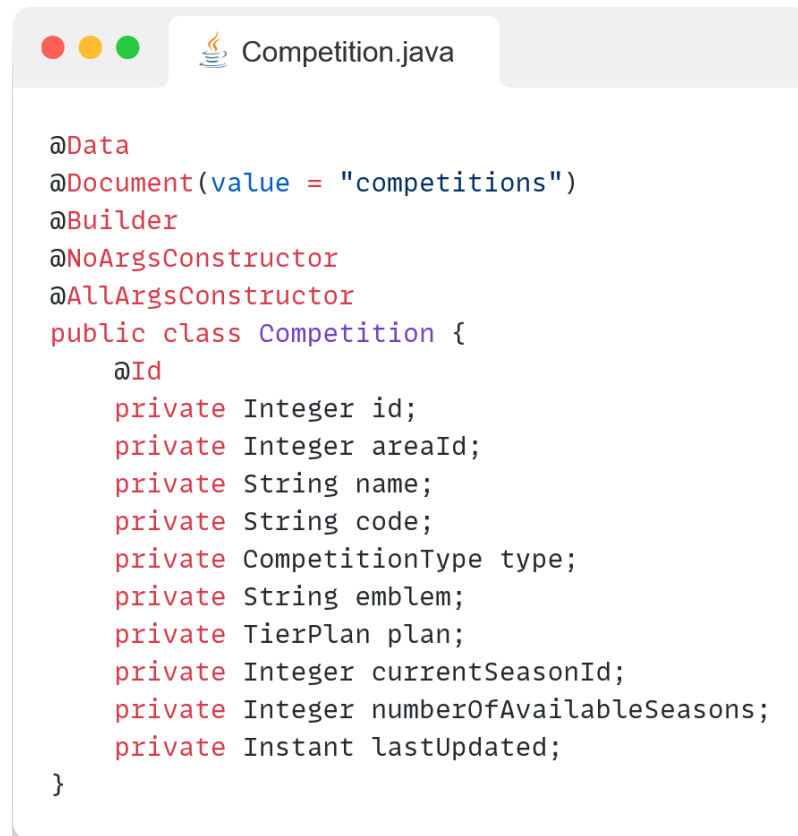
A Java futtatókörnyezetként az Eclipse Foundation által fejlesztett Temurin disztribúciót [12] preferáltam, amelyet az Adoptium projekt részeként terjesztenek. A Temurin egy nyílt forráskódú, ingyenesen elérhető OpenJDK implementáció, amely kiváló teljesítményt és stabilitást kínál. Előnyben részesítettem más disztribúciókkal (pl. Oracle JDK) szemben, mert közösségi támogatottsága és licencezési rugalmassága jobban illeszkedik a projekt nyílt forráskódú

szellemiségéhez. A Temurin használata biztosította, hogy a fejlesztés és az éles környezet konzisztens legyen, különösen a Docker és a Google Cloud Platform integrációja során.

A Spring Bootot a REST API implementálására használtam, amely a frontend és a MongoDB közötti kommunikációt biztosította. A Spring WebClient segítségével külső API-ból gyűjtöttem be a meccseredményeket, amelyeket a MongoDB-ben tároltam el. Az így feldolgozott adatokat a REST API-n keresztül „konyhakészen” szolgáltam ki a frontend számára. A Spring Data MongoDB könyvtár zökkenőmentes integrációt biztosított az adatbázissal, míg a keretrendszer függőséginjektálása és biztonsági moduljai (Spring Security) növelték a kód hatékonyságát és olvashatóságát.

## 2.5 Spring Dotenv és Lombok

A környezeti változók kezelésére a Spring Dotenv függőséget [13] alkalmaztam. Ez a könyvtár lehetővé tette, hogy az érzékeny adatokat (pl. adatbázis hitelesítő adatok) egy .env fájlban tároljam, különválasztva a forráskódtól, ami javította a biztonságot és a konfiguráció rugalmasságát.



```
@Data
@Document(value = "competitions")
@Builder
@NoArgsConstructor
@AllArgsConstructor
public class Competition {
    @Id
    private Integer id;
    private Integer areaId;
    private String name;
    private String code;
    private CompetitionType type;
    private String emblem;
    private TierPlan plan;
    private Integer currentSeasonId;
    private Integer numberOfAvailableSeasons;
    private Instant lastUpdated;
}
```

5. ábra - Aombok annotációinak használata a Competition modellnél

A kód olvashatóságának növelésére a Lombok könyvtárat [14] használtam, amely annotációkkal automatikusan generál boilerplate kódot. A projektben kizárólag a @Builder, @Data és konstruktor annotációkat alkalmaztam, például a felhasználói és meccseredmény entitások

definiálásához. Ez csökkentette a felesleges kódmennyiséget, így a logikai implementációkra fókuszálhattam.

## 2.6 TypeScript és React

A frontend fejlesztéséhez a TypeScript nyelvet [15] és a React könyvtárat [4] választottam. A TypeScript statikus típusellenőrzése növelte a kód megbízhatóságát, különösen az API-hívások és az állapotkezelés során. A React komponensalapú felépítése és virtuális DOM-ja gyors és rezponzív felületet biztosított a tippelési felület, a ranglisták és a felhasználói profilok megjelenítéséhez. A TypeScript és a React kombinációja modern és skálázható frontend fejlesztést tett lehetővé.

## 2.7 Tailwind CSS, Axios, Zustand, Formik, Yup és React Router

A felhasználói felület stílusának kialakításához a Tailwind CSS-t [16] használtam, amely egy utility-first CSS keretrendszer. A Tailwind CSS lehetővé tette, hogy előre definiált segédosztályok (pl. flex, p-4, bg-blue-500) használatával gyorsan és hatékonyan építsem fel a dizájnt anélkül, hogy egyedi CSS fájlokat kellett volna írnom. A projekt során az alapvető stílusozásra fókuszáltam, például a gombok, táblázatok és űrlapok megjelenésére, így a Tailwind alapértelmezett stílusai jelentősen felgyorsították a fejlesztést, és egységes vizuális megjelenést biztosítottak az alkalmazásnak.

Az API-hívások kezelésére az Axios könyvtárat [17] alkalmaztam, amely egy Promise-alapú HTTP kliens, és leegyszerűsítette a backenddel való kommunikációt. Az Axios segítségével valósítottam meg a tippek beküldését és a meccseredmények lekérdezését a Spring Boot REST API-jától. Az Axios interceptorok használatával automatizáltam a hitelesítési tokenek kezelését: a kérések előtt ellenőriztem a token érvényességét, és ha szükséges volt, automatikusan megújítottam azt egy külön API-hívással, így biztosítva a folyamatos hitelesítést a felhasználók számára. Ez a megoldás javította az alkalmazás stabilitását, és csökkentette a manuális hibakezelés szükségességét.

Az állapotkezeléshez a Zustandot [18] választottam, amely egy könnyű, Redux-szerű globális állapotkezelő könyvtár. A Zustand egyszerűsége miatt ideális volt a projekthez, mivel lehetővé tette a globális állapotok hatékony kezelését anélkül, hogy bonyolult boilerplate kódra lett volna szükség. A projektben egy dedikált store-ban tároltam a bejelentkezett felhasználó adatait, például a nevét és a hitelesítési állapotát, így a különböző komponensek könnyen hozzáférhettek ezekhez az információkhoz, például a navigáció során a bejelentkezési státusz ellenőrzéséhez.

Az űrlapok kezelésére a Formikot [19] és a Yup [20] validációs sémákat használtam. A Formik leegyszerűsítette az űrlapok állapotkezelését és validációját React környezetben, míg a Yup segítségével definiáltam a validációs szabályokat. Például a bejelentkezési és regisztrációs űrlapoknál a Yup biztosította, hogy az email mező formátuma helyes legyen, és a jelszó minimum 8 karakter hosszú legyen, így a felhasználók számára egyértelmű visszajelzést adhattam a hibákról (pl. „Érvénytelen email cím”). A Formik és a Yup együttes használata hozzájárult ahhoz, hogy az űrlapok kezelése pontos és felhasználóbarát legyen.

A navigáció kezelésére a React Routert [21] integráltam a projektbe. A React Router egy népszerű könyvtár a React alkalmazásokban történő útvonalkezeléshez, amely lehetővé tette, hogy

az alkalmazás különböző oldalai (pl. főoldal, ranglista, bejelentkezés) között a böngésző újratöltése nélkül navigálhassak. A React Router BrowserRouter komponensét használtam az útvonalak definiálására, például a /login útvonal a bejelentkezési oldalt, a /leaderboard pedig a ranglistát jelenítette meg. A navigáció során a useNavigate hookot alkalmaztam, amely lehetővé tette, hogy programozottan irányítsam át a felhasználókat, például sikeres bejelentkezés után a főoldalra. A React Router biztosította, hogy a navigáció zökkenőmentes legyen, és a felhasználók könnyen váltogathassanak az oldalak között, még ha a frontend fejlesztése során az egyszerűbb megoldásokra koncentráltam is.

Ezek az eszközök – a Tailwind CSS, Axios, Zustand, Formik, Yup és React Router – együttesen biztosították, hogy a frontend működőképes és a projekt céljainak megfelelő legyen. A Tailwind CSS az egyszerű stílusozást, az Axios a backend kommunikációt, a Zustand a felhasználói kontextus kezelését, a Formik és a Yup az űrlapok kezelését, míg a React Router a navigációt támogatta, így egy koherens frontend rendszert hozva létre.

## 2.8 LLM használata unit tesztekhez és Javadoc dokumentációhoz

A backend fejlesztése során a unit tesztek készítéséhez és a Javadoc dokumentáció generálásához egy mesterséges intelligencia alapú nyelvi modellt, a Grok 3-at használtam, amelyet az xAI fejlesztett [40]. A Grok 3 segítségével automatizáltam a Spring Boot alkalmazás unit tesztjeinek létrehozását, például a FootballService és AuthService osztályok metódusainak tesztelésére. A Grok 3 képes volt a kód kontextusának megértésére, és olyan teszteseteket generált, amelyek lefedték a tipikus forgatókönyveket, például a sikeres fogadás leadását (makeMatchScoreBet) vagy a felhasználó bejelentkeztetését (login). Emellett a Grok 3 a Javadoc kommentárokat is elkészítette, amelyek részletesen dokumentálták a metódusok működését, paramétereit és visszatérési értékeit, így javítva a kód olvashatóságát és fenntarthatóságát. Bár az LLM használata jelentősen felgyorsította a tesztelési és dokumentációs folyamatot, a generált tesztek nem mindig fedték le az összes edge case-t, például a ritkább hibakezelési forgatókönyveket, így a manuális tesztelés továbbra is nélkülözhetetlen maradt a projekt során.

## 2.9 Postman és Bruno a funkcionális teszteléshez

A backend REST API végpontjainak funkcionális teszteléséhez kezdetben a Postman [41] eszközt használtam, amely egy népszerű API tesztelő platform, és lehetővé tette a HTTP kérések egyszerű küldését, a válaszok ellenőrzését, valamint a tesztek automatizálását. A Postman segítségével teszteltem az autentikációs végpontokat (pl. /api/auth/register, /api/auth/login), a fogadási végpontokat (pl. /api/football/bets/match-score), és a ranglista végpontokat (pl. /api/leaderboard/weekly). A Postman biztosította a kérések paramétereinek, fejléceinek (pl. Authorization token) és törzsének egyszerű kezelését, valamint a válaszok státuszkódjainak és tartalmának ellenőrzését. A tesztelés során azonban kipróbáltam a Bruno [42] nevű nyílt forráskódú alternatívát is, amely könnyűsúlyú és helyi fájl alapú tesztelést kínál, így alkalmas volt a projekt kezdeti fázisában a gyors iterációkra. A Bruno használata során azonban hiányoztak bizonyos fejlett funkciók, például a Postman által biztosított környezetváltozók (environment variables) kezelése és a tesztek automatizálásának rugalmassága, ezért végül visszatértem a Postman használatához. A Postman kényelmesebbnek és funkciókban gazdagabbnak bizonyult, különösen a JWT tokenek kezelésére szolgáló pre-request scriptek és a teszteredmények exportálása terén. A manuális tesztelés mindkét eszközzel hatékonyabbnak bizonyult, mint az LLM által generált unit

tesztek, mivel lehetővé tette a valós forgatókönyvek szimulálását és a hibák azonnali észlelését, például a jogosultságkezelési problémák vagy a hibás válaszok esetén.

### 3. Kivitelezési folyamatok

#### 3.1 Az adatbázis és az adatok tárolásának kivitelezése külső API-ból

A meccs adatok begyűjtésére a [football-data.org](https://football-data.org) [22] ingyenes csomagját használtam, amely egy REST API szolgáltatás, és URL paraméterekkel lehetővé teszi a kívánt adatok lekérdezését a csomag által biztosított kereteken belül. Az API használata során azonban több kihívással is szembesültem. Eredetileg egy másik, teljesen ingyenes API-t terveztem használni, amely azonban a fejlesztés közben megszűnt – sem az API, sem a hozzá tartozó weboldal nem volt már elérhető, és még a Wayback Machine [23] segítségével sem tudtam nyomát találni. Ez a váratlan helyzet gyors döntést igényelt, így a [football-data.org](https://football-data.org) API mellett tettem le a voksomat, bár ez kompromisszumokkal járt. Az ingyenes csomag korlátozott funkcionalitása miatt például el kellett vetnem egy eredetileg tervezett csapatösszerakó funkciót, mivel annak megvalósításához túl sok API-hívásra lett volna szükség, amit az ingyenes csomag percalapú limitjei (10 hívás percenként) nem tettek lehetővé, még időzítmény megoldások alkalmazásával sem.


A felmerült kihívás új lehetőségeket is nyitott, és több alternatívát is megvizsgáltam az adatok begyűjtésére. Az egyik opció egy saját webscraper fejlesztése lett volna, amely például az [eredmények.com](https://www.researchgate.net/publication/321111111) [24] weboldaltól gyűjti be az adatokat, és azokat az általam kívánt formátumban tárolja. A webscraping egy olyan technika, amelynek során egy program automatizáltan kinyeri az adatokat egy weboldal HTML struktúrájából, például meccs adatokat vagy eredményeket. Ehhez gyakran használnak olyan eszközöket, mint a Pythonban népszerű BeautifulSoup [28] vagy Scrapy [29], amelyek lehetővé teszik a HTML dokumentumok elemzését és az adatok strukturált formában történő kinyerését. A webscraping előnye, hogy API hiányában is lehetővé teszi az adatok begyűjtését, de számos technikai és etikai problémát vet fel.

Technikai szempontból a weboldalak gyakran alkalmaznak bot-védelmi mechanizmusokat, például a Cloudflare [30] szolgáltatásait, amelyek megnehezítik az automatizált adatgyűjtést. A Cloudflare olyan módszereket használ, mint az IP-címek ellenőrzése és a HTTP-kérések fejléceinek elemzése (pl. User-Agent string), hogy kiszűrje a gyanús forgalmat [31]. Emellett JavaScript-alapú kihívásokat is alkalmazhat, amelyek a böngésző környezetét vizsgálják, például canvas fingerprinting technikával, hogy megállapítsa, valódi felhasználóról vagy botról van-e szó [32]. A webscraper körében bevett gyakorlatok közé tartozik a proxyk használata az IP-címek rotálására, valamint a fejlécek módosítása, hogy a kérések valódi böngészőből érkezőnek tűnjenek. Fejlett esetekben headless böngészőket, például a Puppeteer, használnak a JavaScript kihívások megkerüléséhez, de a Cloudflare rendszerei gyakran még ezeket is észlelik [33].

A projektben azonban több okból is elvettem a webscrapinget. Egyrészt időigényes fejlesztést igényelt volna, mivel a célzott weboldalak HTML struktúrája bármikor megváltozhat, ami a scraper meghibásodásához vezethetett volna. Másrészt a webscraping etikai kérdéseket is felvet, hiszen extra terhelést jelent a célzott weboldal számára, különösen akkor, ha az nem biztosít hivatalos API-t az adatok elérésére. Emellett fennáll annak a kockázata, hogy az oldal IP-alapú tiltással vagy más szűrőkkel, például a Cloudflare-hez hasonló rendszerekkel blokkolja a

hozzáférést. A webscraping jogi szempontból is kockázatos lehet, mivel sok weboldal használati feltételei tiltják az automatizált adatgyűjtést. Ezeket a szempontokat mérlegelve végül a football-data.org API mellett maradtam, amely bár korlátozott, de stabil és dokumentált megoldást kínált, és nem vetett fel etikai vagy jogi problémákat.

Az API ingyenes csomagjának percalapú limitje (10 hívás percenként) nem tette lehetővé a valós idejű adatfrissítést, ezért egy optimalizált adatkezelési stratégiát dolgoztam ki. A Spring WebClient [25] segítségével a backend oldalon kértem le az adatokat, amelyeket aztán a MongoDB adatbázisban tároltam el.



```
@Configuration
public class WebClientConfig {

    @Value("${api.url}")
    private String apiUrl;

    @Value("${api.key}")
    private String apiKey;

    @Bean
    public WebClient webClient() {
        return WebClient.builder()
            .baseUrl(apiUrl)
            .defaultHeader("X-Auth-Token", apiKey)
            .codecs(configurer -> configurer
                .defaultCodecs()
                .maxInMemorySize(10 * 1024 * 1024))
            .build();
    }
}
```

*6. ábra - A WebClient alap konfigurációja, az API által elvárt egyedi token fejléccel*

Az adatokat két kategóriába soroltam: a statikus és kevésbé dinamikus adatokat (pl. csapatok, ligák, szezonok adatai) havonta egyszer frissítettem, míg a dinamikusabb meccsadatokat (pl. eredmények, státuszok) 15 percenként. A frissítési folyamatot a Spring Scheduling [26] cron jobjaival automatizáltam, amelyek a DataFetchService osztály metódusait hívják meg ütemezett időközönként. A DataFetchService felelős az adatok lekérdezéséért a WebClient segítségével, majd azok megfelelő formátumban történő eltárolásáért a MongoDB-ben. Az így tárolt adatokat Data Transfer Object-ek (DTO-k) segítségével alakítottam át, hogy a frontend számára könnyen feldolgozható formában adhassam át őket, például a meccsek listáját vagy a ranglistákhoz szükséges adatokat.



```

@Component
@Slf4j
@RequiredArgsConstructor
public class DataFetchScheduler {

    private final DataFetchService dataFetchService;
    private final FootballService footballService;

    @Scheduled(cron = "0 0 2 1 * *")
    public void scheduledMonthlyDataFetch() {
        try {
            log.info("Starting monthly scheduled data fetch");
            dataFetchService.fetchAndSaveAreas();
            Thread.sleep(60000);
            dataFetchService.fetchAndSaveCompetitionsWithSeasons();
            Thread.sleep(60000);
            dataFetchService.fetchAndSaveTeamsWithPlayers();
            log.info("Finished monthly scheduled data fetch");
        } catch (Exception e) {
            log.error("Error during monthly scheduled data fetch", e);
        }
    }

    @Scheduled(cron = "0 0/15 * * * *")
    public void scheduledFrequentlyDataFetch() {
        try {
            log.info("Starting frequently scheduled data fetch");
            dataFetchService.fetchAndSaveMatches();
            footballService.updateBetStatuses();
            log.info("Finished frequently scheduled data fetch");
        } catch (Exception e) {
            log.error("Error during frequently scheduled data fetch", e);
        }
    }
}

```

*7. ábra - Az ütemező osztály, a havi és 15 percenkénti adatgyűjtéssel*

Az adatbázis használata nem korlátozódott a külső API-ból származó adatok tárolására. A felhasználók, fogadások és pontok kezelésére is szükségem volt a MongoDB-re, így az adatbázis kapacitását teljes mértékben kihasználtam. A MongoDB választása azonban újabb technikai kihívást jelentett, mivel ez az adatbázisrendszer alapértelmezésként UUID (Universally Unique Identifier) [27] típusú azonosítókat használ, amelyek alfanumerikus karakterekből álló, kötőjellel elválasztott stringek (pl. 123e4567-e89b-12d3-a456-426614174000). Ezzel szemben a football-data.org API által biztosított azonosítók klasszikus integer alapú számok voltak, amelyeket célszerűbb volt megtartani, hiszen ezekre hivatkoztak a különböző API-végpontok. Az UUID-k használata ebben az esetben felesleges bonyolultságot jelentett volna, például egy további azonosító oszlop bevezetésével, ami redundanciát okozott volna az adatbázisban.



```

@Data
@Document
@Builder
@AllArgsConstructor
@NoArgsConstructor
public class DatabaseSequence {
    @Id
    private String id;
    private Integer sequence;
}

@Service
@RequiredArgsConstructor
public class SequenceGeneratorService {
    private final MongoOperations mongoOperations;

    public int generateSequence(String sequenceName) {
        DatabaseSequence counter = mongoOperations.findAndModify(
            Query.query(Criteria.where("_id").is(sequenceName)),
            new Update().inc("sequence", 1),
            FindAndModifyOptions.options().returnNew(true).upsert(true),
            DatabaseSequence.class);
        return counter != null ? counter.getSequence() : 1;
    }
}

```

8. ábra - A DatabaseSequence modell és a SequenceGeneratorService osztály

Bár felmerült a lehetőség, hogy másik adatbázisrendszerre váltsak – például a MariaDB-re [28], amely hagyományos relációs adatbázisrendszerként egyszerűbben kezelte volna az integer alapú azonosítókat –, a MongoDB mellett döntöttem, mert szerettem volna kipróbálni és megismerni ezt a NoSQL technológiát, amely a projekt rugalmas adatstruktúráihoz egyébként is jól illeszkedett. A problémát végül úgy oldottam meg, hogy az egész adatbázist integer alapú azonosítókra állítottam be. A külső API-ból származó adatoknál ez nem jelentett nehézséget, mivel azok már eleve integer azonosítókat tartalmaztak. A saját kollekciókhoz – például a felhasználókhoz (users) és fogadásokhoz (match\_score\_bets) – azonban egy egyedi megoldást kellett implementálnom. Ehhez létrehoztam egy segédkollekciót, amelyet DatabaseSequence-nek neveztem el, és egy SequenceGeneratorService osztályt, amely a szekvenciák kezelését végzi.

Amikor új felhasználó vagy fogadás jött létre, a SequenceGeneratorService növelte a megfelelő szekvenciát (pl. user\_sequence, match\_score\_bet\_sequence), és ennek alapján generált egy új integer alapú azonosítót. Ez a megoldás biztosította az azonosítók konzisztenciáját az egész adatbázisban, és elkerültem az olyan hibákat, amelyek az azonosító típusok (pl. Integer vs. String) eltéréséből adódhattak volna.

```
MatchScoreBet.java

@Data
@Builder
@NoArgsConstructor
@AllArgsConstructor
@Document(collection = "match_score_bets")
public class MatchScoreBet {
    @Id
    private Integer id;
    @Transient
    public static final String SEQUENCE_NAME = "match_score_bet_sequence";
    private Integer userId;
    private Integer matchId;
    private Integer homeScoreBet;
    private Integer awayScoreBet;
    private BetStatus status;
    private Instant date;
}
```

10. ábra - Az egyedi szekvencia használata a MatchScoreBet modellben

A MongoDB-ben a kapcsolatok kezelése is eltér a hagyományos relációs adatbázisoktól. Bár a MongoDB támogatja a relációs jellegű kapcsolatokat, ezek használata jelentősen lassabb lehet, és a NoSQL adatbázisok esetében nem is bevett gyakorlat. Ehelyett a kapcsolatokat úgy valósítottam meg, hogy a kapcsolódó kollekciók azonosítóit tároltam a megfelelő dokumentumokban. Például a match\_score\_bets kollekcióban a userId és matchId mezők hivatkoznak a users és matches kollekciók \_id mezőire. A Spring Data MongoDB repository-k segítségével ezek az azonosítók alapján könnyen lekérdezhettem a szükséges adatokat, legyen szó

```
DataFetchService.java

private void savePlayers(TeamsResponse teamsResponse) {
    try {
        log.debug("Saving player");
        List<TeamExternalDTO> teamsDTO = teamsResponse.getTeams();
        List<Player> players;
        players = teamsDTO.stream()
            .map(TeamExternalDTO::getSquad)
            .flatMap(List::stream)
            .collect(Collectors.toList());

        List<Player> savedPlayers = playerRepository.saveAll(players);
        log.info("Successfully saved {} players", savedPlayers.size());
        log.debug("First saved player: {}", savedPlayers.getFirst());
    } catch (NoSuchElementException e) {
        log.debug("Players not found");
    } catch (Exception e) {
        log.debug("Error during MongoDB player save operation", e);
    }
}
```

9. ábra - A játékosok elmentésére szolgáló metódus

meccsekről, szezonokról, ligákról vagy csapatokról. Ez a megközelítés hatékony és a MongoDB filozófiájához illeszkedő adatkezelést tett lehetővé.

A fejlesztési folyamat során felmerült kihívások – például az API megszűnése, a limitált hívásszám és az azonosítók kezelése – értékes tanulási lehetőséget biztosítottak. A választott megoldások, mint a Spring Scheduling használata az ütemezett frissítésekhez, a WebClient integrálása az API-hívásokhoz, és a DatabaseSequence segédkollekció alkalmazása az azonosítók kezelésére, lehetővé tették, hogy a projekt adatkezelési rendszere stabil és hatékony legyen, miközben a MongoDB nyújtotta rugalmasságot is ki tudtam használni.



```
public void fetchAndSaveTeamsWithPlayers() {
    List<Integer> competitionIds = getIdsForPlanAndType();

    try {
        log.debug("Starting teams fetch from API for {} leagues", competitionIds.size());

        competitionIds.forEach(competitionId -> {
            try {
                TeamsResponse response = webClient
                    .get()
                    .uri(uriBuilder -> uriBuilder
                        .path("/competitions/{competitionId}/teams")
                        .build(competitionId))
                    .retrieve()
                    .bodyToMono(TeamsResponse.class)
                    .block();

                if (response != null) {
                    saveTeam(competitionId, response);
                    savePlayers(response);
                } else {
                    log.warn("Null response received for teams with players for competition ID: {}", competitionId);
                }
            } catch (Exception e) {
                log.error("Error during MongoDB team save operation", e);
            }
        });

        log.info("Completed processing teams with players for all competition IDs");
    } catch (Exception e) {
        log.error("Error in fetchAndSaveTeamsWithPlayers", e);
        throw e;
    }
}
```

11. ábra - A WebClient alkalmazása az adatok lekérésére

### 3.2 A felhasználókezelés kivitelezése

A felhasználókezelés a projekt egyik alapvető komponense, amely biztosítja a felhasználók regisztrációját, bejelentkezését, jogosultságkezelését, valamint a munkamenetek kezelését. A felhasználókezelés implementációjához a Spring Security keretrendszert [33] használtam, amely egy robusztus és testre szabható megoldás a hitelesítés és jogosultságkezelés megvalósítására Java alapú alkalmazásokban. A Spring Security integrálásával egy JWT (JSON Web Token) [34] alapú hitelesítési rendszert alakítottam ki, amely stateless munkamenetkezelést biztosít, így az alkalmazás skálázhatósága és hatékonysága is javult. Az alábbiakban részletesen bemutatom a felhasználókezelés kivitelezésének lépéseit, a felhasznált technológiákat, valamint a technikai döntések hátterét.

A felhasználókezelés alapvető konfigurációját a `WebSecurityConfig` osztályban valósítottam meg, amely a Spring Security beállításait tartalmazza. A Spring Security konfigurációját a `@Configuration` és `@EnableWebSecurity` annotációk segítségével inicializáltam, így biztosítva, hogy a biztonsági szűrők és szabályok az alkalmazás indításakor betöltődjenek. A `SecurityFilterChain` metódus definiálja a HTTP kérésekhez kapcsolódó biztonsági szabályokat. A CSRF (Cross-Site Request Forgery) védelmet kikapcsoltam, mivel a JWT alapú hitelesítés stateless, és nem tárol munkameneteket, így a CSRF támadások kockázata minimális [35]. A CORS (Cross-Origin Resource Sharing) konfigurációt is engedélyeztem, hogy a frontend alkalmazás – amely különálló domainről érkezik – hozzáférhessen a backend API-hoz. A CORS beállítások lehetővé tették, hogy minden eredetről (\*) érkező kérést elfogadjak, és a leggyakrabban használt HTTP metódusokat (GET, POST, PATCH, DELETE, OPTIONS) támogassam.

```
WebSecurityConfig.java

@Bean
public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
    http
        .csrf(AbstractHttpConfigurer::disable)
        .cors(cors -> cors.configurationSource(corsConfigurationSource()))
        .authorizeHttpRequests(auth -> auth
            .requestMatchers(HttpMethod.OPTIONS, "/**").permitAll()
            .requestMatchers("/api/auth/**").permitAll()
            .requestMatchers("/api/data/**").hasRole("ADMIN")
            .requestMatchers("/api/football/**").hasAnyRole("USER", "ADMIN")
            .requestMatchers("/api/user/**").hasAnyRole("USER", "ADMIN")
            .requestMatchers("/api/leaderboard/**").hasAnyRole("USER", "ADMIN")
            .anyRequest().authenticated()
        )
        .sessionManagement(session -> session
            .sessionCreationPolicy(SessionCreationPolicy.STATELESS)
        )
        .authenticationProvider(authenticationProvider())
        .addFilterBefore(jwtAuthenticationFilter(), UsernamePasswordAuthenticationFilter.class);

    System.out.println("SecurityFilterChain initialized with /api/football/** -> hasAnyRole('USER', 'ADMIN')");
    return http.build();
}
```

12. ábra - A `WebSecurityConfig` osztály `securityFilterChain` metódusa, amely a HTTP kérések jogosultsági szabályait definiálja

A jogosultságkezelést a Spring Security `authorizeHttpRequests` metódusával valósítottam meg, amely különböző útvonalakhoz eltérő hozzáférési szabályokat definiál. Például az `/api/auth/**` útvonalak mindenki számára elérhetők, mivel ezek a regisztrációt és bejelentkezést szolgálják. Az `/api/data/**` útvonalakhoz csak az ADMIN szerepkörrel rendelkező felhasználók férhetnek hozzá, míg az `/api/football/**`, `/api/user/**` és `/api/leaderboard/**` útvonalakhoz a USER vagy ADMIN szerepkör szükséges. Minden egyéb kéréshez hitelesítés szükséges, amelyet a `anyRequest().authenticated()` szabály biztosít. A munkamenetkezelést stateless módra állítottam a `SessionCreationPolicy.STATELESS` beállítással, így az alkalmazás nem tárol szerveroldali munkameneteket, és minden kérésnél a JWT token alapján történik a hitelesítés.

A hitelesítési folyamatot egy egyedi `JwtAuthenticationFilter` szűrővel egészítettem ki, amelyet a Spring Security szűrőláncába illesztettem a `UsernamePasswordAuthenticationFilter` elé. Ez a szűrő minden bejövő kérésnél ellenőrzi a HTTP fejlécben található `Authorization` mezőt, amely a JWT tokenet tartalmazza a „Bearer” előtaggal. Ha a token érvényes és nem szerepel a feketelistán (amelyet a `TokenBlacklistService` kezel), a szűrő kinyeri a tokenből a felhasználó nevét, betölti a felhasználó adatait a `UserDetailsService` segítségével, és beállítja a Spring Security kontextusát (`SecurityContextHolder`). Ez biztosítja, hogy a jogosultságellenőrzés a kérés további feldolgozása során megfelelően működjön. A `UserDetailsService` implementációja a `UserRepository` segítségével keresi meg a felhasználót a felhasználónév alapján, és a `UserDetailsMapper` segítségével alakítja át a `User` entitást a Spring Security által elvárt `UserDetails` formátummá.



```

@Override
protected void doFilterInternal(
    @NonNull HttpServletRequest request,
    @NonNull HttpServletResponse response,
    @NonNull FilterChain filterChain
) throws ServletException, IOException {
    final String authHeader = request.getHeader("Authorization");
    if (authHeader == null || !authHeader.startsWith("Bearer ")) {
        filterChain.doFilter(request, response);
        return;
    }
    final String jwt = authHeader.replace("Bearer ", "");
    if (tokenBlacklistService.isBlacklisted(jwt)) {
        filterChain.doFilter(request, response);
        return;
    }
    final String username = tokenService.extractUsername(jwt);
    if (username != null && SecurityContextHolder.getContext().getAuthentication() == null) {
        UserDetails userDetails = userDetailsService.loadUserByUsername(username);
        if (tokenService.isValid(jwt)) {
            UsernamePasswordAuthenticationToken authToken = new UsernamePasswordAuthenticationToken(
                userDetails,
                null,
                userDetails.getAuthorities()
            );
            authToken.setDetails(new WebAuthenticationDetailsSource().buildDetails(request));
            SecurityContextHolder.getContext().setAuthentication(authToken);
        }
    }
    filterChain.doFilter(request, response);
}

```

*13. ábra - A `JwtAuthenticationFilter` osztály `doFilterInternal` metódusa*

A jelszavak biztonságos tárolásához a BCrypt algoritmust [36] használtam, amelyet a `PasswordEncoder` bean biztosít. A BCrypt egy adaptív hash algoritmus, amely a jelszavakat sózott hash formájában tárolja, így jelentősen megnehezíti a jelszavak visszafejtését, még ha az adatbázis kompromittálódik is. A `DaoAuthenticationProvider` segítségével integráltam a `UserDetailsService`-t és a `PasswordEncoder`-t a Spring Security hitelesítési folyamatába, így a bejelentkezés során a jelszavak ellenőrzése biztonságosan történik.

A tokenkezelést a `TokenService` osztály végzi, amely a JWT tokenek generálását, validálását és a bennük tárolt információk kinyerését kezeli. A tokenek generálásához a JJWT könyvtárat [37] használtam, amely lehetővé teszi a JWT tokenek létrehozását és ellenőrzését. A `generateAccessToken` metódus egy JWT tokenet állít elő, amely tartalmazza a felhasználó nevét (subject), szerepköreit (roles claim), kiállítási idejét (issuedAt), és lejáratási idejét (expiration). A

token aláírása egy titkos kulccsal történik, amelyet a SECRET\_KEY konfigurációs tulajdonságból származtatok, és a HMAC-SHA algoritmus segítségével biztosítom a token integritását. A token lejáratát a JWT\_EXPIRATION tulajdonság határozza meg, amely lehetővé teszi a token élettartamának tesztelését. A generateRefreshToken metódus egy egyszerű UUID alapú refresh token-t állít elő, amelyet a felhasználó munkamenetének meghosszabbítására használok. A TokenService emellett biztosítja a tokenek validálását (isTokenValid) és a bennük tárolt információk kinyerését (pl. extractUsername, extractExpiration), így a hitelesítési folyamat során ellenőrizhető, hogy a token érvényes-e és nem járt-e le.

A kijelentkezés és a tokenek érvénytelenítése érdekében egy TokenBlacklistService osztályt implementáltam, amely egy ConcurrentHashMap alapú feketelistát használ a kijelentkezés során érvénytelenített tokenek tárolására. Amikor egy felhasználó kijelentkezik, mind az access token, mind a refresh token feketelistára kerül, így azok többé nem használhatók hitelesítésre. A JwtAuthenticationFilter minden kérésnél ellenőrzi, hogy a token szerepel-e a feketelistán, és ha igen, a kérést elutasítja. Ez a megoldás biztosítja, hogy a kijelentkezés után a tokenek ne legyenek tovább használhatók, még akkor sem, ha azok lejáratát még nem telt le.

A felhasználókezelés logikáját az AuthService osztályban valósítottam meg, amely a regisztrációt, bejelentkezést, tokenfrissítést és kijelentkezést kezeli. A regisztráció során az AuthService ellenőrzi, hogy a megadott email cím és felhasználónév még nem létezik-e az adatbázisban, majd egy új User entitást hoz létre. A felhasználó jelszavát a PasswordEncoder segítségével kódolom, és a felhasználó szerepkörként alapértelmezés szerint a „USER” szerepkört állítom be. Az azonosító generálásához a SequenceGeneratorService-t használom, amely biztosítja, hogy a felhasználók integer alapú azonosítókat kapjanak, összhangban az adatbázis többi részével. A regisztráció végén a TokenService segítségével generálok egy access token-t és egy refresh token-t, amelyeket egy AuthResponseDTO objektumban adok vissza a frontend számára.

A bejelentkezés során az AuthService a Spring Security AuthenticationManager-ét használja a felhasználó hitelesítésére. A UsernamePasswordAuthenticationToken segítségével adom át a felhasználónevet és jelszót, amelyeket az AuthenticationManager a DaoAuthenticationProvider segítségével ellenőriz. Sikeres hitelesítés esetén a UserRepository segítségével lekérdezem a felhasználót, majd a TokenService segítségével új access és refresh tokeneket generálok. A refresh token-t a felhasználó entitásában tárolom, hogy később a tokenfrissítés során ellenőrizni tudjam. A tokenfrissítés (refreshToken metódus) lehetővé teszi, hogy a felhasználó új access token-t kapjon anélkül, hogy újra be kellene jelentkeznie, amennyiben

a refresh token érvényes. A kijelentkezés (logout módszer) során a TokenBlacklistService segítségével érvénytelenítem a tokeneket, és törölöm a refresh token a felhasználó entitásából.



```
private AuthResponseDTO getAuthResponse(User user) {
    String jwtToken = tokenService.generateAccessToken(user);
    String refreshToken = tokenService.generateRefreshToken();
    user.setRefreshToken(refreshToken);

    userRepository.save(user);

    return AuthResponseDTO.builder()
        .accessToken(jwtToken)
        .refreshToken(refreshToken)
        .user(user.getUsername())
        .userId(user.getId())
        .roles(user.getRoles())
        .build();
}

public AuthResponseDTO login(LoginRequestDTO request) {
    authenticationManager.authenticate(
        new UsernamePasswordAuthenticationToken(
            request.getUsername(),
            request.getPassword()
        )
    );

    User user = userRepository.findByUsername(request.getUsername())
        .orElseThrow(() -> new RuntimeException("User not found"));

    return getAuthResponse(user);
}
```

*14. ábra - Az AuthService osztály register módszere*

A felhasználókezelés implementációja során fontos szempont volt a biztonság és a hatékonyság egyensúlyának megteremtése. A Spring Security és a JWT alapú hitelesítés biztosította, hogy a felhasználók adatai védettek legyenek, míg a stateless munkamenetkezelés és a token alapú hitelesítés lehetővé tette az alkalmazás skálázhatóságát. A TokenBlacklistService használata további biztonsági réteget adott a kijelentkezés kezeléséhez, míg a BCrypt algoritmus a jelszavak biztonságos tárolását garantálta. A felhasználókezelés ezen implementációja szilárd alapot biztosított a projekt számára, és lehetővé tette, hogy a felhasználók biztonságosan és hatékonyan használhassák az alkalmazást.

### 3.3 A REST végpontok és szervizeik létrehozása

A projekt REST API-ját öt kontroller osztály kezeli, amelyek logikailag elkülönítik a különböző funkciókat, így biztosítva a kód rendszerezett és átlátható felépítését. Az összes végpont alapja az /api előtag, amelyet a kategóriák szerinti útvonalak követnek (pl. /api/auth, /api/data). Az egyes végpontok elérhetőségét és jogosultsági szabályait a WebSecurityConfig osztályban definiált securityFilterChain módszerben állítottam be, ahogy azt a felhasználókezelés



fejezetben részleteztem. Az alábbiakban bemutatom az egyes kontrollerek osztályokat, a hozzájuk tartozó végpontokat, valamint a mögöttes szerviz osztályok szerepét a logika megvalósításában.

```
AuthController.java

@RestController
@RequestMapping("/api/auth")
@RequiredArgsConstructor
public class AuthController {
    private final AuthService authService;
    @PostMapping("/register")
    public ResponseEntity<AuthResponseDTO> register(@Valid @RequestBody RegisterRequestDTO request) {
        return ResponseEntity.ok(authService.register(request));
    }
}
```

#### 15. ábra - Az autentikációs osztály regisztrációs végpontjának módszere

Az AuthController az autentikációs végpontokat kezeli az /api/auth útvonal alatt. Ezek a végpontok a regisztrációt, bejelentkezést, tokenfrissítést és kijelentkezést szolgálják, és mind POST metódussal érhetőek el. A végpontok a megfelelő DTO-kat (Data Transfer Object) várják bemenetként, például a RegisterRequestDTO-t a regisztrációhoz vagy a LoginRequestDTO-t a bejelentkezéshez.

```
RegisterRequestDTO.java

@Data
@Builder
@AllArgsConstructor
@NoArgsConstructor
public class RegisterRequestDTO {
    @NotBlank(message = "Email is required")
    @Email(message = "Email must be a valid email address")
    String email;

    @NotBlank(message = "Username is required")
    @Size(min = 3, max = 20, message = "Username must be between 3 and 20 characters")
    String username;

    @NotBlank(message = "Password is required")
    @Size(min = 6, message = "Password must be at least 6 characters long")
    String password;
}
```

#### 16. ábra - A RegisterRequestDTO Jakarta Validation validációval

A bemenetek validálását a Jakarta Validation könyvtár [36] @Valid annotációjával végzem, amely a DTO-kban definiált validációs szabályok (pl. kötelező mezők, email formátum) alapján ellenőrzi az adatokat. A validált kéréseket az AuthService osztálynak adom át, amely a hitelesítési logikát kezeli, például a felhasználó regisztrálását, a JWT tokenek generálását, vagy a kijelentkezés



során a tokenek feketelistára helyezését. A válaszok AuthResponseDTO formátumban érkeznek, amely tartalmazza a generált tokeneket, a felhasználó nevét, azonosítóját és szerepköreit. A kijelentkezés végpont (/api/auth/logout) esetében a kérés fejlécében található Authorization mezőből kinyert access tokenet és a kérés törzsében érkező refresh tokenet is figyelembe veszem a tokenek érvénytelenítéséhez.

A DataController az /api/data útvonal alatt található végpontokat kezeli, amelyek az adatok külső API-ból történő lekérdezését és tárolását szolgálják. Ezek a végpontok kizárólag ADMIN szerepkörrel rendelkező felhasználók számára érhetőek el, ahogy azt a securityFilterChain metódusban definiáltam. A kontroller négy POST metódust biztosít: /fetch-areas, /fetch-competitions-with-seasons, /fetch-teams-with-players és /fetch-matches. Mindegyik végpont a DataFetchService osztály megfelelő metódusát hívja meg, amely a Spring WebClient segítségével lekéri az adatokat a football-data.org API-ból, majd eltárolja azokat a MongoDB adatbázisban. A végpontok nem várnak bemenetet, és sikeres végrehajtás esetén egy üres ResponseEntity-t adnak vissza 200 OK státusszal. Hibakezelést is implementáltam: ha a lekérdezés során kivétel történik, a végpont 500 Internal Server Error státusszal tér vissza. A DataFetchService osztály felelős az adatok strukturált formában történő mentéséért, például a ligák, csapatok és meccsek megfelelő kollekciókba történő tárolásáért, amelyeket később a többi végpont használ fel.



```
@GetMapping("/bets/match-score/{userId}/{matchId}")
public ResponseEntity<?> getMatchScoreBet(@PathVariable Integer userId, @PathVariable Integer matchId) {
    return footballService.getMatchScoreBet(userId, matchId);
}

@PostMapping("/bets/match-score")
public ResponseEntity<DefaultResponseDTO> makeMatchScoreBet(@RequestBody MatchScoreBet bet) {
    bet.setId(sequenceGenerator.generateSequence(MatchScoreBet.SEQUENCE_NAME));
    bet.setDate(Instant.now());
    bet.setStatus(BetStatus.LIVE);
    return footballService.makeMatchScoreBet(bet);
}

@PatchMapping("/bets/match-score/{userId}")
public ResponseEntity<DefaultResponseDTO> updateMatchScoreBet(
    @PathVariable Integer userId,
    @RequestBody UpdateBetDTO updatedBet
) {
    return footballService.updateMatchScoreBet(userId, updatedBet);
}

@DeleteMapping("/bets/match-score/{userId}/{matchId}")
public ResponseEntity<DefaultResponseDTO> cancelMatchScoreBet(@PathVariable Integer userId, @PathVariable Integer matchId) {
    return footballService.cancelMatchScoreBet(userId, matchId);
}
```

### *17. A FootballController osztály fogadással kapcsolatos végpontjai*

A FootballController az /api/football útvonal alatt kezeli a fogadásokkal és meccs adatokkal kapcsolatos végpontokat, amelyek USER vagy ADMIN szerepkörrel rendelkező felhasználók számára érhetőek el. A kontroller többféle HTTP metódust támogat, hogy a fogadások kezelését és a meccs adatok lekérdezését rugalmasan biztosítsa. A /bets/match-score végpont (POST) lehetővé teszi a felhasználók számára, hogy új fogadást tegyenek egy meccs eredményére (MatchScoreBet). A kérésben érkező fogadási adatokat a FootballService osztály makeMatchScoreBet metódusa dolgozza fel, amely ellenőrzi, hogy a meccs létezik-e, még nem kezdődött-e el, és a felhasználó nem tett-e már fogadást az adott meccsre. A fogadás azonosítóját a SequenceGeneratorService segítségével generálom, és a fogadás státuszát LIVE értékre állítom. A /bets/match-

score/{userId}/{matchId} végpont (GET) lekérdezi egy adott felhasználó adott meccsre tett fogadását, míg a /bets/match-score/{userId} végpont (PATCH) lehetővé teszi a fogadás módosítását, a /bets/match-score/{userId}/{matchId} végpont (DELETE) pedig a fogadás törlését, mindaddig, amíg a meccs nem kezdődött el. A /get/leagues végpont (GET) a ligák listáját adja vissza, a /get/{leagueId}/upcoming-matches végpont (GET) pedig egy adott liga közelgő meccseit, amelyeket a következő 28 napban rendeznek. Végül a /get/{userId}/bets/match-score végpont (GET) egy felhasználó összes fogadását listázza ki. A válaszok DTO-k formájában érkeznek (pl. MatchResponseDTO, MatchScoreBetResponseDTO), amelyek a frontend számára könnyen feldolgozható formátumban tartalmazzák az adatokat.



```

public ResponseEntity<DefaultResponseDTO> cancelMatchScoreBet(Integer userId, Integer matchId) {
    ...
    Optional<MatchScoreBet> existingBet = matchScoreBetRepository.findByUserIdAndMatchId(userId, matchId);
    if (existingBet.isEmpty()) {
        return ResponseEntity
            .status(HttpStatus.OK)
            .body(DefaultResponseDTO.builder()
                .success(false)
                .message("No bet found to cancel for match: " + matchId)
                .build());
    }
    Match match = matchRepository.getMatchById(matchId);
    if (match == null || !List.of(Status.SCHEDULED, Status.TIMED).contains(match.getStatus())) {
        return ResponseEntity
            .status(HttpStatus.OK)
            .body(DefaultResponseDTO.builder()
                .success(false)
                .message("Match not found or already started: " + matchId)
                .build());
    }
    matchScoreBetRepository.delete(existingBet.get());
    return ResponseEntity
        .status(HttpStatus.OK)
        .body(DefaultResponseDTO.builder()
            .success(true)
            .message("Bet cancelled successfully for match: " + matchId)
            .build());
    ...
}


```

18. ábra - A FootballService osztály cancelMatchScoreBet metódusának részlete (a try-catch blokkot nem tartalmazza)

A FootballService osztály biztosítja a FootballController mögöttes logikáját, és a Spring Data MongoDB [37] repository-k segítségével kommunikál az adatbázissal. A makeMatchScoreBet metódus például a MatchRepository és MatchScoreBetRepository segítségével ellenőrzi a meccs státuszát és a meglévő fogadásokat, majd menti az új fogadást. A updateBetStatuses metódus a fogadások státuszát frissíti a meccsek eredményei alapján, és a LeaderboardService-t hívja meg a ranglista frissítéséhez. A getUpcomingMatchesForLeague metódus a MatchRepository segítségével lekéri a közelgő meccseket, amelyeket a MatchResponseMapper segítségével DTO-kká alakítok. A FootballService hibakezelést is tartalmaz: minden metódus try-catch blokkban fut, és hibák esetén megfelelő hibaüzenetet ad

vissza a DefaultResponseDTO vagy más DTO-k segítségével. A logolást a SLF4J [38] könyvtár segítségével valósítottam meg, amely lehetővé teszi a hibák naplózását és a rendszer működésének nyomon követését.

A LeaderboardController az /api/leaderboard útvonal alatt biztosít végpontokat a ranglisták lekérdezéséhez, amelyek USER vagy ADMIN szerepkörrel rendelkező felhasználók számára érhetők el. A kontroller két GET metódust tartalmaz: a /weekly végpont a heti ranglistát, a /monthly végpont pedig a havi ranglistát adja vissza. Mindkét végpont a LeaderboardService osztály megfelelő metódusát hívja meg, amely a felhasználók fogadásai és pontjai alapján állítja össze a ranglistát. A válaszok LeaderboardEntry objektumok listájaként érkeznek, amelyek tartalmazzák a felhasználók nevét, pontszámát és helyezését. A LeaderboardService a MatchScoreBetRepository segítségével gyűjti össze a fogadásokat, és a meccsek eredményei alapján számolja ki a pontokat, amelyeket időalapú szűréssel (heti vagy havi) rendez a ranglista készítésekor.



```
public List<LeaderboardEntry> getMonthlyLeaderboard() {
    ZonedDateTime monthStart = ZonedDateTime.now()
        .withDayOfMonth(1)
        .truncatedTo(ChronoUnit.DAYS);

    Instant startInstant = monthStart.toInstant();
    Instant endInstant = monthStart.plusMonths(1).toInstant();

    return getLeaderboardForPeriod(startInstant, endInstant);
}

private List<LeaderboardEntry> getLeaderboardForPeriod(Instant start, Instant end) {
    List<UserPoints> periodPoints = userPointsRepository.findByMatchDateBetween(start, end);

    Map<Integer, Integer> userPointsMap = periodPoints.stream()
        .collect(Collectors.groupingBy(
            UserPoints::getUserId,
            Collectors.summingInt(UserPoints::getPoints)
        ));

    return userPointsMap.entrySet().stream()
        .map(entry -> LeaderboardEntry.builder()
            .userId(entry.getKey())
            .totalPoints(entry.getValue())
            .build())
        .sorted(Comparator.comparing(LeaderboardEntry::getTotalPoints).reversed())
        .collect(Collectors.toList());
}
```

19. ábra - A havi ranglétráért felelős metódusai a LeaderboardService osztálynak

A UserController az /api/user útvonal alatt kezeli a felhasználói adatokkal kapcsolatos végpontokat, szintén USER vagy ADMIN szerepkörrel rendelkező felhasználók számára. A kontroller két GET metódust biztosít: a /userinfo végpont egy UserRequestDTO alapján adja vissza a felhasználó adatait, míg a /{userId} végpont egy adott felhasználó azonosítója alapján kérdezi le

az adatokat. Mindkét végpont a UserService osztály megfelelő metódusát hívja meg, amely a UserRepository segítségével lekéri a felhasználói adatokat, és UserResponseDTO formátumban adja vissza azokat. A UserService biztosítja, hogy a lekérdezett adatok (pl. felhasználónév, email, szerepkörök) a frontend számára megfelelő formátumban legyenek elérhetők, és hibakezelést is tartalmaz a nem létező felhasználók esetére.

A REST végpontok és szervizek implementációja során fontos szempont volt a modularitás és a fenntarthatóság. Az öt controller logikailag elkülönítette a különböző funkcionálisokat, míg a szerviz osztályok (pl. AuthService, DataFetchService, FootballService, LeaderboardService, UserService) a mögöttes üzleti logikát kezelték, így a kód könnyen bővíthető és karbantartható maradt. A DTO-k használata biztosította, hogy a backend és a frontend közötti kommunikáció strukturált és típusbiztos legyen, míg a Spring Data MongoDB repository-k hatékony adatbázis-hozzáférést tettek lehetővé. A hibakezelés és a naplózás implementálása pedig hozzájárult az alkalmazás stabilitásához és hibakereshetőségéhez.

### 3.4 A frontend elkészítése

A frontend fejlesztése során a cél egy egyszerű, de működőképes felhasználói felület létrehozása volt, amely támogatja a backend által biztosított funkcionálisokat, például a bejelentkezést, regisztrációt, fogadások leadását és a ranglisták megtekintését. A 2.1-es fejezetben bemutatott eszközök (React, TypeScript, Tailwind CSS, Axios, Zustand, Formik, Yup, React Router) segítségével készítettem el az alkalmazást, amelynek mappastruktúrája logikusan elkülöníti a különböző funkcionálisokat. A src mappán belül az api könyvtár a backend kommunikációt, a components könyvtár a React komponenseket, az pages könyvtár az oldalakat, a store könyvtár az állapotkezelést, a types könyvtár a típusdefiníciókat, a validation könyvtár a validációs sémákat, az assets könyvtár pedig a statikus fájlokat (pl. soccer-ball.svg) tartalmazza.



```
import { LoginForm } from '../components/LoginForm';

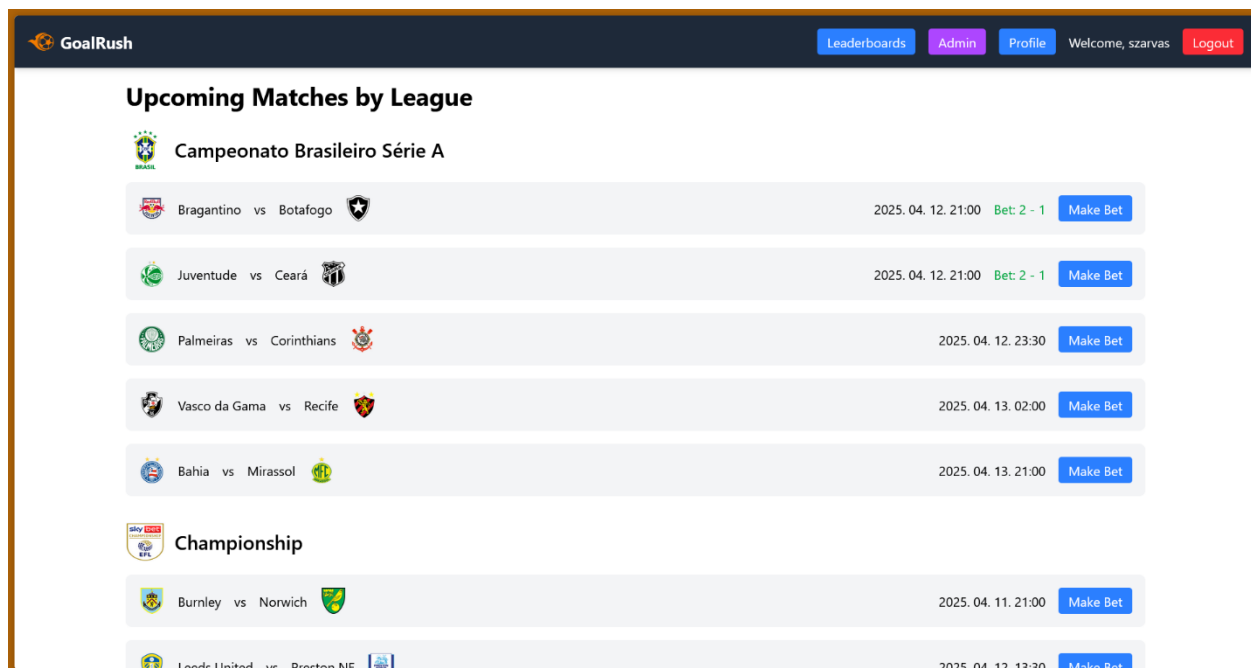
export const Login = () => {
  return (
    <div className="min-h-screen flex items-center justify-center bg-gray-100">
      <div className="bg-white p-6 rounded shadow-md">
        <h2 className="text-2xl font-bold mb-4">Login</h2>
        <LoginForm />
      </div>
    </div>
  );
};
```

20. ábra - A Login.tsx, ami megjeleníti a LoginForm.tsx-et

A bejelentkezési oldal (Login.tsx) az alkalmazás belépési pontja, amely a LoginForm.tsx komponenst használja a bejelentkezési űrlap megjelenítésére. A felhasználó a következő lépéseket

követi: 1) Megadja a felhasználónevét és jelszavát az űrlap mezőiben. 2) A "Bejelentkezés" gombra kattint, amely az authAPI.ts fájlban definiált API hívást indít a /api/auth/login végpontra. 3) Sikeres bejelentkezés esetén a rendszer átirányítja a felhasználót a főoldalra (Home.tsx). 4) Ha a bejelentkezés sikertelen (pl. hibás jelszó), a Notification.tsx komponens egy hibaüzenetet jelenít meg. A bejelentkezési oldal egyszerű felépítésű, a Tailwind CSS segédosztályokkal stílusoztam, például a gombok és az űrlapmezők elrendezéséhez flex, p-6 és bg-gray-100 osztályokat használtam.

A regisztrációs oldal (Register.tsx) hasonló felépítésű, és a RegisterForm.tsx komponenst használja. A regisztráció lépései: 1) A felhasználó megadja az email címét, felhasználónevét és jelszavát. 2) A "Regisztráció" gombra kattint, amely a /api/auth/register végpontot hívja meg. 3) Sikeres regisztráció esetén a rendszer bejelentkezteti és átirányítja a főoldalra. 4) Ha az email vagy felhasználónév már létezik, a Notification.tsx komponens egy hibaüzenetet mutat, például "Az email cím már használatban van". A regisztrációs űrlap validációját a Yup sémák biztosítják, például az email formátumának ellenőrzésére vagy a jelszó minimum 6 karakteres hosszának biztosítására.



21. ábra - Főoldal a bejelentkezett felhasználóknak

A főoldal (Home.tsx) a bejelentkezett felhasználók számára érhető el, és a közelgő meccsek listáját jeleníti meg, amelyeket a /api/football/get/{leagueId}/upcoming-matches végpontból kér le. A használati lépések: 1) A felhasználó kiválaszt egy ligát a menüből, amely a /api/football/get/leagues végpont adatait használja. 2) A kiválasztott liga közelgő meccsei egy táblázatban jelennek meg, amely tartalmazza a csapatok nevét, a meccs időpontját és a fogadás gombot. 3) A felhasználó egy meccs mellett a "Make Bet" gombra kattint, amely egy modalt nyit meg, ahol megadhatja a tippelt eredményt (pl. 2-1). 4) A "Make Bet" gombra kattintva a rendszer a /api/football/bets/match-score végpontot hívja meg, és egy "Fogadás sikeresen leadva" üzenetet

jelenít meg. A főoldal tetején a Header.tsx komponens egy navigációs sávot biztosít, amely tartalmazza a felhasználó nevét, a "Leaderboard" és "Logout" gombokat.

```

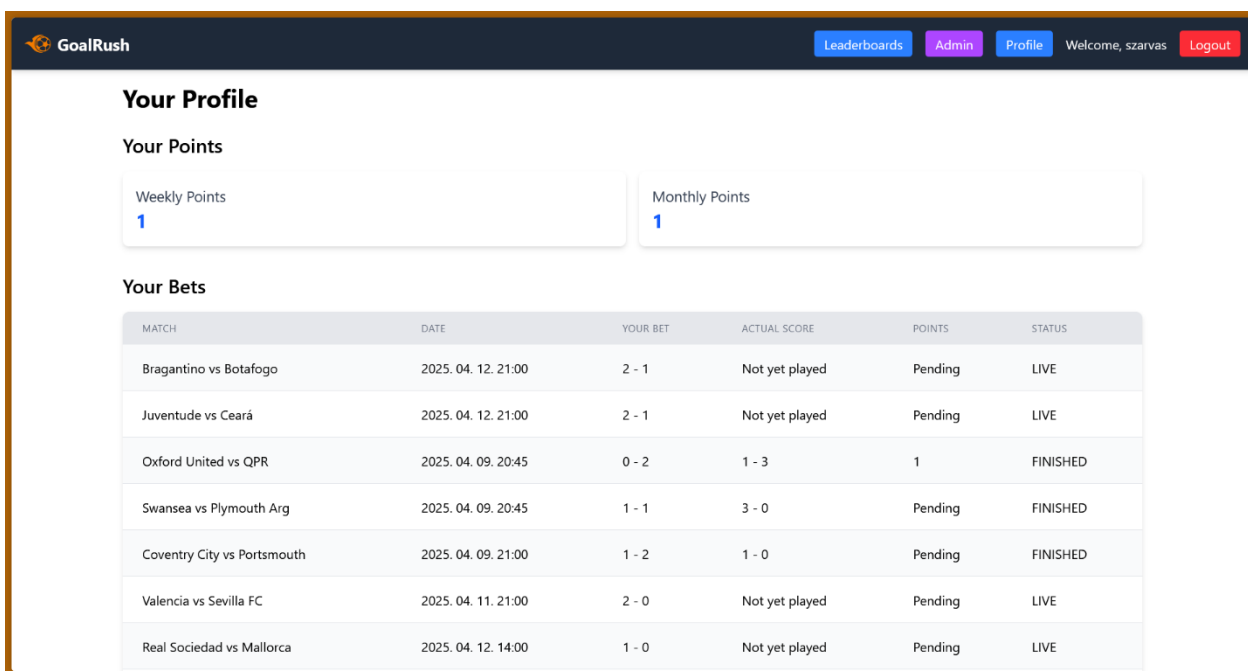
Leaderboards.tsx

<tbody className="divide-y divide-gray-200">
  {data.map((entry, index) => (
    <tr key={entry.userId} className={index % 2 === 0 ? 'bg-gray-50' : 'bg-white'}>
      <td className="px-6 py-4 whitespace-nowrap">{index + 1}</td>
      <td className="px-6 py-4 whitespace-nowrap">{entry.username}</td>
      <td className="px-6 py-4 whitespace-nowrap">{entry.totalPoints}</td>
    </tr>
  ))}
</tbody>;

```

22. ábra - A táblázat adatokat tartalmazó részének kódja

A ranglista oldal (Leaderboards.tsx) a felhasználók heti és havi rangsorát mutatja, amelyeket a /api/leaderboard/weekly és /api/leaderboard/monthly végpontokból kér le. A használati lépések: 1) A felhasználó a navigációs sáv "Ranglista" gombjára kattint, amely a ranglista oldalra irányít. 2) Az oldalon két fül található: "Heti" és "Havi", amelyek között a felhasználó váltani tud. 3) A kiválasztott fül alapján a rendszer megjeleníti a ranglistát egy táblázatban, amely tartalmazza a felhasználók nevét, pontszámát és helyezését. 4) A ranglista oldal egyszerű dizájnnal rendelkezik, a táblázat sorait a egy egyszerű elágazással emeltem ki, ami a páros sorokat szürke, a páratlanokat fehér háttérűvé teszi.



23. ábra - A profil oldal

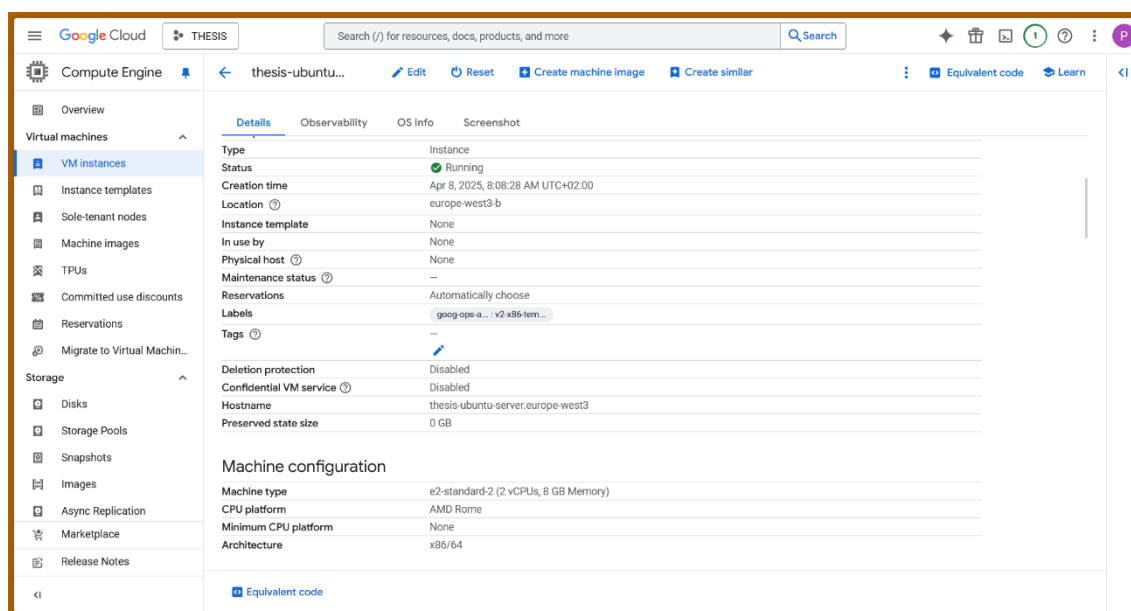


A profil oldal (Profile.tsx) a felhasználó adatait jeleníti meg, amelyeket a `/api/user/userinfo` végpontból kér le. Ezen az oldalon megtalálhatja még a felhasználó a heti és havi pontjait, valamint a fogadásait tekintheti meg, ezeket a már említett végpontokról kér.

A frontend fejlesztése során a hangsúly a funkcionalitáson volt, és bár a vizuális dizájn egyszerű maradt, a felület használható és a projekt céljainak megfelelő. A navigáció zökkenőmentes, a fogadások leadása és a ranglisták megtekintése intuitív, a hitelesítési folyamat pedig az Axios interceptorok és a Zustand store-ok segítségével stabilan működik.

### 3.5 Az elkészült projekt publikálása Google Cloud Platformon keresztül

A projekt publikálása során célom az volt, hogy az alkalmazás elérhető legyen az interneten keresztül, így a felhasználók bárholnan használhassák a rendszert. Mivel a frontend fejlesztése során a hangsúly a backend funkcionalitásának vizuális megjelenítésén volt, szerettem volna egy kis plusz munkát beletenni a projektbe a publikálási folyamat során. Eredetileg a saját házi szerveremen terveztem futtatni az alkalmazást, amely egy költséghatékony megoldás lett volna, azonban a szerverem karbantartásra szorult, így alternatív megoldásként a Google Cloud Platform (GCP) [8] szolgáltatásait vettem igénybe. A GCP ingyenes próbaidőszaka 90 napig biztosít 300 dollárnyi kreditet, amelyet a felhasználó szabadon allokálhat a szükséges szolgáltatásokra, például virtuális gépek, tárhely vagy hálózati erőforrások használatára [44].



24. ábra - A létrehozott virtuális gép technikai adatai

A projekt futtatásához a GCP Compute Engine szolgáltatását választottam, amely lehetővé teszi virtuális gépek (VM-ek) létrehozását és kezelését. Egy Ubuntu 24.04 LTS operációs rendszerrel rendelkező virtuális gépet hoztam létre, mivel ez a disztribúció széles körű támogatást nyújt a szükséges eszközökhöz, például a Dockerhez és az NGINX-hez. A virtuális gép konfigurációjakor egy, az alapértelmezéstől eltérő, több RAM-mal rendelkező e2-standard-2 géptípust választottam, amely az ingyenes próbaidőszak keretein belül is használható, és elegendő erőforrást biztosított a projekt futtatásához (2 vCPU, 8 GB RAM).

A VM létrehozása után a GCP biztosít egy böngészőalapú SSH klienst a szerver eléréséhez, azonban én a PuTTY [45] SSH klienst preferáltam, mivel ez egy kényelmesebb és testreszabhatóbb terminált kínál. A PuTTY használatához a PuTTYgen eszközzel generáltam egy 2048 bites kulcspárt az RSA algoritmus segítségével. A generált publikus kulcsot feltöltöttem a GCP Compute Engine metaadatai közé, míg a privát kulcsot a PuTTY-ban konfiguráltam, így biztonságos SSH kapcsolaton keresztül tudtam elérni a szervert. Ez a megoldás nemcsak kényelmesebb volt, hanem a kulcsalapú hitelesítés révén hasonló biztonságot is nyújtott, mint a böngészőalapú kliens.

A szerver előkészítése során első lépésként telepítettem a Dockert [7], amely lehetővé tette a konténerizált alkalmazások futtatását. A már meglévő docker-compose.yaml fájlt használtam fel, amely definiálta az adatbázis (MongoDB) és a backend szolgáltatások közötti kapcsolatokat. A Docker Compose segítségével egyetlen paranccsal (docker-compose up -d) elindítottam a MongoDB konténert, amely a projekt adatbázisaként szolgált. A backend alkalmazást (Spring Boot) egy screen munkamenetben futtattam, amely egy Linux segédprogram, és lehetővé tette, hogy a folyamat a terminál bezárása után is futva maradjon [46]. A screen használata különösen hasznos volt, mivel a PuTTY kapcsolat megszakadása esetén sem állt le a backend futtatása, és a screen -r paranccsal bármikor vissza tudtam térni a munkamenethez.

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
bcf2c05e276a	mongo-express	"/sbin/tini -- /dock_..."	3 days ago	Up 2 hours	0.0.0.0:8111->8081/tcp, [::]:8111->8081/tcp	mongo-express
fdc7a423797d	mongo	"docker-entrypoint.s..."	3 days ago	Up 3 days	0.0.0.0:9229->9229/tcp, [::]:9229->9229/tcp, 0.0.0.0:27017->27017/tcp, [::]:27017->27017/tcp	mongo

```

root@thesis-ubuntu-server:/home/szarvas# cat /etc/nginx/nginx.conf
user www-data;
worker_processes auto;
pid /run/nginx.pid;
error_log /var/log/nginx/error.log;
include /etc/nginx/modules-enabled/*.conf;

events {
    worker_connections 768;
    # multi_accept on;
}

http {
    ##
    # Basic Settings
    ##

    sendfile on;
    tcp_nopush on;
    types_hash_max_size 2048;
    # server_tokens off;

    # server_names_hash_bucket_size 64;
    # server_name_in_redirect off;

    include /etc/nginx/mime.types;
    default_type application/octet-stream;

    ##
    # SSL Settings
    ##

    ssl_protocols TLSv1 TLSv1.1 TLSv1.2 TLSv1.3; # Dropping SSLv3, ref: P000LE
    ssl_prefer_server_ciphers on;

    ##
    # Logging Settings
    ##

    access_log /var/log/nginx/access.log;

    ##
    # Gzip Settings

```

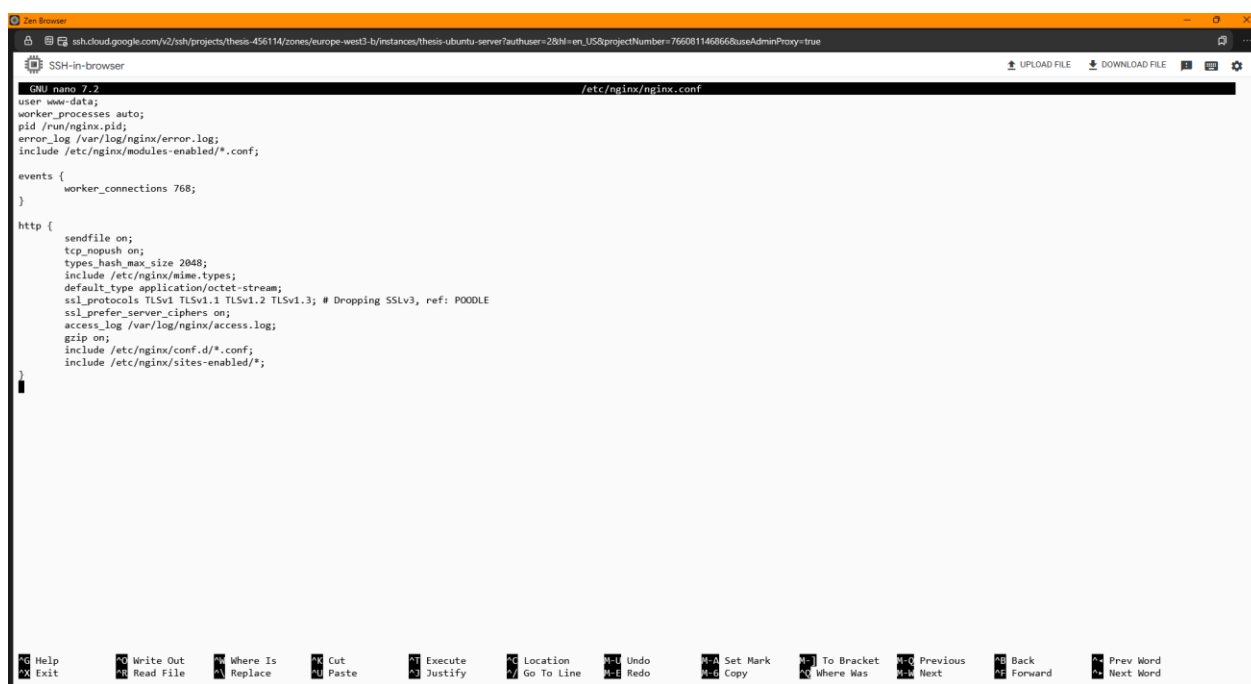
25. ábra - A böngészőalapú SSH kliens

A frontend alkalmazást egy NGINX [47] reverse proxy mögé helyeztem, amely biztosította a HTTP kérések kezelését és a frontend kiszolgálását. Az NGINX konfigurációját úgy állítottam be, hogy a frontend statikus fájljait (amelyeket a vite build paranccsal generáltam) a /var/www/html könyvtárból szolgálja ki. A reverse proxy beállításakor a backend API végpontjait (pl. /api/auth, /api/football) is integráltam az NGINX konfigurációjába, hogy a frontend és a backend közötti kommunikáció zökkenőmentes legyen. Ehhez a következő konfigurációt használtam az NGINX nginx.conf fájljában: a /api útvonalra érkező kéréseket a proxy\_pass direktíva segítségével a



backend localhoston futó portjára (alapértelmezés szerint 8080) irányítottam. Emellett a CORS (Cross-Origin Resource Sharing) beállításokat is módosítanom kellett a backendben, hogy a frontend domainjéről érkező kéréseket a szerver elfogadja. A Spring Boot alkalmazásban a WebSecurityConfig osztályban már definiált CORS konfigurációt finomhangoltam, hogy a frontend domainjét explicit módon engedélyezzem.

A szerver nyilvános elérhetőségének biztosításához a NO-IP [48] dinamikus DNS szolgáltatását használtam, amely lehetővé tette, hogy egy ingyenes domain nevet (pl. goalrush.ddns.net) rendeljek a GCP virtuális gép publikus IP-címéhez. A NO-IP szolgáltatás különösen hasznos volt, mivel a GCP által biztosított IP-cím dinamikusan változhat, és a NO-IP automatikusan frissítette a DNS rekordokat, ha az IP-cím megváltozott. A biztonságos kommunikáció érdekében a Let's Encrypt [49] ingyenes SSL tanúsítványát alkalmaztam, amelyet a Certbot eszközzel telepítettem az NGINX szerverre. A Certbot automatikusan generálta és telepítette a tanúsítványt, valamint beállította az automatikus megújítást, így a HTTPS protokollon keresztül titkosított kapcsolatot biztosítottam a felhasználók böngészője és a szerver között. Az SSL tanúsítvány telepítése után az NGINX konfigurációját úgy módosítottam, hogy minden HTTP kérést HTTPS-re irányítson át, ezzel növelve a kommunikáció biztonságát.



```
GNU nano 2.2 /etc/nginx/nginx.conf
user www-data;
worker_processes auto;
pid /run/nginx.pid;
error_log /var/log/nginx/error.log;
include /etc/nginx/modules-enabled/*.conf;

events {
    worker_connections 768;
}

http {
    sendfile on;
    tcp_nopush on;
    types_hash_max_size 2048;
    include /etc/nginx/mime.types;
    default_type application/octet-stream;
    ssl_protocols TLSv1 TLSv1.1 TLSv1.2 TLSv1.3; # Dropping SSLv3, ref: POODLE
    ssl_prefer_server_ciphers on;
    access_log /var/log/nginx/access.log;
    gzip on;
    include /etc/nginx/conf.d/*.conf;
    include /etc/nginx/sites-enabled/*;
}
```

26. ábra - Az nginx.conf fájl a NANO eszközzel megjelenítve

A publikálási folyamat során felmerült néhány kihívás is. Az NGINX konfigurációjának finomhangolása időigényes volt, különösen a CORS beállítások és a reverse proxy helyes működésének biztosítása terén. Emellett a GCP tűzfal szabályait is módosítanom kellett, hogy a 80 (HTTP) és 443 (HTTPS) portok nyitva legyenek a bejövő forgalom számára, mivel alapértelmezés szerint ezek a portok zárolva voltak. A GCP konzolon keresztül létrehoztam egy új tűzfal szabályt, amely engedélyezte a TCP protokollon keresztül érkező forgalmat ezekre a portokra. A Let's

Encrypt tanúsítvány telepítése során is figyelni kellett arra, hogy a NO-IP domain helyesen legyen konfigurálva, mivel a Certbot a domain elérhetőségét ellenőrzi a tanúsítvány kiállítása előtt.

A projekt sikeres publikálása után az alkalmazás elérhetővé vált a NO-IP domainen keresztül, és a felhasználók biztonságosan, HTTPS protokollon keresztül használhatták a rendszert. A GCP Compute Engine használata lehetővé tette, hogy a házi szerverem karbantartása nélkül is működőképes legyen az alkalmazás, míg az NGINX, a Let's Encrypt és a NO-IP együttes alkalmazása biztosította a stabil és biztonságos működést. A publikálási folyamat értékes tapasztalatot nyújtott a felhőalapú rendszerek konfigurálásában és a szerverüzemeltetésben, amelyeket a jövőbeni projektek során is kamatoztatni tudok.

### 3.6 Használati útmutató

A projekt használata kétféle módon lehetséges: a publikus weboldalon keresztül, amely a Google Cloud Platformon fut, vagy a forráskód klónozásával és helyi futtatásával. Az alábbiakban röviden bemutatom mindkét módszert, hogy a felhasználók könnyen elindíthassák és használhassák az alkalmazást.

Az alkalmazás elérhető egy publikus weboldalon keresztül, amelyet a Google Cloud Platformon publikáltam, és egy NO-IP domain névvel láttam el. A weboldal elérése a következő lépésekkel történhet:

Nyissa meg a böngészőjét, és navigáljon a következő címre: [goalrush.ddns.net](http://goalrush.ddns.net) [50].

A weboldal betöltése után a bejelentkezési vagy regisztrációs oldalon találja magát. Regisztráljon egy új fiókot a "Register" gombra kattintva, vagy jelentkezzen be a már meglévő felhasználónevével és jelszavával a "Login" gombra kattintva.

Sikeres bejelentkezés után a főoldalra kerül, ahol megtekintheti a közelgő meccseket, leadhatja fogadásait, és megtekintheti a ranglistákat a navigációs sáv segítségével.

A weboldal HTTPS protokollon keresztül működik, így a kommunikáció titkosított és biztonságos. A használathoz nincs szükség további telepítésekre, csupán egy modern böngészőre (pl. Google Chrome, Mozilla Firefox).

A projekt forráskódja elérhető a GitHub tárolóban, és a tárolóban megtalálható README.md fájl tartalmazza a helyi telepítési lépéseket.

A két módszer közül a publikus weboldal használata a legegyszerűbb, mivel nem igényel telepítést vagy konfigurációt, míg a helyi futtatás azok számára ajánlott, akik a forráskódot szeretnék tanulmányozni vagy módosítani. Mindkét esetben a felhasználók számára biztosított a teljes funkcionalitás, beleértve a fogadások leadását, a ranglisták megtekintését és a felhasználói profil kezelését.

## 4. Összefoglalás

A szakdolgozatom célja egy online football tippjáték fejlesztése volt, amelyben a felhasználók saját csapatot állíthatnak össze valós játékosokból, és a játékosok valós meccseken nyújtott teljesítménye (például xG, gólok, gólpasszok) alapján pontokat szerezhetnek. A játékosok heti és havi ranglistákon versenyeznek, ahol a cél minél több pont elérése, amiért jutalmakat kaphatnak. Emellett a rendszer lehetőséget biztosít gólokra, gólpasszokra, lapokra és egyéb statisztikákra való tippelésre, amelyek extra pontokat érhetnek. A projekt során egy teljes értékű fullstack webalkalmazást készítettem, amely backend és frontend komponensekből áll, és a Google Cloud Platformon keresztül publikáltam.

A projekt fejlesztése során számos új technológiával ismerkedtem meg, amelyek jelentősen bővítették a tudásomat. A MongoDB használata új terület volt számomra, mivel korábban nem dolgoztam NoSQL adatbázisokkal. A dokumentumalapú struktúra lehetővé tette a rugalmas adatkezelést, például a meccsek és fogadások tárolását, de a sémák tervezése új kihívásokat jelentett. A Spring Boot keretrendszerrel hatékonyan tudtam megvalósítani a backend logikát, például az autentikációt és a fogadások kezelését. A frontend fejlesztéséhez Reactet és TypeScriptet használtam, amelyekkel egy egyszerű, de működőképes felületet készítettem. A Tailwind CSS segített a gyors stílusozásban, míg a Zustand és az Axios a globális állapotkezelést és a backend kommunikációt támogatta. A projekt publikálása során a Google Cloud Platformmal dolgoztam, amely szintén új tapasztalatot jelentett. A virtuális gép konfigurálása, a Docker és NGINX használata, valamint a HTTPS titkosítás beállítása (NO-IP és Let's Encrypt segítségével) értékes ismereteket nyújtott a felhőalapú rendszerek üzemeltetéséről.

A projekt egyik legnagyobb pozitívuma, hogy az eredeti tervekkel ellentétben nem egy egyszerű Java alkalmazás készült, hanem egy teljes értékű fullstack webalkalmazás, amelyet teljesen egyedileg terveztem. A backend stabilan kezeli a felhasználók autentikációját, a meccsek lekérdezését és a fogadások logikáját, míg a frontend lehetővé teszi a regisztrációt, bejelentkezést, fogadások leadását és a ranglisták megtekintését. A webalkalmazás a Google Cloud Platformon keresztül bárhol elérhető, és a HTTPS protokoll biztosítja a kommunikáció biztonságát. A manuális tesztelés Postman és Bruno segítségével alapos ellenőrzést tett lehetővé, míg a Grok 3 által generált unit tesztek és Javadoc dokumentáció felgyorsította a fejlesztést.

Azonban nem minden célkitűzés valósult meg a tervezett ütemezés szerint. Az eredeti tervek alapján a játék tartalmazott volna egy csapatösszerakó funkciót, amelyben a felhasználók valós játékosokból állíthatnak össze csapatot, és a játékosok teljesítménye (pl. xG, gólok, gólpasszok) alapján kapnak pontokat. Emellett a gólokra, gólpasszokra, lapokra és egyéb statisztikákra való tippelési lehetőségeket is szerettem volna implementálni, amelyek extra pontokat értek volna, sikertelen tipp esetén pedig részleges pontvisszatérítést biztosítottak volna a tipp közelségének függvényében. Ezeket a funkciókat azonban nem tudtam megvalósítani, mivel a football-data.org API ingyenes verziója nem biztosított elegendő adatot a játékosok részletes statisztikáihoz, és a lekérdezési kvóta is korlátozott volt. A játékosok inicializálása és értékük meghatározása sem valósult meg, mivel az API nem tartalmazott elegendő információt a játékosok piaci értékéről vagy teljesítményéről. A pontrendszer és a megszerzett pontok elköltésének implementálása (pl. játékosok cseréje, keret fejlesztése) szintén elmaradt, mivel a csapatösszerakó funkció hiánya miatt ezek a mechanizmusok nem voltak értelmezhetők a jelenlegi rendszerben.

A nem megvalósult célkitűzések fontos tanulságot nyújtottak: a külső függőségekkel való munka kockázatos lehet, különösen, ha azok ingyenes verzióira támaszkodunk. A jövőben ezt úgy küszöbölném ki, hogy egy másik API-t választanék, amely részletesebb adatokat biztosít, vagy saját adatgyűjtési mechanizmust építenék ki, például web scraping segítségével. Ez a tapasztalat megtanított arra, hogy a projekt tervezése során alaposabban kell felmérni a külső erőforrások korlátait, és alternatív megoldásokat kell készíteni az ilyen helyzetekre.

A projekt jövőbeli fejlesztési lehetőségei számos izgalmas irányt kínálnak. A backend jelenleg monolitikus architektúrára épül, amely stabil, de nem skálázható hatékonyan nagyobb terhelés esetén. Egy lehetséges fejlesztési irány a monolitikus backend microservice alapú architektúrára cserélése, például egy Eureka szerver használatával, amely lehetővé tenné a szolgáltatások közötti kommunikációt és a terheléelosztást. Ez a megközelítés rugalmasabbá tenné a rendszert, és könnyebbé válna az új funkciók implementálása, például a csapatösszerakó vagy a részletesebb tippelési lehetőségek különálló szolgáltatásokként. A microservice architektúra emellett lehetővé tenné a rendszer skálázhatóságának növelését, például egy API Gateway (pl. Spring Cloud Gateway) integrálásával, amely a kérések kezelését és a terheléelosztást optimalizálná.

A frontend fejlesztése szintén jelentős potenciált rejt. A jelenlegi felület egyszerű, és sokat lehetne javítani a vizuális dizájnban, például egy modernebb UI könyvtár (pl. Material-UI vagy Ant Design) integrálásával. A felhasználói élmény fokozható lenne valós idejű értesítésekkel a meccsek eredményeiről, például WebSocket technológia használatával, amely lehetővé tenné, hogy a felhasználók azonnal értesüljenek a fogadásaik kimeneteléről. Egy másik izgalmas fejlesztési irány egy React Native alapú multiplatform mobilalkalmazás készítése, amely Android és iOS rendszereken egyaránt futna. Ez a mobilalkalmazás szélesebb közönség számára tenné elérhetővé a játékot, és olyan funkciókat is kínálhatna, mint a push értesítések a meccsek kezdéséről vagy a ranglisták frissüléséről. A mobilalkalmazás fejlesztése során a jelenlegi React alapú frontend kód egy része újrafelhasználható lenne, ami felgyorsítaná a fejlesztést.

A publikálási folyamat során a Google Cloud Platform használata ideiglenes megoldás volt, mivel a saját házi szerverem karbantartásra szorult. A jövőben, ha sikerül megjavítanom a szerveremet, szeretném áthelyezni az alkalmazást a saját infrastruktúrára, mivel ez költséghatékonyabb megoldás lenne hosszú távon, és nagyobb kontrollt biztosítana a rendszer felett. Emellett a saját szerver használata lehetővé tenné a hardver testreszabását, például egy erősebb CPU vagy több RAM hozzáadását, ami javítaná a rendszer teljesítményét nagyobb terhelés esetén. A Google Cloud Platform használata azonban értékes tapasztalatot nyújtott, és a jövőben is fontolóra venném a felhőalapú megoldásokat, különösen, ha a játék népszerűsége nő, és nagyobb skálázhatóságra van szükség.

A játék funkcionalitásának bővítése érdekében további ötleteket is fontolóra vennék. Például egy közösségi funkció bevezetése, ahol a felhasználók baráti ligákat hozhatnának létre, és egymással versenyezhetnének, növelné a játék közösségi aspektusát. Egy másik lehetőség egy valós idejű chat integrálása, amely lehetővé tenné a felhasználók számára, hogy a meccsek alatt kommunikáljanak egymással, például megvitassák a tippjeiket vagy a csapatösszeállításokat. A játékosok statisztikáinak részletesebb megjelenítése, például egy grafikonokkal és táblázatokkal támogatott elemző felület, szintén vonzóbbá tehetné a játékot azok számára, akik szeretik a

részletes adatokat böngészni. Végül, a jutalomrendszer bővítése, például virtuális trófeák vagy kitűzők bevezetése a ranglistákon elért helyezésekért, motiválná a felhasználókat a további játékra.

Összességében elégedett vagyok a projekt végeredményével, annak ellenére, hogy nem minden célkitűzés valósult meg. Az alkalmazás alapvető funkcionalitása működik, a felhasználók számára biztosított a regisztráció, bejelentkezés, fogadások leadása és a ranglisták megtekintése, mindezt egy biztonságos, HTTPS protokollon keresztül elérhető weboldalon. A fejlesztés során szerzett tapasztalatok – például a MongoDB használata, a fullstack fejlesztés és a felhőalapú publikálás – jelentősen bővítették a tudásomat, és felkészítettek a jövőbeli projektekre. A külső API-k korlátaiból fakadó kihívások fontos tanulságot nyújtottak, és a jövőbeli fejlesztési lehetőségek (pl. microservice architektúra, mobilalkalmazás, saját szerver használata) izgalmas perspektívát kínálnak a játék továbbfejlesztésére. A projekt során nemcsak technikai ismereteket szereztem, hanem megtanultam, hogyan kell egy komplex rendszert végigvinni a tervezéstől a publikálásig, ami számomra nagy sikerélményt jelentett.

## 5. Irodalomjegyzék

A projektben használt képi anyagok

A weboldal fav ikonjához és kezdőlap ikonjához az SVGREPO-ból töltöttem le, <https://www.svgrepo.com/>

Hivatkozások

- [1] TippmixPRO, <https://www.tippmixpro.hu/>
- [2] MongoDB hivatalos oldala, <https://www.mongodb.com/>
- [3] Spring Boot hivatalos oldala, <https://spring.io/projects/spring-boot>
- [4] React hivatalos oldala, <https://reactjs.org/>
- [5] Git hivatalos oldala, <https://git-scm.com/>
- [6] GitHub, <https://github.com/>
- [7] Docker hivatalos oldala, <https://www.docker.com/>
- [8] Google Cloud Platform, <https://cloud.google.com/>
- [9] Express.js hivatalos oldala, <https://expressjs.com/>
- [10] Java hivatalos oldala, <https://www.java.com/en/>
- [11] Java 21 dokumentáció, <https://openjdk.java.net/projects/jdk/21/>
- [12] Eclipse Temurin, <https://adoptium.net/temurin/>
- [13] Spring Dotenv, <https://github.com/paulschwarz/spring-dotenv>
- [14] Lombok hivatalos oldala, <https://projectlombok.org/>
- [15] TypeScript hivatalos oldala, <https://www.typescriptlang.org/>
- [16] Tailwind CSS hivatalos oldala, <https://tailwindcss.com/>
- [17] Axios hivatalos oldala, <https://axios-http.com/>
- [18] Zustand GitHub, <https://github.com/pmndrs/zustand>
- [19] Formik hivatalos oldala, <https://formik.org/>
- [20] Yup GitHub, <https://github.com/jquense/yup>
- [21] React Router hivatalos oldala, <https://reactrouter.com/>
- [22] A football-data.org hivatalos oldala, <https://www.football-data.org>
- [23] Az eredmények magyar nyelvű oldala, <https://www.eredmenyek.com/>

- [21] Football-data.org API, <https://www.football-data.org/>
- [22] Wayback Machine, <https://archive.org/web/>
- [23] Eredmenyek.com, <https://www.eredmenyek.com/>
- [24] Spring WebClient dokumentáció, <https://docs.spring.io/spring-framework/docs/current/reference/html/web-reactive.html#webflux-client>
- [25] Spring Scheduling dokumentáció, <https://docs.spring.io/spring-framework/docs/current/reference/html/integration.html#scheduling>
- [26] MariaDB hivatalos oldala, <https://mariadb.org/>
- [27] UUID specifikáció, <https://www.ietf.org/rfc/rfc4122.txt>
- [28] BeautifulSoup dokumentáció, <https://www.crummy.com/software/BeautifulSoup/bs4/doc/>
- [29] Scrapy hivatalos oldala, <https://scrapy.org/>
- [30] Cloudflare hivatalos oldala, <https://www.cloudflare.com/>
- [31] Cloudflare Bot Management, <https://www.cloudflare.com/products/bot-management/>
- [32] Cloudflare Canvas Fingerprinting magyarázat, <https://blog.cloudflare.com/canvas-fingerprinting-protection/>
- [33] Spring Security hivatalos oldala, <https://spring.io/projects/spring-security>
- [34] JWT hivatalos oldala, <https://jwt.io/>
- [35] OWASP CSRF dokumentáció, <https://owasp.org/www-community/attacks/csrf>
- [36] BCrypt dokumentáció, [https://www.usenix.org/legacy/event/usenix99/provos/provos\\_html/](https://www.usenix.org/legacy/event/usenix99/provos/provos_html/)
- [37] JJWT GitHub, <https://github.com/jwtk/jjwt>
- [38] Jakarta Validation dokumentáció, <https://jakarta.ee/specifications/bean-validation/>
- [39] Spring Data MongoDB dokumentáció, <https://spring.io/projects/spring-data-mongodb>
- [40] SLF4J hivatalos oldala, <http://www.slf4j.org/>
- [41] xAI hivatalos oldala, <https://xai.ai/>
- [42] Postman hivatalos oldala, <https://www.postman.com/>
- [43] Bruno hivatalos oldala, <https://www.usebruno.com/>
- [44] Google Cloud Free Tier dokumentáció, <https://cloud.google.com/free/docs/gcp-free-tier>
- [45] PuTTY hivatalos oldala, <https://www.putty.org/>
- [46] Screen dokumentáció, <https://www.gnu.org/software/screen/manual/screen.html>

[47] NGINX hivatalos oldala, <https://nginx.org/>

[48] NO-IP hivatalos oldala, <https://www.noip.com/>

[49] Let's Encrypt hivatalos oldala, <https://letsencrypt.org/>

[50] GoalRush oldala, <https://goalrush.ddns.net/>



## 6. Köszönetnyilvánítás

Ezúton szeretnék köszönetet mondani mindazoknak, akik hozzájárultak a szakdolgozatom elkészítéséhez, és támogattak a projekt megvalósítása során.

Különös hálával tartozom témavezetőmnek, Major Sándor Rolandnak, aki végigkísért a szakdolgozat elkészítésének folyamatán. Szakmai útmutatása, türelme és értékes tanácsai nélkül nem tudtam volna ilyen eredményt elérni. Mindig rendelkezésemre állt, amikor kérdéseim voltak, és segített abban, hogy a projekt a megfelelő irányba haladjon, különösen a tervezési és implementációs fázisokban. A tőle kapott visszajelzések és ötletek nagyban hozzájárultak a dolgozat minőségéhez, és inspiráltak arra, hogy a lehető legtöbbet hozzam ki magamból. Köszönöm, hogy támogatta a szakmai fejlődésemet, és végig bizalmat szavazott nekem.

Szeretnék köszönetet mondani Bagosi Dávidnak is, aki szakmai tanácsaival segített a weboldal publikálási folyamatában. Az általa nyújtott iránymutatások és gyakorlati tippek nélkülözhetetlenek voltak az üzembe helyezés során, különösen a szerver konfigurálása és a HTTPS titkosítás beállítása terén.

Végezetül, de nem utolsósorban, hálás vagyok a családomnak a kitartó támogatásukért és biztatásukért. Az ő szeretetük és türelmük adott erőt ahhoz, hogy végigvigyem ezt a hosszú és kihívásokkal teli folyamatot, különösen azokban az időszakokban, amikor nehézségekkel szembesültem. Köszönöm, hogy mindig mellettem álltak, és bátorítottak a céljaim elérésében.