



Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Méréstechnika és Információs Rendszerek Tanszék

Eseményszervező alkalmazás fejlesztése Spring és React platformon

SZAKDOLGOZAT

Készítette
Boros Gergő

Konzulens
Imre Gábor

2022. december 8.

Tartalomjegyzék

Kivonat	i
Abstract	ii
1. Bevezetés	1
2. Felhasznált technológiák	3
2.1. Spring	3
2.2. Spring Boot	4
2.2.1. Funkciók	4
2.3. Spring Data JPA	5
2.4. Spring Security	5
2.5. OAuth2	6
2.5.1. Authorization Code Flow	7
2.5.2. AuthSCH	8
2.5.2.1. Működése	8
2.6. Spring AOP	10
2.7. Spring Test & Mockito	10
2.8. PostgreSQL	10
2.9. Maven	11
2.10. Swagger & OpenAPI	12
2.11. Lombok	13
2.12. Mapstruct	14
2.13. React	14
2.13.1. Single Page Application	14
2.13.2. A React és a SoC	15
2.13.3. React Hooks	15
2.14. npm	16
2.15. Chakra UI	16
2.16. FullCalendar	16
2.17. Docker	17
2.18. Git	18

2.19. GitHub Actions	18
2.20. SonarCloud	18
2.21. IntelliJ IDEA	19
2.22. WebStorm	19
3. A program specifikációja	20
3.1. Funkcionális követelmények	20
3.1.1. Naptár	20
3.1.2. Események megtekintése	20
3.1.3. Esemény létrehozása és módosítása	21
3.1.4. Felhasználók kezelése	21
3.1.5. Események szűrése	22
3.2. Use case-k az alkalmazásban	22
3.3. Nem funkcionális követelmények	23
3.3.1. Docker konténerizáció	23
3.3.2. A kód kezelése	23
3.3.3. Tesztelés	23
4. Tervezés	24
4.1. Az alkalmazás felépítése	24
4.2. Adatbázis	25
4.3. A backend tervezése	27
4.4. A frontend tervezése	28
5. Megvalósítás	32
5.1. Konfiguráció	32
5.1.1. Adatbázis	32
5.1.2. Security	33
5.1.3. Thymeleaf	33
5.1.4. GitHub Actions	34
5.2. Naplózás	35
5.3. Docker konténerizáció	36
5.3.1. Backend	37
5.3.2. Frontend	37
5.3.3. Compose	37
5.4. Bejelentkezés	38
5.5. Szűrők	39
5.6. Naptár megjelenítése	40
5.7. Esemény részletes nézete	41
5.8. Események napi bontás nézete	42
5.9. Esemény létrehozása és módosítása	44
5.10. Éjszakai mód	46

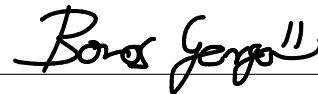
6. Tesztelés	47
6.1. Tesztek fajtái	47
6.1.1. Egység tesztek	47
6.1.1.1. Repository rétegbeli teszt	48
6.1.1.2. Egyszerű kontroller teszt	49
6.1.1.3. Kontroller WebMvc teszt	51
6.1.1.4. Önálló kontroller WebMvc teszt	51
6.1.2. Integrációs tesztek	52
6.1.2.1. Service rétegbeli integrációs teszt	52
6.1.2.2. Egyszerű kontroller integrációs teszt	53
6.1.2.3. Kontroller MockMvc integrációs teszt	54
7. Összegzés	55
7.1. Továbbfejlesztési lehetőségek	55
7.2. Végző	56
Köszönetnyilvánítás	58
Irodalomjegyzék	59

HALLGATÓI NYILATKOZAT

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

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

Budapest, 2022. december 8.



Boros Gergő
hallgató

Kivonat

A szakdolgozati munkám egy fullstack webes alkalmazás fejlesztése volt, amely a BME-VIK Schönherz Kollégiumában élő kollégistáknak nyújt segítséget abban, hogy egy platformon könnyedén értesülhessenek a kollégiumban történő összes közéleti eseményről.

Az elkészült rendszer a Schönherzes kollégisták beléptetését, jogosultságkezelését, események megjelenítését, létrehozását, módosítását, törlését, szűrését, új eseményekről való emailek küldését teszi lehetővé.

A projekt fő célja a már meglévő ismeretek elmélyítése és modern webes alkalmazások során felhasznált technológiákkal kapcsolatos látókör bővítése volt. Törekedtem arra, hogy a lehető legtöbb releváns eszközt felhasználjam az implementáció során, ami egy szoftverfejlesztő számára egy komplexebb webalkalmazás elkészítése során szükséges lehet.

Az alkalmazásom a háromrétegű architektúra elvét követi, miszerint három fő komponenscsaládból áll: a felhasználói felületet megjelenítő, az alkalmazás funkcióinak elérhetőségét biztosító React könyvtárat használó JavaScript-ben írt megjelenítési rétegből, az üzleti logikát megvalósító, a megjelenítési réteggel REST API-t biztosító Java Spring keretrendszerrel írt üzleti logikai rétegből, és az aktuális projekt esetében kevésbé hangsúlyos PostgreSQL-en alapuló adatelérési rétegből.

Dolgozatomban először bemutatom a projekt elkészítése mögött álló motivációt, majd az általam alkalmazott technológiákat, bizonyos aspektusait, illetve azt, hogy miért éppen a használt eszközökre esett a választásom a tervezési folyamat során. Ezek után kitérek a tervezési fázis során használt előzetes ismeretekre, architektúrális elvekre, tervezői döntésekre. Természetesen bemutatásra kerülnek a webalkalmazással szemben felállított követelmények, illetve maga az implementációs folyamat is, melynek célja a rendszer felépülésének, működésének bemutatása, továbbá az érdekesebb, komplexebb komponensek körüljárása. A munka során elengedhetetlen tartomnak a tesztelési folyamatot, amely az utolsó, összegző, reflektáló fejezet előtt kerül szemléltetésre.

Abstract

My thesis work was developing a fullstack web application that helps the college students living in the BME-VIK Schönherz Dormitory, so that they can be easily notified about the social events and gatherings in the dormitory from a single platform.

The finished product is able to handle the admission and authorization of college students of the Schönherz Dormitory, displaying, creating, modifying, deleting, filtering of events and sending notifications of new events.

The main goal of the project was to deepen my existing knowledge and to broaden my horizon about the technologies used to build modern web applications. I tried to use as many relevant tools as possible during the implementation, which may be necessary for a software developer to create a complex web application.

My application follows the principles of three-layered architecture, according to which it consist of three main components: the presentation written in JavaScript using the React library, that displays the userinterface and provides access to the application's functions, the application tier in developed in Java and the Spring Framework, that implements business logic and provides the REST API to the presentation layer, and the data tier based on PostgreSQL, which is less prounounced in the scope of the current project.

In my thesis, I first present my motivation behind the creation of the project, then the technologies I used, certain aspects of them, and the reasons why I chose them during the design process. After that I cover the existing knowledge used in the design phase, the architectural principles and design choices. Of course, the prior requirements set for the web application are presented as well as the implementation process itself, the purpose of which is to describe the structure and operation of the system, as well as descriptions of more interesting and complex components. During my work, I considered the testing process essential which is illustrated before the concluding, reflective chapter.

1. fejezet

Bevezetés

A Budapesti Műszaki és Gazdaságtudományi Egyetem Villamosmérnöki és Informatikai Karának Schönherz Kollégiumának közléte elsőre egy bonyolult rendszerbe tagolódik. "A Ház"-nak titulált Schönherz Kollégiumban számtalan öntevékeny kör működik, amelyek úgynevezett reszortokba rendeződnek. A körök között minden egyetemista megtalálhatja az érdeklődési körének a megfelelőt. A körök rengeteg közösségi eseményt rendeznek az év összes időszakában és ezekről a rendezvényekről a kollégisták és más egyetemi polgárok rengeteg kommunikációs csatornán kapnak értesítést. Levelező listákról, poszterekről, szórólapokról, Facebook eseményeken keresztül, személyes beszélgetésekből. A köröknek, de még a reszortoknak sincs egy bevett kommunikációs platformja, ahova résztvevőket invitálhatnának a megrendezendő eseményeikre.

Ennek a problémának a megoldására jött létre korábban egy webes platform, ahol a körök meghirdethetik az eseményeiket különböző paraméterekkel, és egy átlagos kollégista az oldalt meglátogatva értesülhet róluk. Az oldalt viszont napjainkban csak nagyon kevés kör használja, A Házban megrendezendő események nagyon kis százaléka kerül dokumentálásra az oldalon, és a közelők sem használják, ezért a körök manapság ismét visszatértek ahhoz a praktikához, hogy a fentebb említett csatornák közül a lehető legtöbbet küldenek ki. Ez a habitus rengeteg kénytelen elektronikus levelet jelent egy olyan ember számára, aki nem szeretne ilyen eseményeken részt venni, azonban folyamatosan bombázzák őt a körök ilyen "hirdetésekkel". A probléma a levelező listáról való leiratkozáshoz vezethet, mindazonáltal ez később azt eredményezheti, hogy lemaradhatunk fontos információkról, amely az adott levelező listán kerül publikálásra.

A szakdolgozati témám keresése közben mindig azon gondolkodtam, hogy milyen hasznos dologgal tudnék én hozzájárulni szeretett kollégiumom közösségi életéhez. A emailjeim böngészése, és egy baráti beszélgetés során vetődött fel a téma, hogy lehet újra kellene éleszteni ezt a koncepciót valakinek. Meg is nyitottam ezt a korábbi platformot és kihívásként megfogalmazódott bennem, hogy megpróbálom megépíteni a szóban forgó weboldalt egy teljesen más technológia stack-kel úgy,

hogy a lehető legtöbb olyan eszközt vonom be a fejlesztési folyamatba, amely lefedí egy projekt teljes életciklusát, illetve a későbbi szoftverfejlesztői pályafutásom alatt hasznos lehet és még nem igazán találkoztam vele. Szerettem volna egy tényleg minőségi kész terméket megvalósítani, melynek elkészítése után olyan tudással gazdagodom, hogy egy fejlesztői csapat bármelyik szerepében megállom a helyem. A látókör bővítésének víziója miatt, a szakdolgozatom elkészítése során a legtöbb időt a különböző technológiák megismerése, az irodalomkutatás, és nem a fejlesztés ölelte fel a legtöbb időt, emiatt ezeknek az eszközöknek az ismertetése is fontos szerepet kap jelen dolgozatban.

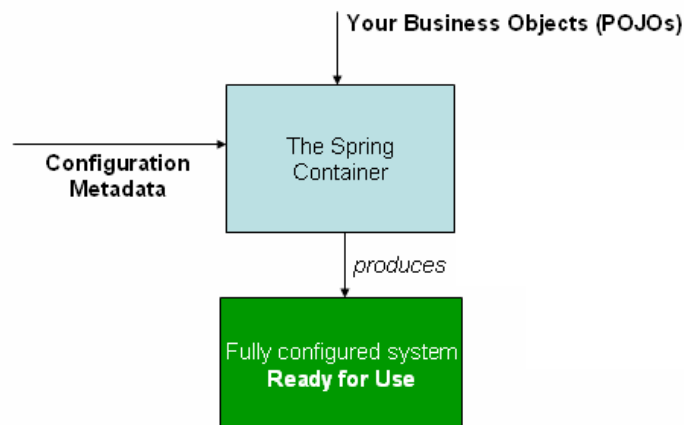
2. fejezet

Felhasznált technológiák

A szakdolgozati munkám tervezési fázisában a technológiák kiválasztásakor a korszerűség mellett másik fő szempontom volt az, hogy minél sokrétűbb, egy alkalmazás fejlesztésének teljes folyamatát lefedő eszközcsoaggal dolgozzak.

2.1. Spring

A Spring a világ legelterjedtebb Java keretrendszere, mely nyílt forráskódú és középpontjában az Inversion of Control (IoC) technika áll. Az Inversion of Control lényege, hogy a programunk komponenseinek életciklusának kezelését, példányosítását, paraméterezését kiemeljük az adott elemünk programkódjából, és egy külső, erre a célra írt másik komponensre, jelen esetben a Spring keretrendszer IoC konténerére bizzuk. A Spring esetében az IoC konténer által menedzselte komponenseket Bean-eknek nevezzük.



2.1. ábra. Az IoC konténer konfigurációs metaadatokkal a POJO-kból (Plain Old Java Object) Beaneket állít elő.
[18]

A Spring keretrendszer segítségével képesek vagyunk hatékonyan, egyszerűen és biztonságosabban magas teljesítményű Java alkalmazásokat írni, melyek fejlesztése

közben kevesebb időt kell töltenünk a konfigurálással. Természetesen lehetőségünk saját magunknak is befolyásolni a konfigurációs beállításokat különböző módokon:

- Java kódban, konfigurációs osztályok segítségével
- Annotációkkal
- XML fájlokon keresztül

Napjainkban a modernebb applikációk és nagyobb volumenű projektek fejlesztése során inkább az annotációkkal és konfigurációs osztályokkal való munka az elterjedtebb, hiszen sok Bean esetében elég sok idő és energiabefektetést igényel XML fájlokon keresztül beállítani Bean-jeinket.

2.2. Spring Boot

A Spring Boot [16] egy olyan nyílt forráskódú projekt, amely a Spring keretrendszerre épül, és könnyedén készíthetünk segítségével Java vagy Kotlin alapú webalkalmazásokat és mikroszolgáltatásokat.

2.2.1. Funkciók

A Spring Boot segítségével teljesen önálló Spring alapú webalkalmazásokat készíthetünk. A beépített Tomcat, Jetty vagy Undertow webszerver segítségével .war fájlok telepítésére nincs szükség. A Spring Initializr [21] segítségével úgy hozhatunk létre egy új Spring Boot projektet, hogy kiválaszthatjuk kedvenc build automatizáló eszközünket, legyen az Maven vagy Gradle, továbbá nyelvet is, amely Java vagy Kotlin lehet. Ezeken felül a Spring Initializr-ben megadhatjuk a projektünk függőségeit is. Maven esetében egy megkaphatunk POM (Project Object Models), Gradle esetében pedig egy build.gradle generált fájlt, mely tartalmazza a kiválasztott dependenciákat és pluginokat.

A Spring Boot [6] használata során az alkalmazások inicializálása előre beállított függőségekkel történik, amelyeket nem kell manuálisan konfigurálni. Automatikusan konfigurálja mind az alapul szolgáló Spring keretrendszert, mind a harmadik féltől származó csomagokat a konfigurációs beállítások és különböző best-practice-ek alapján. Annak ellenére, hogy az inicializálás befejeztével felülírhatjuk ezeket az alapértelmezett értékeket, a Java Spring Boot automatikus konfigurálási funkciója lehetővé teszi a Spring-alapú alkalmazások gyors fejlesztését, és csökkenti az emberi hibák lehetőségét.

A Spring Boot egy "opinionated approach"-al rendelkezik a konfigurációt illetően. A projekt igényei alapján a fejlesztő kiválaszthatja, hogy a több mint 50 féle Spring Starters függőség közül, mely csomagokat telepítse, a Spring Boot pedig alapértelmezett konfigurációt kínál, ahelyett, hogy megkövetelné, hogy ezeket

a döntéseket a fejlesztő saját maga hozza meg, és mindent manuálisan állítson be. Természetesen a lehetőség meg van ezeket felül írni, egyesével bekonfigurálni a számunkra szükséges csomagokat, de a Spring megközelítése nagyban felgyorsítja egy projekt inicializációs fázisát.

2.3. Spring Data JPA

A Spring Data JPA [17] része a nagyobb Spring Data eszközcsaládnak, amely esernyője alá vonja a különböző adathozzáférési technológiákat. A szakdolgozatomban a Spring Data JPA-t használtam, mely lehetővé tette az alkalmazáson belüli adatbázissal való kommunikációt Data Access Object-eken (DAO vagy Repository) keresztül. Ezek a DAO-k egy adathozzáférési réteget alkotnak és különböző CRUD (Create Read Update Delete) metódusokat tartalmaznak, melyekkel manipulálhatunk az adatbázisunkban található adatokat.

A Java Persistence API (JPA) a legelterjedtebb Java programozási interfész, amely lehetővé teszi a fejlesztők számára az ORM (Object-Relational Mapping) koncepciójának használatát. Az ORM lényege, hogy a Java objektumok és az adatbázisok bizonyos tábláiban található értékek megfeleltethetők legyenek egymásnak. A JPA lehetővé teszi a relációs adatbázisunk és objektum-orientált Java programunk közötti objektum konvertálást mindkét irányban.

A Spring Data JPA egyfajta absztrakciós réteget helyez a JPA-ra, a DAO osztályainknak egyszerűen implementálnia kell a JpaRepository interfészt, és máris előre meghatározott CRUD metódusokkal gazdagodott az adott Repository-nk.

Természetesen saját magunk által meghatározott adatmanipulációs függvény írására is adott a lehetőség kettő módon [20]:

- Manuálisan deklarált lekérdezés
 - Egy @Query annotációt kapcsolva a metódushoz saját JPQL lekérdezést írhatunk, melyet a függvény meghívásakor a Spring Data JPA befuttat az adatbázisunkba és visszatér az eredménnyel.
- Név alapján automatikus lekérdezés generálás
 - A Spring Data JPA a deklarált metódus nevéből megpróbálja kitalálni, hogy milyen típusú adatbázis műveletre van szükségünk.

2.4. Spring Security

A Spring Security [19] egy hatékony és nagymértékben testreszabható hitelesítési és hozzáférés-felügyeleti keretrendszer. Gyakorlatilag a szabvány a Spring alapú alkalmazások biztosítására.

A Spring Security egy olyan keretrendszer, amely a Java alkalmazások autentikációjának és autorizációjának biztosítására összpontosít. Mint minden Spring projekt, a Spring Security igazi ereje abban rejlik, hogy rendkívül egyszerűen bővíthető, hogy megfeleljen az egyéni követelményeknek.

2.5. OAuth2

A szakdolgozatomban elkészített projekt egyik lényeges eleme az autorizáció, hiszen nem szeretnénk, ha bárki képes lenne egy kör nevében eseményt létrehozni a weboldalon. A Schönherz Kollégiumban tevékenykedő Kollégiumi Számítástechnikai Kör által készített AuthSCH segítségével gördülékeny implementálható volt egy OAuth2 rendszer az alkalmazásom backend moduljába.

Az OAuth2 [2] egy ipari autorizációs protokoll szabvány. Az OAuth2 lehetővé teszi, hogy harmadik félhez tartozó alkalmazásoknak korlátozott hozzáférést kapjanak felhasználói fiókokhoz. Úgy működik, hogy a felhasználói hitelesítést delegálja a felhasználói fiókot üzemeltető szolgáltatásra, és harmadik féltől származó alkalmazások számára engedélyezi a felhasználói fiók elérését. Az OAuth2 csupán a jogokat azonosítja, amivel a bejelentkezni kívánó felhasználó rendelkezik, nem a felhasználót, ezért nem nevezhető autentikációs szolgáltatásnak.

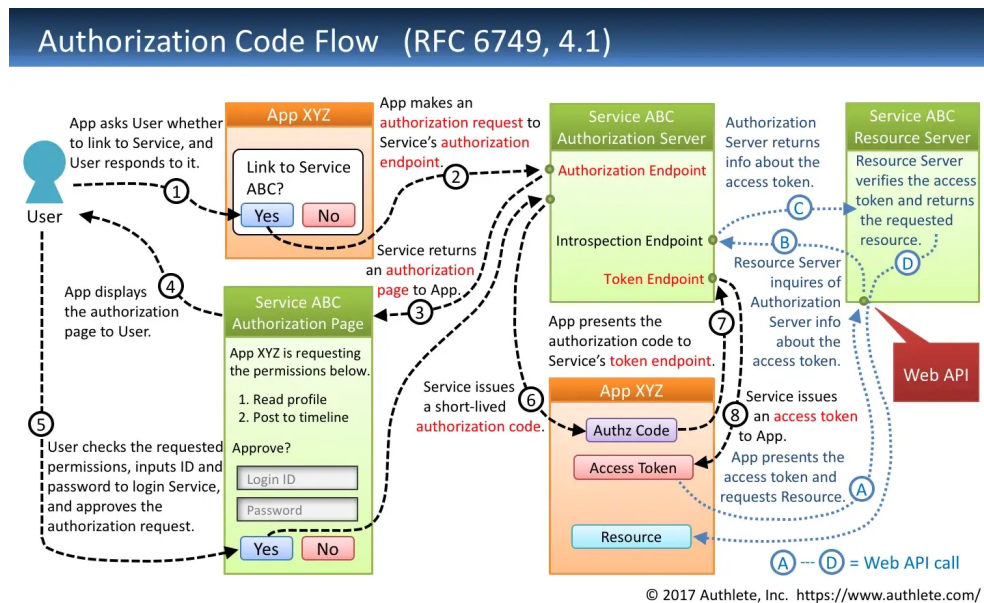
A szakdolgozatom során az alábbi OAuth2-vel kapcsolatos fogalmakkal kellett megismerkedjek:

- **Az OAuth által meghatározott 4 szereppel:**
 - **Resource owner:** Olyan entitás, amely képes hozzáférést biztosítani egy védett erőforráshoz. Ha az erőforrás tulajdonosa egy személy, azt végfelhasználónak nevezzük (end-user). A szakdolgozatom esetében ez az alkalmazást használó felhasználó.
 - **Resource server:** A védett erőforrásokat tartalmazó szerver, amely képes elfogadni és válaszolni ezekkel a védett erőforrásokkal kapcsolatos kérések-re Access Tokeneket használva.
 - **Client:** Egy applikáció, amely képes védett kéréseket intézni a védett erőforrás tulajdonosa személyében és autorizációjával. A "Client" nem specifikálja, hogy milyen típusú kliensről van szó, lehet egy asztali applikáció, egy program, ami egy szerveren fut, vagy más eszköz.
 - **Authorization Server:** A szerver, amely kibocsátja az access tokeneket a Client-nek a Resource Owner sikeres autentikációja és autorizációja után.
- **Authorization Grant:** Az Authorization Grant a resource owner védett erőforrásához szükséges autorizációs jogot jelképezi, amit a Client használ, hogy Access Token-t szerezhessen az Authoriation Server-től.

- **Authorization Code:** A Resource Owner jóváhagyása után a Client ezzel juthat hozzá a védett erőforrások eléréséhez szükséges tokenhez, melyet az Authorization Server ad ki.
- **Access Token:** Egy Client ezzel a kóddal érheti el a Resource Server-től a védett erőforrásokat.
- **Refresh Token:** A Refresh Token-t új Access Token szerzésére használják. Az Authorization Server bocsájtja ki a Client-nek, ami új Access Token-ek lekérésére használhatja, ha a korábbi már lejárt vagy érvénytelen.

2.5.1. Authorization Code Flow

Az autorizációs folyamatot egy fejlesztő a tervezés során gondosan megválaszthatja, ugyanis az OAuth2 erre több megoldást, folyamatot is ad. A saját projektem során az Authorization Code Flow-t használtam.



2.2. ábra. Authorization Code Flow

[24]

A folyamat egyszerűen leírható a dolgozatom esetében az alábbi lépésekben:

1. A Resource Owner (jelen esetben az alkalmazást használó felhasználó, kolléga) belép az alkalmazásba és elindítja az autorizációs folyamatot.
2. Az alkalmazás továbbirányítja őt egy oldalra, ahol engedélyezheti az alkalmazásnak a kért erőforrások elérését.
3. Az engedélyezés után az Authorization Server továbbítja a Client felé az Authorization Code-ot.
4. A Client elküldi a kapott Authorization Code-ot az Authorization Server-nek, hogy hitelesítje. Ha érvényes a kapott kód a szerver visszaküldi az Access Token-t a Client-nek.

5. A Client elküldi a hozzáférni kívánt védett erőforrások listáját a kapott Access Token-nel együtt a Resource Server-nek.

6. A Resource Server ellenőrzi az Access Token érvényességét, és amennyiben az érvényes, akkor elküldi a kérvényezett védett erőforrásokat a Client-nek.

2.5.2. AuthSCH

Az AuthSCH [8] rendszer azzal a céllal készült, hogy a VIK-en elterjedt 3 nagy beléptetőrendszert (BME címtár, SCH Account, VIR Account) egységesítse és a felhasználó számára egyszerűen kezelhetővé tegye a belépési folyamatot, anélkül hogy gondolkodnia kellene, melyik felhasználóját használja belépésre az adott felhasználói felületen. Az AuthSCH-t a kollégiumunk közéletének szinte minden területén használják.

2.5.2.1. Működése

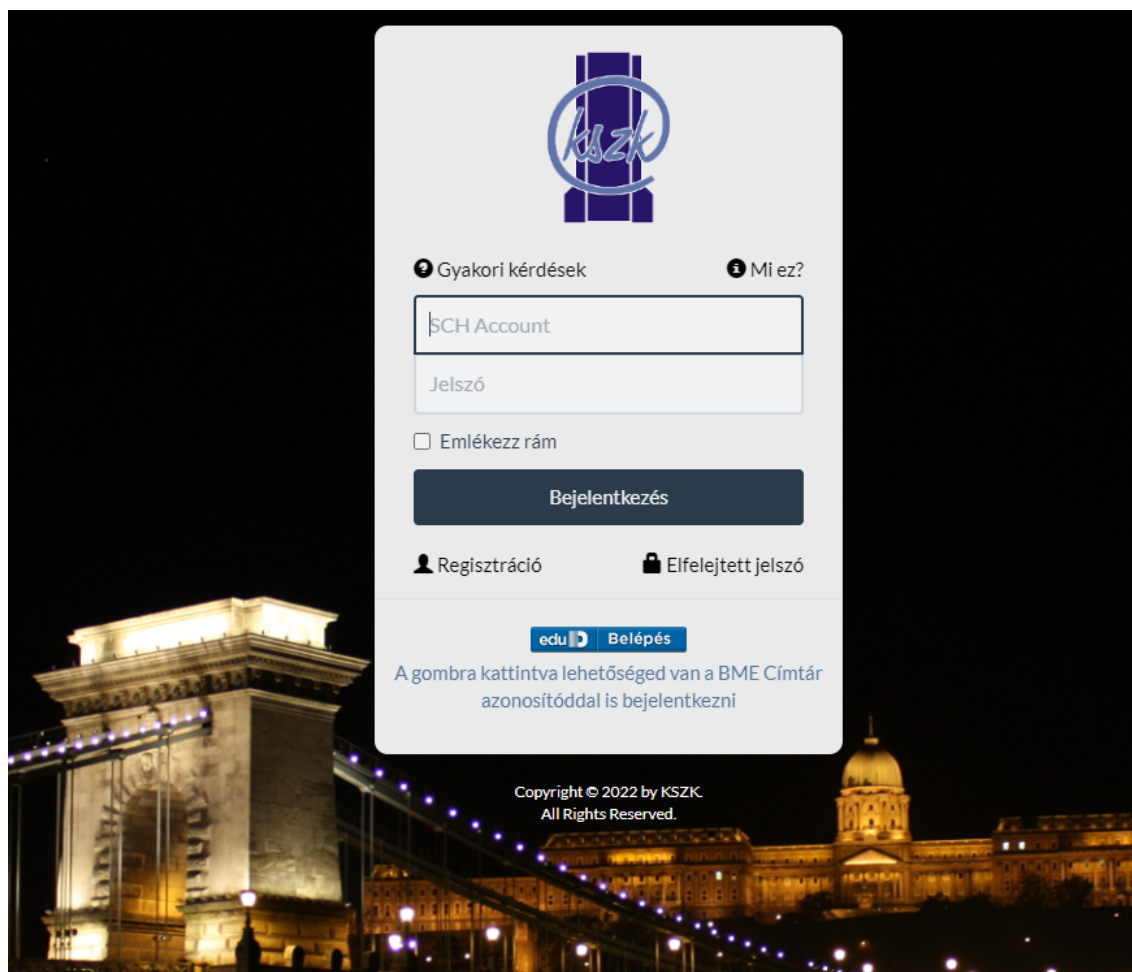
Az `https://auth.sch.bme.hu/console/index` címen be kell regisztrálnunk az alkalmazásunkat, hogy tudjuk használni a saját szoftverünkben. A regisztráció után az AuthSCH egy kliens azonosítót rendel az alkalmazáshoz, melyet a beléptetési oldalra való átirányításkor használunk az alábbi módon:

```
https://auth.sch.bme.hu/site/login?response_type=code&client_id=<kliens azonosító>&state=<felhasználó  
ra jellemző egyedi azonosító>&scope=<jogosultságok>
```

- **client_id:** a korábban említett, AuthSCH-től kapott alkalmazás kliens azonosítója
- **state:** ezzel a paraméterrel tudjuk biztosítani, hogy az autorizációs folyamatba ne tudjon beférközni egy harmadik fél, a jelen projekt esetében ez egy felhasználót egyértelműen azonosító UUID
- **scope:** az AuthSCH által támogatott típust vagy scope-ot írja le, igazából különböző típusú adatlistákat határoz meg, a dolgozatom során az alábbiakat használtam:
 - **basic:** a felhasználó AuthSCH azonosítója
 - **displayName:** a felhasználó neve (vezetéknév és keresztnév)
 - **mail:** a felhasználó e-mail címe

Minden oldalra való első belépéskor az AuthSCH a belépési képernyő előtt egy engedélykéréssel fordul a kliens felé, amelyben kilistázza, hogy milyen jogokhoz, milyen scope-ban szeretne hozzáférni az adatainkhoz az alkalmazás, a felhasználó pedig eldöntheti, hogy megadja a jogokat az alkalmazásnak, vagy sem.

Későbbi bejelentkezések alkalmával a felhasználó egy belépési képernyővel szembesül, ahol megadhatja a felhasználó nevét, jelszavát, vagy akár a BME-n használt eduID beléptető rendszert is használhatja.



2.3. ábra. Az AuthSCH bejelentkezési képernyője

Az oldal a beléptetés után visszairányítja az AuthSCH-ban általunk megadott oldalra és GET paraméterben vissza adja a már korábban taglalt Authorization Code-ot. Ezek nincs más dolgunk, mint meghívni a token végpontot a `https://auth.sch.bme.hu/oauth2/token` címen és POST paraméterként átadni a következő két értéket:

```
grant_type=authorization_code&code=<a kapott Authorization Code>
```

Erre válaszul megkapjuk az Access Token-t a következő formában:

```
{
  "access_token": "6f05ad622a3d32a5a81aee5d73a5826adb8cbf63",
  "expires_in": 3600,
  "token_type": "bearer",
  "scope": "<jogosultságok>",
  "refresh_token": "e8abb8a1640cce4b4d9b6162d59cd50a2e3f4de"
}
```

Végezetül pedig az Access Token azért kiemelten fontos, mert az Access Token által azonosított jogosultságai alapján adatik meg a lehetősége egy beléptett felhasználónak, hogy tud-e új eseményeket létrehozni, vagy már meglévőket manipulálni különböző módokon a felületen.

2.6. Spring AOP

Az objektumorientált programozást (OOP) kiegészítő Aspektus-Orientált programozás (AOP) is kitüntetett szereppel rendelkezik a Spring ökoszisztémában [15]. Az AOP egy másik gondolkodási módot nyújt számunkra a fejlesztés során. Az objektumorientált programozásban egy egységet egy osztályt számítunk, az aspektusorientált programozás esetében, pedig több osztály is alkothat egy egységet, feltéve ha hasonló aspektusokkal rendelkeznek. Az AOP fő célja a modularitás növelése a Separation Of Concerns elv alapján, miszerint a programot külön komponensekre bontjuk fel úgy, hogy az egyes komponensek külön vonatkozásokat fedjen le. Az üzleti logika ezáltal elválk a keresztivatkázásoktól (cross-cutting concern), melynek egy tipikus példája a naplózás.

2.7. Spring Test & Mockito

A Spring keretrendszer IoC konténere sokkal könnyebbé teszi az egység és integrációs tesztek írását a Java applikációnk fejlesztése során, és a Spring Testing az a modul, amely tartalmazza a különböző típusú tesztek megírásához szükséges csomagokat.

A Mockito egy mocking keretrendszer, mely segítségével könnyen tudunk egység teszteket írni Java alkalmazásunkhoz, és mockolni tudjunk interfészeket és objektumokat, szóval egyfajta mű, "dummy" funkcionalitás adható hozzá ezekhez a mockokhoz, amit aztán egység tesztekben használni tudunk.

A Spring Testing és Mockito modulokat ötvözve készítettem el az alkalmazásom tesztjeit is, mely közé tartoznak egység, integrációs és különböző rétegbeli tesztek is. A dolgozatomban a tesztelésnek egy külön fejezetet szenteltem, ahol bővebb bemutatásra kerülnek az imént leírt tesztelési technológiák segítségével megírt implementáció.

2.8. PostgreSQL

A PostgreSQL [11] egy nyílt forráskódú relációs adatbáziskezelő rendszer. Más adatbázisokhoz hasonlóan a non-procedurális SQL nyelvet használja alapjaként és sok modern funkciót támogat:

- komplex lekérdezések
- idegen kulcsok
- triggerek
- frissíthető nézetek
- tranzakciós integritás

- verziókezelés időbélyegek mellett (MVCC)

Az eléggé liberális licenc miatt pedig a PostgreSQL-t bárki ingyenesen használhatja, módosíthatja és terjesztheti bármilyen célra, legyen az privát, kereskedelmi vagy tudományos

A jelenlegi szakdolgozat feladatának kontextusában pedig elég a Maven POM fájlában dependenciaként deklarálni, majd megfelelően konfigurálni egy properties fájlban, és már is képes a Spring Boot projektünk csatlakozni JDBC-n keresztül a lokálisan futó adatbázis szerverhez.

2.9. Maven

A Maven [22] egy parancssori build automatizáló eszköz, amely igen elterjedt és könnyedén segít áthidalni a mindennapi problémáinkat a fejlesztés során. Rengeteg előnye közé tartozik, hogy plugineket importálhatunk be, erősen testreszabható, képes függősek letöltésére, akár tranzitívan is, továbbá build szerverek (például Jenkins) is tud Mavent használva buildelni.

A konfigurációs fájl a `pom.xml`, amely a projekt gyökerében található. Ennek főbb, a projektben is használt elemei:

- `<groupId>`, `<artifactId>`: azonosítják a projektet
- `<version>`: verzió
- `<dependencies>`: függőségek
- `<build>`: a build testreszabása, jellemzően pluginekkel
- `<properties>`: konfigurálható placeholderek

A projektünkben függőségeket deklarálhatunk a következő módon:

```
<dependency>
  <groupId>org.projectlombok</groupId>
  <artifactId>lombok</artifactId>
  <version>1.18.22</version>
</dependency>
```

A különböző Maven projektek `.jar` fájllai, plugin-jai, vagy artifact-jei a Maven Repository-ban tárolódnak. Ez a könyvtár lehet lokális, a felhasználó számítógépén, vagy a Maven által központi repository, ahol az npm-hez hasonlóan rengeteg gyakran használt könyvtár megtalálható.

A függőségeket a Maven először a lokális repository-ból próbálja meg beszerezni, aztán a központi (central) repositoryból tölti le, és mivel ezek a függőségek is rendelkezhetnek `pom.xml`-el, ezért tranzitívan telepítődik a többi függőség is.

A central és local repository-n kívül egyéb remote repository-kat is deklarálhatunk az alábbi módon:

```
<repositories>
  <repository>
    <id>my-internal-site</id>
    <url>http://myserver/repo</url>
  </repository>
</repositories>
```

A Maven a build életciklus koncepciója köré van szervezve. Ez azt jelenti, hogy a buildelési, disztribúció folyamata egy artifactnak előre meghatározott. Egy embernek elég megtanulnia néhány Maven parancsot, hogy akármilyen Maven projektet lebuildeljen vagy akár deployoljon, és a `pom.xml` fájl biztosítja, hogy a felhasználó azt az eredményt éri el, amit elvárt. A build életciklus az alábbi fázisokból épül fel:

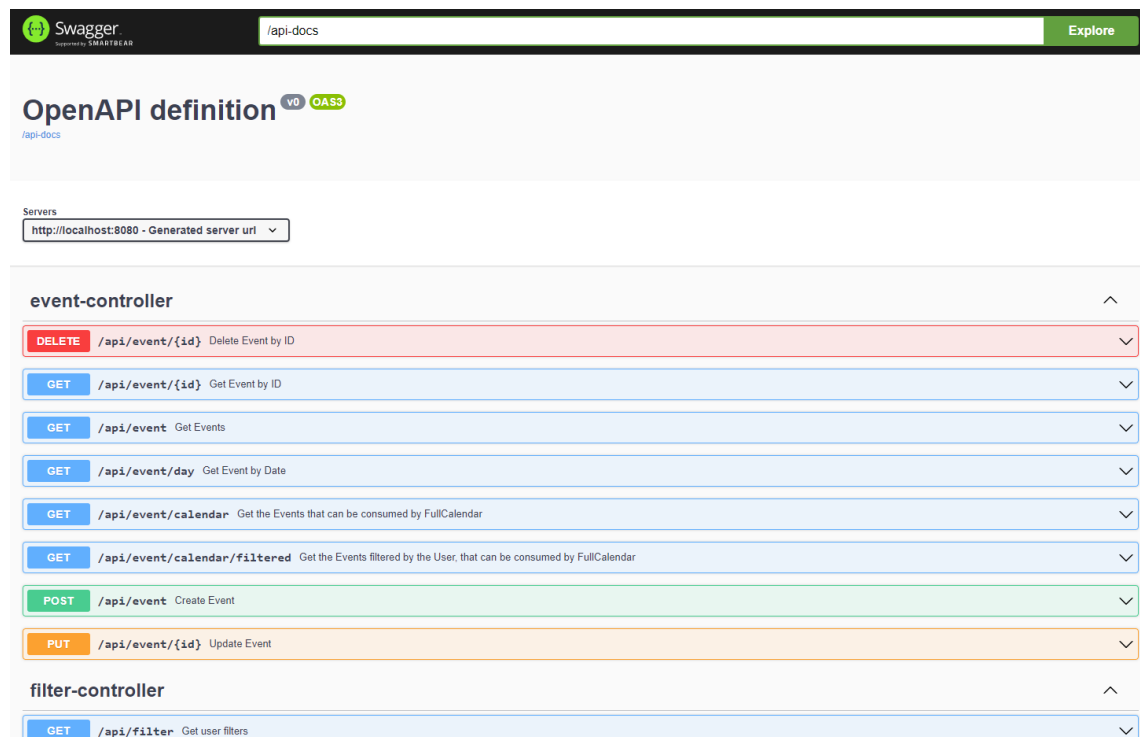
- `validate`: projekt validálása
- `compile`: `src/main/java` alatti Java fájlok fordítása
- `test`: unit tesztek lefuttatása (hiba esetén leáll)
- `package`: a `main`-ből fordított class fájlok és a `src/main/resources` (`.war` esetén `src/main/webapp` is) alatti fájlok csomagolása `.jar/war` formátumba
- `integration-test`: integrációs tesztek előkészítése (teszt szerverre való telepítéskor használandó)
- `verify`: minőségi ellenőrzés
- `install`: kész termék másolása a local repositoryba
- `deploy`: kész termék feltöltése egy távoli repositoryba

Minden kimeneti fájl, legyen az `.jar` vagy `.war` a projekt *target* könyvtárba kerül. Az `mvn clean` paranccsal viszont törölhetjük ezt, illetve a verziókezelőben is érdemes ezeket a generált fájlokat ignorálnunk.

2.10. Swagger & OpenAPI

Az OpenAPI [23] Specification (OAS) egy olyan ipari nyelv-agnosztikus sztenderd interfészt definiál RESTful API-k (Representational State Transfer, Application Programming Interface) számára, amely lehetővé teszi, hogy emberek és számítógépek is megértsék egy végpontokkal definiált szolgáltatáscsomag funkcionalitását anélkül, hogy elérhető lenne a forráskódja, dokumentáció, vagy megvizsgálnánk a hálózati forgalmat. Helyesen definiálva a felhasználó megértheti, és használhat egy távoli szolgáltatást kevés implementációs logika ismeretével is.

Egy OpenAPI definíció használható dokumentáció generációs eszközök által, hogy leírják egy API funkcionalitását, továbbá kódgenerációs eszközök szervereket és klienseket generáljanak sokféle programozási nyelven. Formátumát könnyű megérteni, YAML-ben (Yet Another Markup Language/YAML Ain't Markup Language) vagy JSON-ben (JavaScript Object Notation) íródik. A Swagger az OpenAPI specifikáció köré épülő nyílt forráskódú eszközök készlete, amely segíthet a REST API-k tervezésében, felépítésében, dokumentálásában és felhasználásában.



2.4. ábra. A szakdolgozatom Swagger UI által generált REST API dokumentációja.

A Swagger UI segítségével vizualizálhatjuk és interaktálhatunk egy API végpontjaival, anélkül, hogy akármilyen implementációs logikát is ismernénk. Automatikusan generálódik az OpenAPI specifikációból. A jelen projekt esetében különböző annotációkkal kellett ellátnom a REST API-m végpontjait, hogy a Swagger UI helyesen legenerálja számomra a képen is látható interaktív dokumentációt.

2.11. Lombok

A Lombok egy olyan annotáció alapú Java plugin, amely segít lecsökkenteni a repetitív, unalmas, boilerplate kód írását az alkalmazásunk fejlesztése során. Tipikusan példa *getter*, *setter*, *toString* metódusok és különböző konstruktorok helyettesítésére szolgál. Integrálható különböző fejlesztői környezetekbe és build eszközökbe is.

2.12. Mapstruct

A Mapstruct [9] kód generátor, ami nagy mértékben leegyszerűsíti a leképezések, mappolások implementációját Java Bean típusok között. A generált mapping kód sima metódus invokációkat tartalmaz ezért gyors, típusbiztos és könnyű megérteni.

Több rétegű applikációk esetén szükséges lehet a leképezés az entitás modellek között, például REST-ful webszolgáltatások esetén az entitások és Data Transfer Object-ek (DTO) között. Erre a leképezésre azért van szükség, mert egy entitás esetében a frontendnek nem szükséges minden olyan adat, amit a backend felhasznál. A mapping kód megírása hosszú, és könnyen elvéthető feladat lehet, amit a Mapstruct automatizál helyettünk. Más mapping keretrendszerekkel ellentétben a Mapstruct fordítási időben működik, ami biztosítja a nagy teljesítményt, és a gyors fejlesztői visszajelzést a könnyű hibaellenőrzés által.

A Mapstruct egy annotációs processzor, ami a Java fordítóba épül és parancsori build automatizáló eszközök esetében is használhatjuk, például Mavenben vagy Gradleben.

2.13. React

A React [13] egy JavaScript könyvtár, melyet felhasználói felületek készítésére használnak. A React segítségével az alkalmazásunk fejlesztésekor kisebb komponenseket írunk meg, melyek applikációtól függetlenül újrahasznosíthatóak, a saját állapotukat kezelik, továbbá ezekből a kisebb komponens alapú építőeszközökből egy komplexebb felhasználói felületet is építhetünk. Ez az architektúra megkönnyíti a fejlesztő életét, hiszen egy webalkalmazás fejlesztésének folyamat során újra és újra már ismerős, visszatérő komponensek kerülnek használatra a funkciók implementálása során. A hibakeresés sem bonyolult egy React program esetében, a komponensek elszigeteltsége miatt egyértelműen megállapítható, hogy melyik építőelem nem funkcionál helyesen.

2.13.1. Single Page Application

Az SPA-k, vagy Single Page Application-ök olyan webes alkalmazások, amelyek a nevükből adódóan is egyetlen egy oldalból állnak. Egy Single Page Application megnyitásakor a felhasználó böngészője előre letölti az adott oldalt meghatározó JavaScript fájlokat (single page load), majd futtatja őket a böngésző a beépített JavaScript motorjával. Az oldal tartalma (HTML, CSS, JavaScript kód) dinamikusan változik a felhasználó interakciói alapján. A Single Page Application központi elmélete, hogy az oldalon történő átmenetek simábbak, gyorsabbak legyenek, ezáltal hasonlóbb érzetet adjanak, mint a natív alkalmazások. A React segítségével is ilyen SPA tervezési mintát követő felhasználói felülettel rendelkező webalkalmazások készíthetők.

2.13.2. A React és a SoC

Korábbi szekcióban szerepelt a Separation of Concerns elmélete, azonban a React kicsit unortodox megoldást követ a probléma körüljárása során. Más webes keretrendszerek esetében népszerű, hogy szétválasztják egy komponens, oldal megjelenítési és üzleti logikáját. Az Angular tipikus fent tart egy `.html` fájlt a megjelenítésnek és egy `.ts` fájlt a komponenshez tartozó működési logikának. A React esetében ez a két funkcionalitás egy komponensben valósul meg. Egy komponens viszont csak a saját magát érintő funkcionalitásokkal foglalkozik, tehát igazából a "Concern" vonal meghúzása más irányban történik meg egy React alkalmazás esetében. A React-ben JSX (JavaScript eXtension) kódot használhatunk oldalunk felhasználói felületének leírására. A JSX kód nagyjából egy az egyben hasonlít a HTML kódra, azzal a különbséggel, hogy később sima JavaScript-re fog fordulni.

2.13.3. React Hooks

A React Hook-ok [12] a 16.8-as frissítésével kerültek be a React ökoszisztémájába. Eddig, amikor komponensekről beszéltünk, nem specifikáltuk, hogy egy komponens milyen típusú objektum. A frissítés előtt kizárólag osztály alapú komponensek építésére volt lehetőség, azonban ma már a hook-ok segítségével függvények (function) is helyettesíthetők őket. A motiváció a Hook-ok mögött az, hogy sok olyan problémát megoldanak a fejlesztés során, ami előjön egy nagyobb alkalmazás fejlesztése és fenntartása során.

Az osztályok használata közben eddig rengetegszer duplikált kód került megírásra, ugyanis nagyon nehéz volt állapottartó logikát átvinni egyik komponensből a másikba, vagy általában ez a komponens hierarchiánk megváltoztatásával járt, ami egy fájdalmas és hosszadalmas művelet.

A fejlesztőknek releváns probléma volt még, hogy a komplex osztályok egy idő után nagyon felduzzadtak, érthetetlenek, bonyolultak lettek, olyan szinten, hogy egy komponens életciklusainak metódusaiban egymástól teljesen független függvényhívások, alkalmazáslogika jelent meg. Ezek a problémák bug-ok, hibák, mellékhatások megjelenéséhez vezettek.

Az állapottartó logika átvitelének nehézsége miatt pedig nehéz volt ezeket a nagyra nőtt komponenseket kisebb egységekre bontani, vagy akár tesztelni őket. Az emberek elkezdtek különböző állapotkezeléssel foglalkozó könyvtárakat használni, például a Redux-ot, de ezek általában túl nagy absztrakciót vezettek be, rengeteg fájl között kellett zsonglörködni, hogy működésre bírjuk őket, és igazából nem oldották meg a komponens újrahasználatosság problémáját. A Hook-ok segítségével egy komponens feldarabolhatunk kisebb részekre, az alapján, hogy éppen az adott résznek, milyen logikához van köze, ahelyett, hogy ezt az elválasztást az életciklus metódusokhoz kötnénk.

2.14. npm

Az npm [10] a JavaScript legismertebb és legelterjedtebb függőség- és csomagkezelő és nyilvántartó platformja. Minden csomagot egy `package.json` fájl ír le JSON formátumban. Az npm használata során az applikációnk projektkönyvtára is tartalmaz egy `package.json` fájlt, mely leírja `<név>:<verzió>` formátumban a programunk függőségeit.

Egy CLI-n (Command Line Interface) keresztül interaktálhatunk az npm szoftverkönyvtárával. Az `npm install` parancs kiadásával az npm megkeresi a `package.json`-ben leírt függőségeinket a registry-ben és telepíti azokat. Az alkalmazásunkkal is interaktálhatunk az npm-en keresztül, fordíthatjuk, elindíthatjuk, teszteket futtathatunk, továbbá előre specifikált futtatható scripteket is írhatunk a `package.json`-ben.

2.15. Chakra UI

A Chakra UI egy egyszerű, moduláris, könnyen hozzáférhető React alapú felhasználói felület építéséhez kapcsoló komponens könyvtár, olyanokhoz hasonlóan, mint a Material-UI, Bootstrap. A Chakra UI egy egyedi, előre megírt React komponenseket tartalmazó könyvtár. Könnyen integrálható, reszponzív építőelemeinek segítségével felgyorsíthatjuk az alkalmazásunk felületének tervezését, javítva a fejlesztési élményt és modern, hozzáférhető felhasználói felületet fejleszthetünk. Természetesen nem feltétlen kell megelégednünk az előre konfigurált elemek kinézetével, saját meglátásunk szerint is testreszabhatjuk őket. A komponens gyűjteménye tág igényeket is kielégítenek, accordion-októl kezdve, alert-eken át, az egyszerűbb gombok, konténer elemekig. A Chakra UI-t a fentebb említett npm segítségével könnyen behúzhatjuk az alkalmazásunkba, mint függőség.

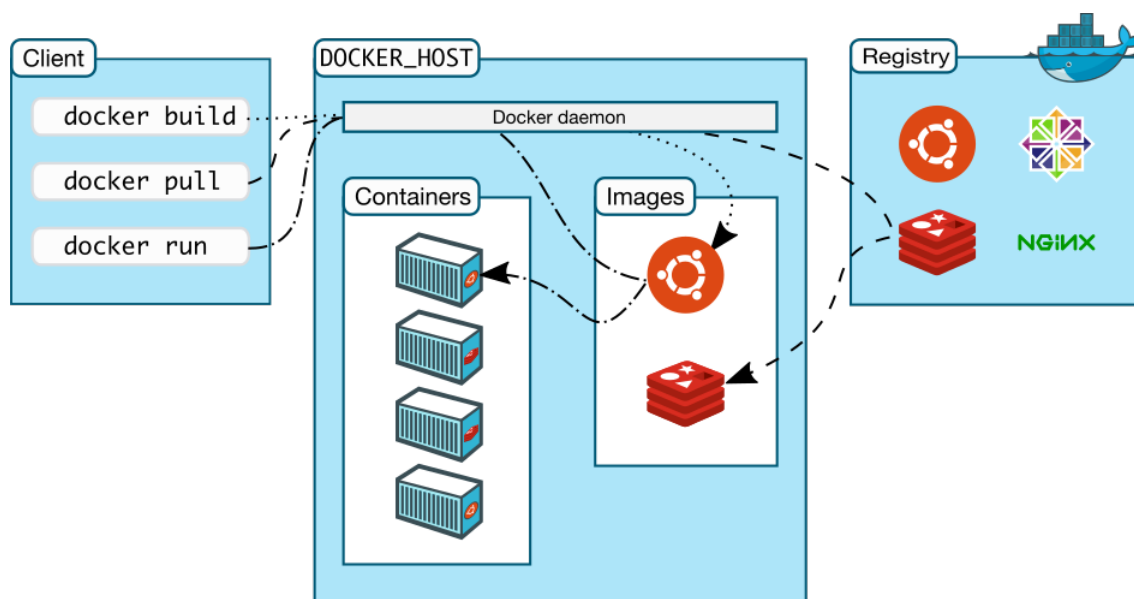
2.16. FullCalendar

A FullCalendar [4] egy olyan JavaScript könyvtár, melyet integrálhatunk React, Vue, Angular vagy TypeScript webalkalmazásunk fejlesztése során. Importálása után felvehetjük a csomag által specifikált formátumú eseményeinket egy tömbben és könnyen megjeleníthetjük őket, majd különböző eseménykezelőket rendelhetünk hozzájuk, legyen az a kattintás, vagy mouse-over kezelése. Természetesen a naptár kinézetét különböző tag-ekkel vagy akár Bootstrap segítségével testre is szabhatjuk. A Chakra UI-hoz (és természetesen más csomagok) hasonlóan ezt is az alkalmazásunk `package.json` fájljában deklarálva függőségként használhatjuk.

2.17. Docker

A Docker egy nyílt forráskódú konténerizációs technológia, amellyel az alkalmazásainkat egy egységbe, úgynevezett konténerbe csomagolhatjuk és futtathatjuk egy lazán elszigetelt, hordozható környezetben. A Docker által nyújtott módszertan segítségével lecsökkenthetjük a kód írása és szállítása, tesztelése és az éles környezetbe kerülése közötti időt. Az izoláltság miatt akár több ilyen konténert is futtathatunk egy gazdagépen, a konténerek hordozhatósága pedig garantálja, hogy más gépeken futtatva is pontosan ugyanolyan környezet áll elő, mint amelyet előkonfiguráltunk. A virtuális gépekkel ellentétben egy Docker konténer erőforrásai közösek a gazdagéppel és csak azokat az eszközöket tartalmazza, melyeket specifikáltunk és feltétlenül szükségesek az adott alkalmazásunk futtatásához. Egy Docker konténer nem tartalmaz operációs rendszert, egész egyszerűen megosztják a mögöttes kernelt a többi konténerrel, ezáltal kisebb a méretük, mint egy virtuális gépnek, jelentősen növelve a hordozhatóságát egy konténernek környezetek között.

A konténer működését futtatás előtt egy *Dockerfile* írja le, melyben a konténert konfigurálhatjuk, például azt, hogy milyen szükséges eszközök legyenek telepítve vagy akár milyen portokat nyisson meg a gazdagép vagy esetleg más konténerek felé.



2.5. ábra. A Docker működése
[3]

A gazdagépen futó *docker daemon* parancsokat fogad egy API-n keresztül. A parancsokkal az alkalmazásunkból egy *image*-t készíthetünk, melyet futtatva előáll egy alkalmazásunkat és a hozzá szükséges eszközöket futtató konténer. További parancsokkal befolyásolhatjuk már futó konténerek viselkedését is, összeköthetjük őket, különböző műveleteket végezhetünk rajtuk.

2.18. Git

A Git [1] napjaink legelterjedtebb elosztott modellű verziókezelő rendszere, mely képes a fejlesztés alatt álló dokumentumok, tervek, forráskódok és egyéb olyan adatok verzióinak kezelésére, amelyeken több ember dolgozik egyidejűleg. Egy nagyobb projekt esetében több fejlesztő dolgozik szimultán, lehet valaki éppen egy hibát javít, más pedig egy új funkciót implementál. A Git, mint verziókezelő rendszer minden egyes változtatást nyomon követ, ami a kijelölt fájlokon történik. A lokális változtatásokat közzétehetjük a többi csapattag számára a projekt számára kijelölt tárolóba (repository). A git segítségével "ágakat" (branch) hozhatunk létre, amely több munkafolyamatot függetlenít egymástól, melyeket később egybevonhatunk (merge), így ellenőrizve, hogy a párhuzamosan végzett fejlesztések véglegesített (commit) kompatibilisek egymással. Az önállóan dolgozók is profitálhatnak a verzikövetésből, hiszen egy hibát okozó rész lefejlesztése után visszaállíthatjuk egy működőképes állapotába alkalmazásunkat. Ezeken felül megadja a lehetőséget, hogy ne csak a saját számítógépünkön tároljuk a kódunkat, biztonságosabbá téve a fejlesztési folyamatot.

2.19. GitHub Actions

A GitHub Actions [5] egy olyan eszköz, amely lehetővé teszi hogy egy folyamatos integrációs és folyamatos szállítási (CI/CD, continous integration, continous development) környezetet alakítsunk ki. Segítségével a fejlesztők a kód változtatásokat naponta vagy akár óránként többször integrálják egy közös felületen, egyeztetik, ellenőrzik és tesztelik. Gyors és hatékony folyamat alakul ki, hamar felismerhetőek az új fejlesztésből adódó problémák, lecsökkentve a hibás kód javítására szánt időt. A központi elmélete, hogy gyors visszajelzést kapjunk az újonnan implementált fejlesztésünkről, és a rendszer kiadhatóságáról. A Jenkinssel, egy másik népszerű CI/CD eszközzel ellentétben a Github Actions esetén nem pipelineokat, hanem munkafolyamatokat, másnéven workflow-kat, deklarálhatunk `.yaml` fájlokban.

Egy workflow egy vagy több kisebb feladatot foglal magába. Egy munkafolyamat feladatokból (job) áll, a feladathoz tartozó lépések (step) egy futtatókörnyezetben (runner) hajtódnak végre. A konfigurációt leíró `.yaml` fájl része a verziókezelő tárolónknak a `.github/workflows` könyvtárban, és meghatározhatjuk, hogy milyen repository-val kapcsolatos művelet esetén milyen művelet (fordítás, tesztelés, telepítés) hajdódjon végre. Ezen felül akár manuálisan, vagy időszakosan is futtathatjuk ezeket a munkafolyamatokat.

2.20. SonarCloud

A SonarCloud [14] egy több mint 25 különböző nyelvet támogató, felhőalapú statikus analízis eszköz és kódminőség platform, amely folyamatosan biztosítja kódunk

karbantarthatóságát, megbízhatóságát és biztonságát. Beköthetjük a saját publikus GitHub tárolónkba, így képes ellenőrizni kódolási konvenciókat, duplikált kódot, teszt fedettséget, kód komplexitást, potenciális hibákat és sebezhetőségeket. A tárolónként végzett vizsgálatok eredményeit a SonarCloud felhőben tárolja, így nyomon követhetőek későbbi javítások, változtatások.

2.21. IntelliJ IDEA

Az IntelliJ IDEA [7] egy JetBrains által készített integrált fejlesztőkörnyezet. Az IDEA rengeteg saját felhasználói felülettel rendelkező beépített eszközt nyújt egy átlagos fejlesztő számára, amelyek nagyban megkönnyítik a mindennapi munkát. Egy fejlesztő által talán legtöbbet használtak és leghasznosabbak az alábbiak:

- Git verziókezelőt
- Különböző csomagkezelőket
- Kód kiegészítés
- Statikus analízis
- Refaktorálási lehetőségek
- Szintaxis kiemelés
- Adatbázis kapcsolódási lehetőség

Az alkalmazásom Java-ban írt Spring Boot alapú backend modulját is IntelliJ IDEA fejlesztői környezetben készítettem el.

2.22. WebStorm

A WebStorm [25] szintén egy JetBrains által készített integrált fejlesztőkörnyezet, amely nagyban hasonlít az IntelliJ IDEA-hoz, azzal a különbséggel, hogy JavaScript-hez és hozzá kapcsolódó más technológiák fejlesztéséhez használják. Felhasználói felülete és funkciói nagyban megegyeznek az összes többi JetBrains-es fejlesztőkörnyezethez.

A projektem JavaScript-es React-et használó frontend részét a WebStorm segítségével írtam.

3. fejezet

A program specifikációja

Mielőtt egyből a projekt architektúráis tervezésébe vagy fejlesztésébe fogtam volna, meghatároztam saját magam, és főleg a webalkalmazás számára különböző követelményeket és specifikációkat. Ebben a fejezetben ezeknek a funkcionális, nem funkcionális követelményeknek, illetve a feladatkiírásban is röviden leírt specifikációknak a körüljárására kerül sor.

3.1. Funkcionális követelmények

Az applikáció felhasználói felülete könnyen kezelhető, modern, újrahasználgató elemeket, felhasználók számára közérthető jelzéseket tartalmaz. Lehetőség van világos vagy sötét mód kiválasztására, amely változtatja az oldal színösszeállítását.

3.1.1. Naptár

Az applikáció főoldalán megtekinthető az eseményeket tartalmazó naptár. A naptár jelzi az esemény nevét, kezdési és végzési időpontját az eseménynek, továbbá az egérrel egy esemény felé érve elolvashatjuk az adott rendezvény rövid leírását is. Az eseményeket meg lehet napi, heti, vagy havi felosztásban is tekinteni. Kényelmi funkcióként rendelkezik egy gombbal, ami az aktuális jelen napra irányítja a felhasználót. Egy eseményre kattintva megtekinthetjük annak összes adatát. Egy napra kattintva az adott naphoz tartozó összes esemény összes paraméterével találkozhatunk.

3.1.2. Események megtekintése

Az esemény adatainak megtekintésére létezik egy oldal, ahol az alábbi információkat megtaláljuk egy rendezvényről:

- Név
- A rendező öntevékeny kör neve
- Kezdet és vég dátum

- Plakát
- Helyszín
- Facebook esemény linkje
- Rövid leírás
- Hosszabb leírás

Egy adott nap eseményeinek megtekintése esetén az események egy-egy kártyán helyezkednek el, azonos formátumban, mintha csak egy eseményre kattintottunk volna a naptárban.

3.1.3. Esemény létrehozása és módosítása

Egy öntevékeny körhöz eseményt csak az adott kör körvezetője vagy PR (Public-relations) felelőse vehet fel. Ezeket a felhasználóról szóló adatokat az AuthSCH-ből kapja az applikáció. A körök vezetői és PR felelősei persze törölhetik vagy akár módosíthatják is a már korábban publikált eseményeket, ha valamilyen változtatást szeretnének rajta utólag végrehajtani. Egy esemény összes paramétere változhat egy ilyen folyamat során. Fontos funkció, hogy a megadott adatok validálásra kerülnek. Kötelező adatnak számítanak:

- Név
- A rendező öntevékeny kör reszortja, amely később nem kerül megjelenítésre, csupán a kör kiválasztásának megkönnyítésére hivatott mező
- A rendező öntevékeny kör neve
- Kezdet és vég dátum
- Helyszín

Az esemény kezdete és vége nem lehet az adott létrehozási pillanatnál korábbi időpont, illetve az esemény vége nem lehet korábban, mint annak kezdete.

3.1.4. Felhasználók kezelése

Az AuthSCH fiókkal rendelkező felhasználók képesek belépni az alkalmazásba. A belépett felhasználók rendelkeznek egy profil menüponttal, ahol megtekinhetik az alábbi AuthSCH-ből érkező adatokat:

- Nevüket
- Email címüket
- Az öntevékeny körök listáját, melyeknek tagja belépett felhasználó

- Eseményszűrés kapcsoló állapotát

A belépett felhasználók a profiljukon megtekinthetik, hogy be van-e kapcsolva nekik az események szűrése, továbbá elérik a szűrők módosítása menüpontot, melynek működését a következő fejezetben mutatom be.

3.1.5. Események szűrése

Az AuthSCH fiókkal rendelkező belépett felhasználóknak meg van a lehetősége, hogy szűrjék az eseményeket. Az események szűrése menüponton beállíthatják, mely öntevékeny kör eseményeit nem szeretnék látni az alkalmazás főoldalán található naptárban, illetve ne kapjanak email-es értesítéseket a létrehozásukkor.

3.2. Use case-k az alkalmazásban



3.1. ábra. Az alkalmazás tervezett use-case diagramja

A fentebb található use-case diagramon vizuálisan is reprezentáltam a "Funkcionális követelmények" szekcióban leírtakat. Látható, hogy a nem bejelentkezett, ismeretlen felhasználók számára nem sok funkció elérhető. Egy bejelentkezett felhasználó hasonló jogokkal rendelkezik, annyi kiegészítéssel, hogy a saját profilját is meg tudja tekinteni, illetve befolyásolhatja, hogy milyen eseményeket jelenítsen meg a naptárban. Egy bejelentkezett felhasználó, aki körvezető vagy PR joggal rendelkezik egy

körhöz, létre tud hozni az öntevékeny köréhez tartozó eseményeket, illetve a körhöz tartozó már meglévő rendezvényeire módosítani, törölni tudja.

3.3. Nem funkcionális követelmények

A program több olyan technológiai eszközt is implementáljon, amellyel a felhasználó a mindennapi használatkor nem szembesül, azonban megkönnyítik az alkalmazást fejlesztő programozó életét. Az alkalmazás kompatibilis és futtatható a modern böngészők többségében.

3.3.1. Docker konténerizáció

Az alkalmazás backend-je és frontend-je egymástól függetlenül teljes mértékben konténerizálható. A frontend és backend konténere is képes lefogatni és automatikusan futtatni a két külön alkalmazást. A két konténer összeköttetésben van egymással, így a frontend felől érkező hívások eléri a backendet. A futtatott frontend konténer felülete elérhető a külvilág számára, a konténert tartalmazó gazdagépen elérhető a felhasználói felület. A fejlesztő képes a két konténert létrehozni és elindítani egy egyszerű `.cmd` fájl futtatásával.

3.3.2. A kód kezelése

A kód a Robert C. Martin által megfogalmazott Clean Code elveket követi, SonarCloud Quality Gate-jén, különböző statikus analízis eszközök értékelésén megfelel. Nem tartalmaz kódduplikációt, sebezhetőséget, biztonsági hotspot-ot, code smell-t. A kódbázis verziókövetve van, minden funkcionalitás külön ágon. Egy funkció ága pull-request-el kerül a fő fejlesztési ága, mely során lefut a SonarCloud ellenőrzője, és a GitHub Action-s front- és backend-re kiterjedő munkafolyamatai is. A commit üzenetek imperatív módon íródnak.

3.3.3. Tesztelés

A program implementál DAO (Repository), üzleti logika (Service), REST API (Controller) rétegbeli teszteket. Az alkalmazás tartalmaz integrációs teszteket és olyan Controller rétegbeli teszteket, melyek különbözőképp konfigurálják a Spring Context-et, és használnak Mockito által ajánlott mock-olási technikákat.

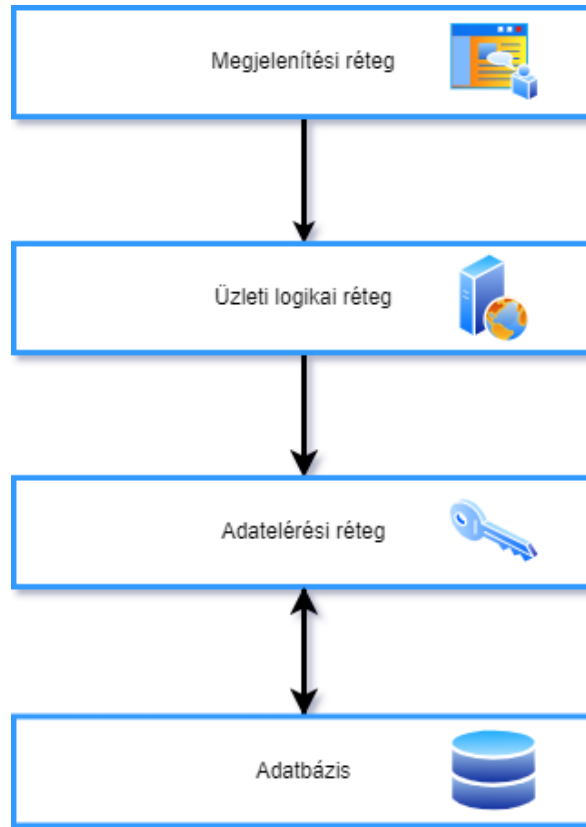
4. fejezet

Tervezés

4.1. Az alkalmazás felépítése

Ebben a fejezetben bemutatom az elkészült alkalmazás architektúráis felépítését, kitérek a tervezési döntések okaira. A webalkalmazás felépítésének megtervezésekor a háromrétegű architektúrát választottam. Ez a tervezési minta megkülönbözteti az alkalmazás három komponensét vagy rétegét:

- megjelenítési réteg:
 - Az alkalmazás JavaScriptet és React-et használó frontend modulja.
 - Az üzleti logikai réteg által szolgáltatott adatok felhasználóbarát megjelenítésére szolgál.
- üzleti logikai réteg:
 - Az alkalmazás Java-ban írt Spring-et használó backend modulja.
 - Olyan metódusokat tartalmaz, melyek helyesen implementálják az alkalmazással szemben elvárt működést és funkciókat.
- adatréteg:
 - A backend modul adatelérési, másnéven DAO vagy Repository rétege.
 - * Feladata az adatbázissal való kommunikáció.
 - A PostgreSQL-t használó adatbázis.
 - * Feladata az alkalmazás által használt adatok hosszan tartó, perzisztens tárolása.



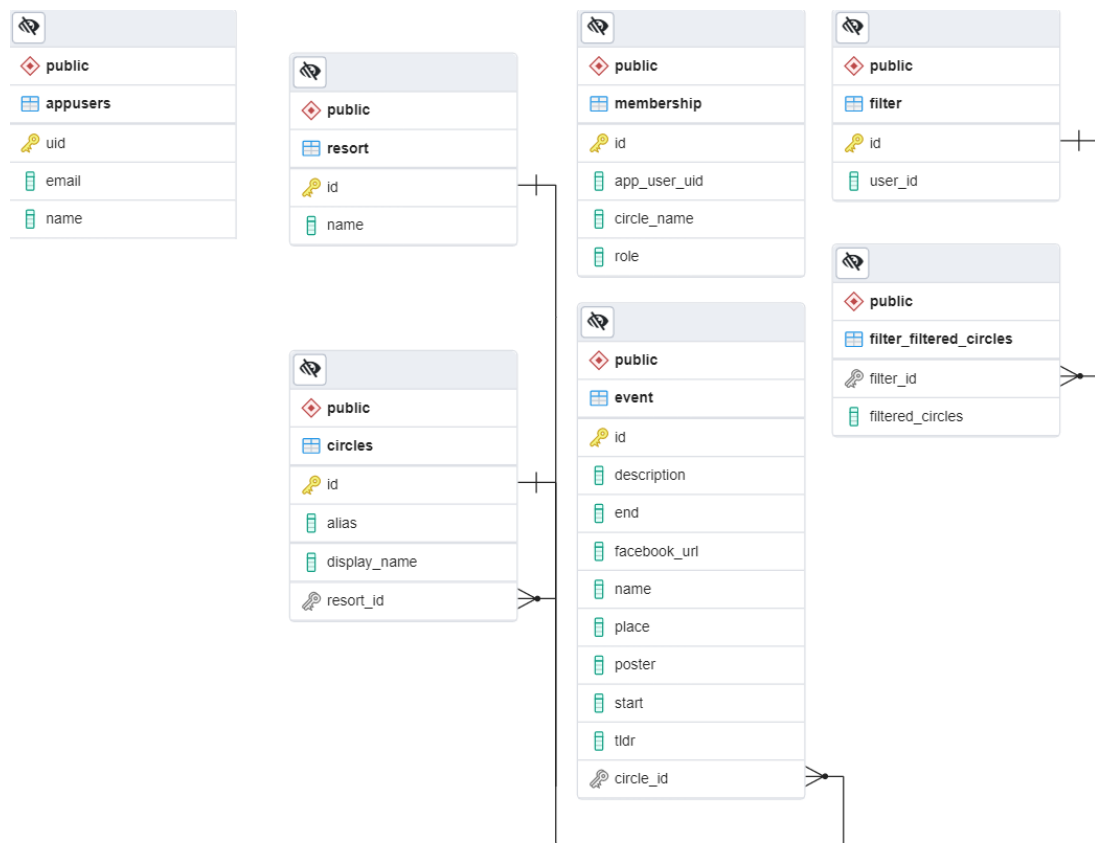
4.1. ábra. A háromrétegű architektúra felépítése

A rétegek csak az alattuk lévő réteggel kommunikálnak, azokkal is csak az API-jaikon keresztül. Az alsóbb rétegek nem küldenek maguktól üzenetet a felsőbb rétegek felé, kizárólag kérésre. Az egyes rétegek könnyen cserélhetőek, ha az API-juk logikai implementációja változatlan marad. A jelen dolgozat esetében a megjelenítési réteg HTTP-n (Hypertext Transfer Protocol) keresztül kommunikál az üzleti logikai réteggel, annak REST API-ján keresztül. A szállított erőforrás dokumentum JSON formátumban utazik.

Az architektúra előnye, hogy rétegek nyilvánvalóan más-más funkcionalitásért felelnek, így felelősségi köröket jelölhetünk ki az alkalmazásunk fejlesztése során, ezáltal külön-külön könnyen bővíthetőek és karbantarthatóak.

4.2. Adatbázis

Az adattárolás módjának a PostgreSQL relációs adatbázist választottam. Az adatbázis logikai felépítését az entitások és a közöttük lévő kapcsolatok felépítésével kezdtem. Ezeket a felső szinten átgondolt objektumokat feleltettem meg az adatbázisban található tábláknak. Az adatbázis sémáját az alábbi ER (Entity Relationship) diagram mutatja be, amely tartalmazza egyes táblák neveit, több mezőjüket és típusaikat.



4.2. ábra. Az alkalmazás adatbázisának felépítése

Természetesen minden tábla rendelkezik elsődleges kulccsal, azonban az összes tábla külön bemutatást érdemel.

Felhasználók A felhasználók adatai az appusers táblában kerülnek tárolásra. Itt megtalálható az egyedi UID-jük, email címük és nevük is. Fontos, hogy a jelszavak nem kerülnek tárolásra, hiszen a már korábban említett AuthSCH-t használják a bejelentkezésre.

Tagságok A membership tábla tárolja le egy felhasználó körtagságait, természetesen egy kör nevével, a felhasználó azonosítójával, és a felhasználó a körben betöltött szerepével.

Szűrők Egy felhasználóhoz tartozó szűrőt a filter táblában találjuk. Egy filter tartalmazza a saját azonosítóját és a felhasználóét.

Reszortok A rezortok a resort táblában találhatóak. Rendelkezenk egy elsődleges kulccsal és névvel.

Körök A körök az összes paraméterükkel a circles táblában találhatóak. Külső külsként hivatkozik a kört magába foglaló rezort azonosítójára.

Szűrt körök Ha létezik egy felhasználó szűrője, akkor a szűrt köröket a `filter_filtered_circles` táblában találjuk. Ebben a táblában egy rekord hivatkozik a már korábban említett `filter` tábla egy rekordjának elsődleges kulcsára külső kulcsként, továbbá tartalmazza az adott szűrő által meghatározott köröket.

Események Az összes esemény az `event` táblában található, minden paraméterével együtt. Egy esemény külső kulcsként hivatkozza be annak az öntevékeny körnek az egyedi azonosítóját, melyhez az esemény tartozik.

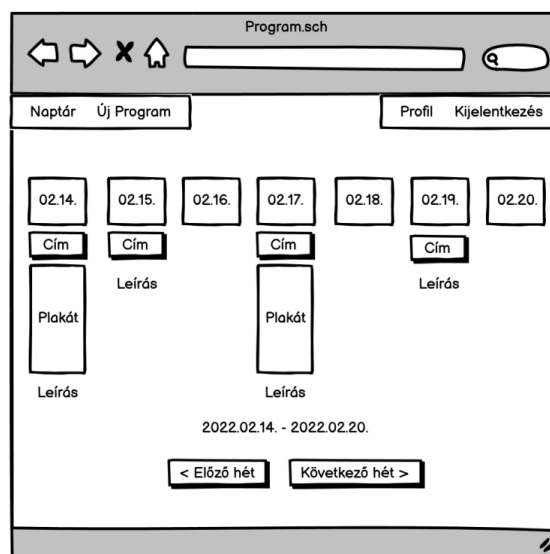
4.3. A backend tervezése

Az alkalmazás backend modulja a Java Spring alkalmazás, amely különböző tranzakciókat és lekérdezéseket futtat az adatbázisban, implementálja az alkalmazás üzleti logikáját, és egy hívható REST API-t nyújt a frontend felé. A modult logikailag az alábbi elkülönülő részekre osztottam szét:

- **Repository:** Itt találhatóak azok a DAO osztályok, melyek megvalósítják a korábban is tárgyalt ORM technikát. Minden modellbeli entitáshoz tartozik egy Repository osztály, mely implementálja a JPA által nyújtott interfészt. Segítségükkel mindenféle adatbázisműveleteket hajthatunk létre az entitások sémája alapján.
- **Domain:** Ez alkotja az alkalmazás modelljét, benne találhatóak az entitások. Minden entitás egy adatbázisbeli táblának feleltethető meg.
- **Mapper:** Itt találhatóak azok az osztályok, melyek a Mapstruct segítségével elvégzik a konvertálást az entitások és DTO-k között.
- **DTO:** Itt találhatóak azok az osztályok, amelyeket a REST API szolgáltat a frontend számára. Minden entitáshoz tartozik általában egy vagy több darab.
- **Szolgáltatás:** Ez a réteg valósítja meg a tényleges üzleti logikát. A kontrollerek minden művelet esetén meghívják a hozzá tartozó Service osztály egy metódusát, mely egy DTO-val tér vissza számára.
- **Kontrollerek:** A REST API implementációjáért felelős. Minden kontroller osztály rendelkezik egy vagy több végpont metódussal, melyek különböző HTTP kérés esetén kerülnek meghívásra. A kérés az esetek túlnyomó többségében egy `ResponseEntity`-be ágyazott DTO-val tér vissza, így a DTO-n kívül extra információként visszaküldünk egy válasz HTTP kódot is, hogy extra betekintést nyújtsunk az eredmény végkimenetelére.

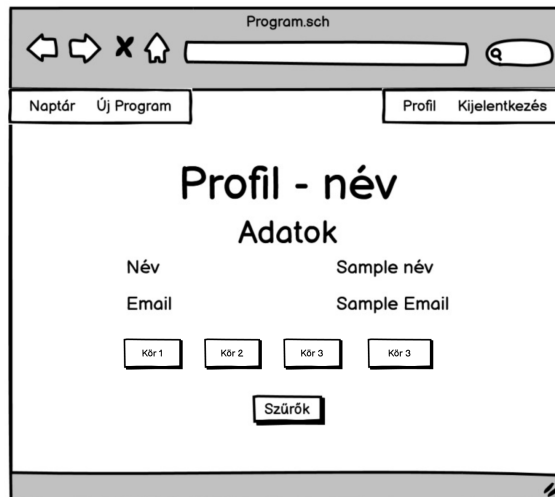
4.4. A frontend tervezése

Technológiaként a React.JS-t választottam egyszerűsége, nem túl meredek tanulási görbéje miatt, továbbá azért, mert a másik opció Angular lett volna, de valami teljesen újba akartam fogni. A React esetében nem igazán lehet beszélni semmilyen érdekes architektúráról, legyen az MVC (Model-View-Controller), vagy MVVM (Model-View-ViewModel), a jelen dolgozat esetében is csak a megjelenítésért felelős. Komponensalapúsága miatt nagyon egyszerű volt a kisebb építőeszközökből bottom-to-top módszerrel felépíteni a teljes felületet. A felület átgondolása során több mock-olt tervezetet, wireframe-t is elkészítettem. A tervezetek az asztali számítógépek által használt böngészők kinézetének felel meg.



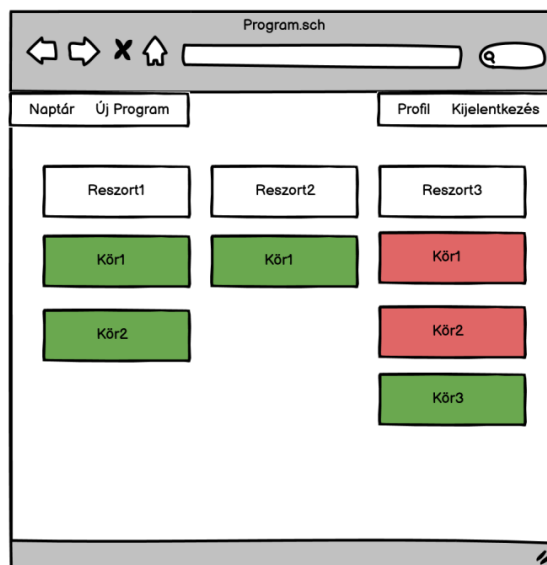
4.3. ábra. Az alkalmazás főoldala

A fent található ábra bemutatja a naptár alapú elrendezést a főoldalon. A naptár kinézetéért és megjelenítéséért a már korábban is tárgyalt FullCalendar npm-es csomag a felelős. Itt találhatóak a nyilvántartott események, melyeket kedvére böngészhet akármelyik felhasználó.



4.4. ábra. A bejelentkezett felhasználó profilja

Értelem szerűen ezen a képernyőn szembesülhet a felhasználó a saját adataival és itt tudja elérni a szűrők menüpontját is.



4.5. ábra. A szűrő módosítását kezelő képernyő tervezete

A 4.6-os ábrán láthatóak az öntevékeny körök rezsortok szerinti bontásban, melyekre szűrőket állíthatunk be. A zöld jelzés azt jelenti, hogy a felhasználó szeretné látni a naptárban az adott körhöz tartozó eseményeket, a piros jelzés értelem szerűen pedig azt, hogy nem.

4.6. ábra. Egy esemény részletes lebontása

A fentebb található ábrán látható egy esemény részletes adatainak tervezete. A rendezvény egy újrahasználatos kártya formájában jelenik meg, emiatt a komponens segítségével napi lebontás is nagyon, azzal a különbséggel, hogy több esemény paraméterei is láthatóak az oldalon, és egy fejléc megmutatja nekünk, hogy az adott napon hány bejegyzett esemény került kiírásra. A különböző jogkörök megkülönböztetése esetében csupán annyi a változtatás, hogy egy esemény öntevékeny körének PR felelőse és körvezetője látja az adott eseményhez tartozó a módosítás és törlés gombot, más felhasználók pedig nem.

Program.sch

Naptár Új program Profil Kijelentkezés

Eseménykezelő

Esemény neve

Reszort

Kör

Esemény kezdete

Esemény vége

Helyszín

Facebook esemény link

Plakát link

Program rövid leírás / TLDR

Az esemény leírása

Létrehozás / Módosítás

4.7. ábra. Az esemény létrehozásának és módosításának egyforma oldala

Egy új esemény létrehozásának és egy már meglévőnek a módosításának képernyője megegyezik, azzal a különbséggel, hogy a módosításnál az adatbekérő űrlap beviteli mezői előre kitöltöttek az adott esemény paramétereivel.

5. fejezet

Megvalósítás

A tervezési fázis után konzulensem tanácsára nem rétegeként, hanem a funkciók mentén hajtottam végre az alkalmazás implementációját. A backend és frontend fejlesztése általában párhuzamosan haladt egymással. Az adatbázis és entitások megtervezésére szánt idő és energiabefektetés megtérült, ugyanis később csak nagyon minimális változtatásokat kellett elvégezni rajtuk.

Az implementációhoz hasonlóan az elkészült alkalmazást is bizonyos funkciók mentén, alulról felfelé (backendtől frontendig) szeretném bemutatni, természetesen a legérdekesebb részeket kiemelve.

5.1. Konfiguráció

Mindenek előtt a Spring Initializr segítségével elkészítettem a backend projektjét és neki is láttam a különböző szolgáltatások konfigurálásának.

Mint ahogy az Spring Boot alapú projektek esetében is szokott lenni, az alkalmazásom YAML alapú konfigurációinak egy része egy `application.yml` fájlban található, amely az alábbi beállításokat tartalmazza:

- az adatbázishoz való csatlakozási és autentikációs információk, JPA beállítások
- OAuth2-vel és AuthSCH-val kapcsolatos konfigurációk
- elektronikus levélküldés
- logolás és API dokumentációgenerálás

5.1.1. Adatbázis

Az adatbázisnak kezdetben nem kell rengeteg adatot tárolnia, azonban köröknek és reszortoknak már létezniük kell az első indítás után is. Inicializálás utáni adatbeillesztésre általában két megszokott mód létezik:

- SQL script alapú

- Java kód alapú

A jelen projekt esetében az utóbbit választottam, ami azt jelenti, hogy ha a rendszernek egy API híváson keresztül szüksége van a körök listájára és az adatbázis még nem tartalmazza őket, akkor az összes öntevékeny kör és reszort injektálásra kerül az adatbázisba. Ezzel a módszerrel kiszűrhetőek különböző hordozhatósági problémák, az alkalmazás még egy Docker konténerben futtatva az első hívás esetén is helyesen funkcionál.

5.1.2. Security

Az alkalmazásom biztonsági konfigurációjáért a Spring Security a felelős. A Java kód alapú annotációkkal való beállítást preferáltam a projekt esetében. A konfigurációs osztály a `WebSecurityConfigurer` osztály megvalósító `WebSecurityConfigurerAdapter` osztályból származik. Itt egy egyszerű `@Override` annotációval ellátott `configure` metódusban beállíthatjuk a Spring Security viselkedését.

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    http
        .csrf().disable()
        .authorizeRequests()
        .antMatchers("/login", "/loggedin", "/logout").permitAll()
        .antMatchers("/api/**").permitAll()
        .and()
        .formLogin()
        .loginPage("/login");
}
```

Jelen projekt esetében megelégedtem a Cross-origin resource sharing kikapcsolásával, mert szerettem volna, hogy extra konfiguráció nélkül meghívható legyen az alkalmazás REST API-ja bármilyen ismeretlen forrásból. Egy éles környezetben ezt természetesen nem érdemes bekapcsolva felejtetni. A függvényben beállítható a bejelentkezési URL `antMatcher`-ek segítségével testreszabhatjuk a végpontokhoz szükséges jogosultságokat is.

5.1.3. Thymeleaf

A Thymeleaf egy Java HTML template engine, mely képes szerver oldalon renderelni számunkra előre megírt dokumentumokat. Spring keretrendszert használó alkalmazásunkba teljesen integrálható. A jelen projekt esetében egyedül az email-ek küldése során került felhasználásra. Az alkalmazás Thymeleaf által generált HTML üzeneteket küld a felhasználóknak, nem sima egyszerű szöveges üzenetet, ezért későbbi

testreszabhatóságnak is teret ad. A beállítása a következőképpen alakult egy konfigurációs osztályban:

```
@Bean(name = "templateEngine")
public ISpringTemplateEngine templateEngine() {
    SpringTemplateEngine engine = new SpringTemplateEngine();
    engine.setTemplateResolver(templateResolver());
    Set<IDialect> dialects = new HashSet<>();
    dialects.add(new Java8TimeDialect());
    engine.setAdditionalDialects(dialects);
    return engine;
}

private ITemplateResolver templateResolver() {
    SpringResourceTemplateResolver resolver = new
    ↪ SpringResourceTemplateResolver();
    resolver.setApplicationContext(applicationContext);
    resolver.setPrefix("classpath:/templates/");
    resolver.setSuffix(".html");
    resolver.setTemplateMode(TemplateMode.HTML);
    return resolver;
}
```

A `templateEngine` metódusban integráljuk a Thymeleaf-et a Spring-es ökoszisztémába, a `templateResolver` függvényben pedig meghatározzuk a sablonok helyeit, formátumait, továbbá egyéb paramétereiket.

5.1.4. GitHub Actions

A projekt két modulja egyetlen egy GitHub repository-ban található, a monorepo elvet követi. Mind a backend, mind a frontend esetében szerettem volna megbizonyosodni, hogy minden egyes változtatásom esetében helyes marad az alkalmazás működése, ezért használtam a két modul esetében a GitHub Actions-t. A teljes projekt gyökérkönyvtárában található `.github/workflows` mappában található két munkafolyamatot leíró YAML fájl, melyek a projekt két moduljára vonatkoznak. A backend és frontend esetében is minden pull-request alkalmával lefut ez a két munkafolyamat.

A backend Maven, a frontend Node.js és npm segítségével kerül buildelésre. A frontend ki is települ a saját GitHub Pages oldalra, tehát ha rendelkeznék egy saját (mondjuk Amazon Web Service) webszerverrel, amin futna a backend, és az API hívásokat is átirányítanánk oda, akkor teljesen remote is üzemelhetne az alkalmazás, így azonban a deploy művelet csak kíváncsiság gyanánt lett implementálva.

5.2. Naplózás

Mindenképp szerettem volna értelmezhető logolást implementálni az alkalmazás backend-jébe, hogy valós időben lehessen követni a beérkező kéréseket és emberi szem számára is könnyen értelmezhető legyen. Ezt a fajta naplózást a már korábban tárgyalt AOP megközelítéssel végeztem el. Két lényeges fogalmat érdemes ismerni a megoldás bemutatása előtt:

- join point: a program végrehajtásának egy lépése, például egy metódus végrehajtása, vagy kivételkezelés
- pointcut: egy predikátum, amely illeszkedik a join point-okra, a pointcut expression nyelvvel lehet leírni

A jelen projekt esetében az applikációban található összes metódushívására és kivételkezelésére határoztam meg a pointcut expression-t:

```
private static final String API_POINT_CUT = "execution(*  
↳ hu.bme.aut.programsch.*.*(..))";  
private static final String EXCEPTION_POINTCUT = "execution(*  
↳ hu.bme.aut.programsch.*.*(..))";
```

Ezek után ezt a két pointcut-ot megadtam két hozzájuk tartozó metódusnak, melyeket a későbbi naplózó metódusok fognak használni:

```
@Pointcut(API_POINT_CUT)  
public void logController() {  
    // empty  
}  
  
@AfterThrowing(value = EXCEPTION_POINTCUT, throwing = "exception")  
public void logsErrors(JoinPoint joinPoint, Throwable exception) {  
    logDetails(joinPoint);  
}
```

Kettő fontos logikai egységre bonthatjuk a kliens kommunikációját a backend-del, ezek a kérés és válasz. Mindkettőnek minden egyes lépésébe szeretnénk betekintést nyerni. Korábban meghatároztuk, hogy mit jelent egy join point, azonban az alkalmazásnak fogalma sincs, hogy mikor csináljon mit. Itt nyer fontos szerepet ismét két fontos fogalom:

- aspect: egy cross-cutting concern, kereszthivatkozás modularizációja, korábban említésre került
- advice: egy művelet egy aspect által egy bizonyos join point-nál

Ezeknek az ismeretében létrehoztam a join point-ot a createJoinPointForLogs metódusban, majd meghatároztam a logDetails függvényben, hogy egy kérés vagy válasz esetén feltétlenül szeretnék információt kapni az alábbiakról:

- a join point
- a hívott osztály
- annak metódusa
- a metódus argumentumai

Nem maradt más, mint meghatározni a kéréseket és válaszokat naplózó függvényeket:

```
@Before("logController()")
public void logRequest(JoinPoint joinPoint) {
    logDetails(joinPoint);
    log.info(createJoinPointForLogs(joinPoint, RestApiMessageType.REQUEST));
}

@AfterReturning("logController()")
public void logsResponse(JoinPoint joinPoint) {
    logDetails(joinPoint);
    log.info(createJoinPointForLogs(joinPoint, RestApiMessageType.RESPONSE));
}
```

A sikeres fejlesztés után az alábbi példához hasonló, kulturált üzeneteket kapunk egy kérés esetén:

```
h.b.a.p.l.controller.EndpointLogger : Class: hu.bme.aut.programsch.web.ResortController
h.b.a.p.l.controller.EndpointLogger : Method: getResorts
h.b.a.p.l.controller.EndpointLogger : Arguments: []
h.b.a.p.l.controller.EndpointLogger : Joinpoint: execution(ResponseEntity hu.bme.aut.programsch.web.ResortController.getResorts())
h.b.a.p.l.controller.EndpointLogger : getResorts method called with no arguments
h.b.a.p.l.controller.EndpointLogger : Class: hu.bme.aut.programsch.service.ResortService
h.b.a.p.l.controller.EndpointLogger : Method: findAll
h.b.a.p.l.controller.EndpointLogger : Arguments: []
h.b.a.p.l.controller.EndpointLogger : Joinpoint: execution(List hu.bme.aut.programsch.service.ResortService.findAll())
h.b.a.p.l.controller.EndpointLogger : findAll method called with no arguments
```

5.1. ábra. Reszortokat lekérdező GET HTTP kérés naplózása

A kéréseket és válaszokat naplózó implementáció mellett az alkalmazásom az összes végpont végrehajtási idejét is logolja.

5.3. Docker konténerizáció

Az egyetemi tanulmányaim során már korábban találkoztam a Docker-rel, azonban a projekt tervezése során felmerült bennem az a gondolat, hogy mindenképpen szeretnék vele jobban megismerkedni, mert hasznos lehet a későbbi szoftverfejlesztői

pályafutásom során. A funkció megvalósítása a projekt életciklusa vége felé készült csak el, amikor már lényegi működést, tesztelhetőséget lehetett kikényszeríteni a projekt két moduljából.

5.3.1. Backend

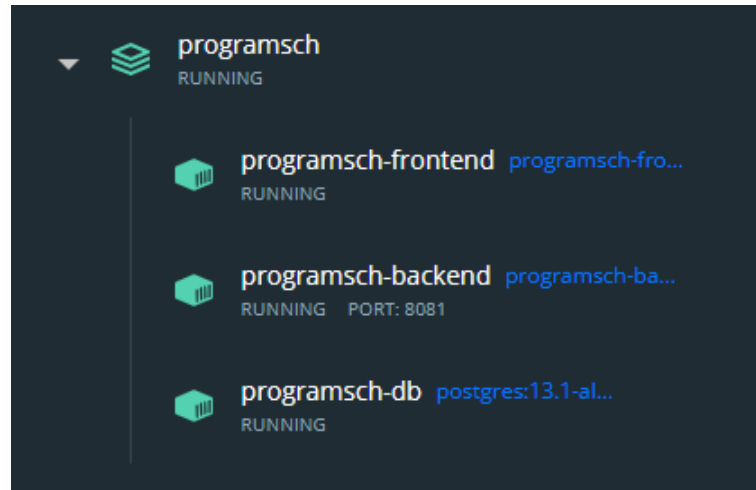
A backend konténeréhez szükséges image a Docker által hostolt image repository-n, a DockerHub-on található adoptopenjdk/openjdk11 image-re épül rá. Ez az image már rendelkezik több olyan eszközzel, többek között Java 11-el, amelyek szükségesek az alkalmazás futtatásához. A backend saját image-ének megépítéséhez készítettem egy Dockerfile-t, melyben megadtam, hogy a Maven 3.8.6-os verzióját szeretném majd feltelepíteni a konténerbe buildelés után. A konténerben így már minden adott, hogy indítás után egy `mvn spring-boot:run` paranccsal futtassa is a projekt backend-jét.

5.3.2. Frontend

A frontend esetében is nagyon hasonló megoldással közelítettem meg a feladatot, azonban az ehhez a modulhoz tartozó image a `node:13.12.0-alpine` image-re épül, amely tartalmazza a Node.js-t. Ezek után csupán a frontend alkalmazás függőségeit kell inicializálnia egy `npm install` paranccsal, majd egyéb beállítások után egy `npm start` kiadása után futtatja is az alkalmazás frontend-jét.

5.3.3. Compose

Egy Docker compose fájlban YAML formátumban specifikálhatunk például szolgáltatásokat, hálózati beállításokat Docker-t használó applikációk esetében. Ebben a fájlban specifikáltam is például az adatbázis konténerének az adatait is, hiszen nem volt szükséges egy egyedi image buildelése, elég volt csupán a PostgreSQL által elkészített `postgres:13.1-alpine` image is. A compose fájlban konfiguráltam a backend és frontend modulokat is, azoknak port beállításait, illetve azt is, hogy milyen image-eket vegyenek alapul a konténerek.



5.2. ábra. Előálltak az alkalmazáshoz tartozó konténerek

A compose fájlt használva, futtatás után a konténerek a Docker Desktop Application-ben csoportosítva jelennek meg és látszódik, hogy a munkánk sikerrel járt, futnak a konténerek. A frontend elérhetőségét a böngészőben, a backend-ét pedig Postman-ben teszteltem, mindkettő helyesen működött az első hívásra.

5.4. Bejelentkezés

A bejelentkezés vagy kijelentkezés lehetősége a navigációs sávon egy gomb formájában érthető el. A gombbal történő felhasználói művelet a frontend-től érkezik a backend-en található LoginController osztály felé, amely kezeli a be- és kilépéseket.

Belépés esetén a következő a folyamat:

1. Az osztály létrehoz egy felhasználóra egyedi állapotot leíró String-et SHA256 hasheléssel.
2. A felhasznált technológiák AuthSCH-val kapcsolatos rész szerint létrehoz egy egyedi belépési URL-t, melyet visszaad a frontend-nek.
3. A frontend átirányítja a felhasználót erre az egyedi címre, aki ott be tud jelentkezni.
4. Az AuthSCH admin konzolján az alkalmazásom esetében megadtam, hogy a /loggedin végpontra irányítsa át a felhasználókat bejelentkezés után, így ez a végpont meghívódik a backend-en.
5. A választ a backend értelmezi, validálja, kinyeri belőle a felhasználóra releváns információkat, majd létrehozza a belépő emberhez tartozó entitást is.
6. A kérés feldolgozása után visszairányításra kerülünk a főoldalra, immár bejelentkezett felhasználóként.

Kilépés esetén közel sem ilyen bonyolult a folyamat, egyszerűen a /logout címre beérkezett kérés után invalidáljuk az aktuális session-t, a SecurityContextHolder-ben tisztára töröljük a kontextust, illetve elveszük a felhasználó jogait.

5.5. Szűrők

A bejelentkezett felhasználóknak megnyílik a lehetőség, hogy módosítsák a szűrőiket a frontenden /filters oldalon, amiért a EditFilters.js komponens a felelős. A komponens lekéri a backendtől a reszortokat és a hozzájuk tartozó köröket, továbbá a jelenlegi szűrők állapotát. A frontend ezeknek az adatoknak a tudatában már képes megjeleníteni a felhasználó szűrőit.

```
const resortList = resorts.map(resort => {
  return <div className="col-md-4" id="resortCard" key={resort.name}>
    <div className="list-group">
      <li className="list-group-item">
        <h4 className="list-group-item-heading">
          <b>{resort.name}</b>
        </h4>
      </li>
      {circleList.map(circle => {
        if (circle.resort.name === resort.name) {
          return <a onClick={() => handleClickOnCircle(circle)}
            ↪ key={circle.displayName} href="#"
              className={handleCircleState(circle.displayName)}>
                {circle.displayName}
              </a>
        }
        return <></>
      })}
    </div>
  </div>
});
```

A reszortok és a hozzá tartozó köreik dinamikusan kerülnek renderelésre. Egy kör neve zölddel jelenik meg, ha a felhasználó nem szeretné szűrni a kör eseményeit, pirossal, ha igen. Értelemszerűen kattintásra az ellenkező színre vált a kör mezője.

Kollégiumi Felvételi és Érdekvédelmi Reszort	Kollégiumi Számítástechnikai Kör	Kultúr Reszort
Szintképviselek Tanácsa	Hallgatói Tudásbázis	La'Place Café
Kollégiumi Felvételi Bizottság	SecuriTeam	Játszóház
	Sysadmin	Impulzus
	NETeam	Bor Baráti Kör
	DevTeam	Bűvész Kör
		Local Heroes Szerepjátzó Kör

5.3. ábra. A filterek beállítását tartalmazó oldal részlete

A felhasználó által végrehajtott változtatások regisztrálva vannak egy gomb eseménykezelőjében, ezáltal a backend-en egyből lefut a frontend-en végzett változtatás.

```
@Transactional
public FilterDto changeUserFilters(FilterDto filterDto) {
    Filter filter = filterRepository.findById(filterDto.getId())
        .orElseThrow(() -> new IllegalArgumentException("Filter not found"));
    filter.setFilteredCircles(filterDto.getFilteredCircles());
    return filterMapper.filterToDto(filterRepository.save(filter));
}
```

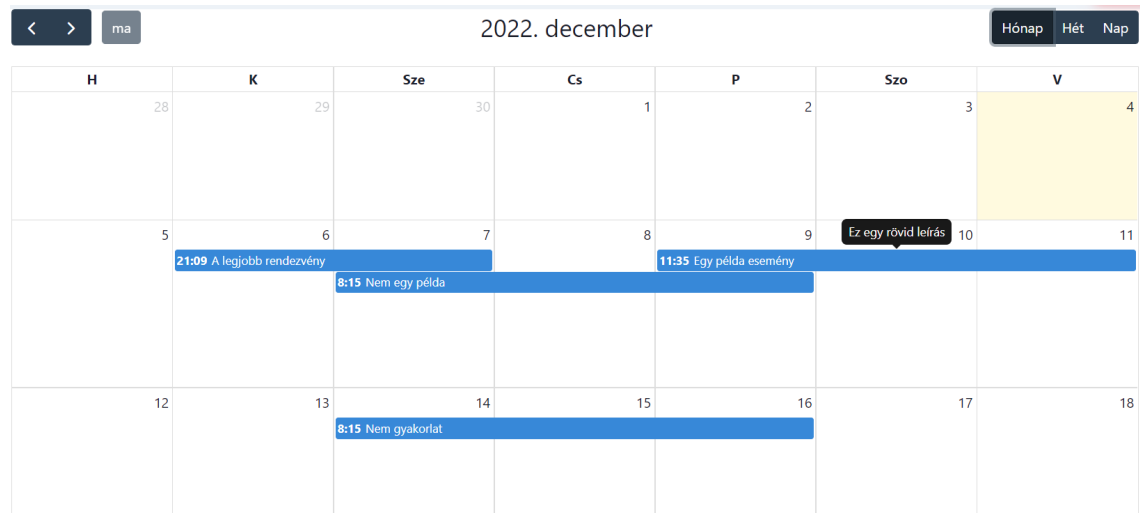
A kontroller validálja a kérést, majd ahogy a fenti kódrészleten is látható, az API végponthoz érkezett módosítási hívás lefut a szolgáltatások rétegbe, ahol ismételt validáció után az adatbázisba is eljut a felhasználó változtatása.

5.6. Naptár megjelenítése

A naptár megjelenítéséért a Calendar.js komponens a felelős. Elvégzi a FullCalendar által szolgáltatott objektum kinézetének konfigurálását, majd elvégzi az események lekérdezését. Ismeretlen felhasználók esetén a naptárban található összes eseményt lekéri a backend-ről, bejelentkezett felhasználó esetén pedig, annak szűrőinek megfelelően jeleníti meg az eseményeket. A kérés, ahogy említettem beérkezik a backend-re bejelentkezett felhasználók esetében az alábbi végpontra:

```
@GetMapping("/calendar/filtered")
@Operation(summary = "Get the Events filtered by the User, that can be consumed by
↳ FullCalendar",
    responses = {
        @ApiResponse(description = "All of the Events",
            content = @Content(mediaType = "application/json",
                array = @ArraySchema(schema =
                    ↳ @Schema(implementation =
                        ↳ FullCalendarEventDto.class))))),
        @ApiResponse(responseCode = "400", description = "Events not
↳ found")})
@LogExecutionTime
public ResponseEntity<List<FullCalendarEventDto>> getFullCalendarFilteredEvents()
↳ {
    if(eventService.findAllFullCalendarFilteredEvents().isEmpty()) {
        return ResponseEntity.badRequest().build();
    }
    return ResponseEntity.ok(eventService.findAllFullCalendarFilteredEvents());
}
```

Érdekességgépp megfigyelhető a végponton található annotációk formájában az OpenAPI & Swagger egyik legnagyobb gyengesége véleményem szerint, rendkívül megnehezíti a kontroller osztályokban található végpontok olvashatóságát. A metódus olyan speciális FullCalendarEventDto osztályban adja vissza a talált eseményeket, melyeket a FullCalendar által szolgáltatott komponense gond nélkül be tud fogadni, így a frontend-nek nem kell semmiféle konverziós műveleteket végrehajtania.



5.4. ábra. Az események megjelenítése a főoldalon havi bontásban

A naptárban a FullCalendar segítségével megjeleníthetjük napi, heti, vagy havi bontásban is, ugorhatunk a jelen napra, lépkedhetünk az időegységek között, továbbá helyesen megjelennek az események. Egérrel egy esemény felé helyezkedve a rendezvény rövid leírása is megjelenik.

5.7. Esemény részletes nézete

A naptárban egy eseményre való kattintás után a felhasználónak lehetősége van egy rendezvény minden paraméterének megtekintésére. A kattintás után a Calendar.js komponens az /eventview/<event.id> címre visz minket, ahol az event.id értelemszerűen az esemény azonosítója. Egy esemény megjelenítéséért a EventViewer.js komponens a felelős, aki a megadott azonosító alapján lekérdezi az API-ból a rendezvény DTO-ját.



5.5. ábra. Egy esemény részletei

Az esemény egy Chakra UI segítségével megformázott kártya formájában jelenik meg, amely egy teljesen újrahasználgató komponens, és a napi bontásban is használatra kerül.

5.8. Események napi bontás nézete

A naptárban lehetőségünk nem csak egy eseményre kattintani, hanem egy egész napra is. Ilyenkor a `DayViewer.js` komponens lekérdezi az API-ból az összes arra a napra bejegyzett eseményt és megjeleníti őket a már korábban említett `Event.js` komponens által kezelt kártyás formában egy apró fejléccel a `/dayview/<dátum>` oldalon.



5.6. ábra. Egy nap eseményeinek megjelenítése

A `DayViewer.js` az `api/event/day` elérhetőségen, megfelelő kérésparaméterrel együtt tudja lekérdezni az arra a napra kiírt eseményeket.

```
const fetchDay = () => {
  fetch(`http://localhost:8080/api/event/day?date=${id}`, {
    method: 'GET',
    headers: {
      'Content-Type': 'application/json',
      'Accept': 'application/json'
    }
  })
  .then((response) => response.json())
  .then(data => setDayEvents(data));
}
```

Ez a kérés eljut az események logikáját megvalósító `EventService` komponensbe, ahol egy egyszerű logika segítségével ki is szortírozzuk a számunkra fontos eseményeket.

```
@Transactional
public List<EventDto> findEventsByDay(String day) {
    List<Event> events = eventRepository.findAll();
    for(Iterator<Event> iterator = events.iterator(); iterator.hasNext();) {
        Event e = iterator.next();
        String monthWithStartDay = e.getStart().getMonth().getValue() + "-" +
            e.getStart().getDayOfMonth();
        String monthWithEndDay = e.getEnd().getMonth().getValue() + "-" +
            e.getEnd().getDayOfMonth();
        if(!monthWithStartDay.equals(day) && !monthWithEndDay.equals(day)) {
            iterator.remove();
        }
    }
    return eventMapper.eventsToDtos(events);
}
```

A szolgáltatás akkor rakja bele az eredmény listába a rendezvényt, ha vagy a kezdő, vagy végző dátumának napja megegyezik a kért nap paraméterrel. Visszatérése után az API visszaadja az eseményeket egy listában, melyet a `DayViewer.js` feldolgoz és megjelenít egymás után a felhasználó számára.

5.9. Esemény létrehozása és módosítása

Az események létrehozását és módosítását egy közös képernyőn érjük el a `/event/<'new'>` vagy esemény azonosító címen. Itt egy űrlapon kitölthetjük egy új rendezvény adatait, vagy felülírhatjuk egy már meglévőét.

Eseménykezelő

Program neve *

Teszt esemény kreálás

Az esemény neve, ahogyan szerepelni fog a naptárban.

Reszort *

Szent Schönherz Senior Lovagrend

A rezsort, ahová az esemény tartozik.

Kör *

Kötelező körhöz rendelni az eseményt.

Esemény kezdete *

2022. 12. 01. 21:09

Az esemény kezdési ideje nem lehet korábbi időpont.

Esemény vége *

2022. 12. 07. 21:09

Az időpont, amikor véget ér az esemény.

5.7. ábra. Az eseménykezelő űrlap

Az űrlapon találhatóak kötelező mezők, melyek validálása igen egyszerű a Chakra UI-ban található FormControl komponens segítségével. A beviteli mező értéke alapján megszabhatjuk egy saját metódusban, hogy milyen fajta értékeket szeretnénk átengedni, illetve milyen üzenetek jelenjenek meg a beviteli mező alatt.

```

const evaluateEndInputText = () => {
  if (!(isEndError() || isEndDateBeforeCurrentDate())) {
    return <FormHelperText>
      Az időpont, amikor vége az eseménynek.
    </FormHelperText>
  }
  if (isEndError()) {
    return <FormErrorMessage>
      Az esemény befejezési ideje kötelező.
    </FormErrorMessage>
  }
  if (isEndDateBeforeCurrentDate()) {
    return <FormErrorMessage>
      Az esemény befejezési ideje nem lehet korábbi időpont.
    </FormErrorMessage>
  }
}

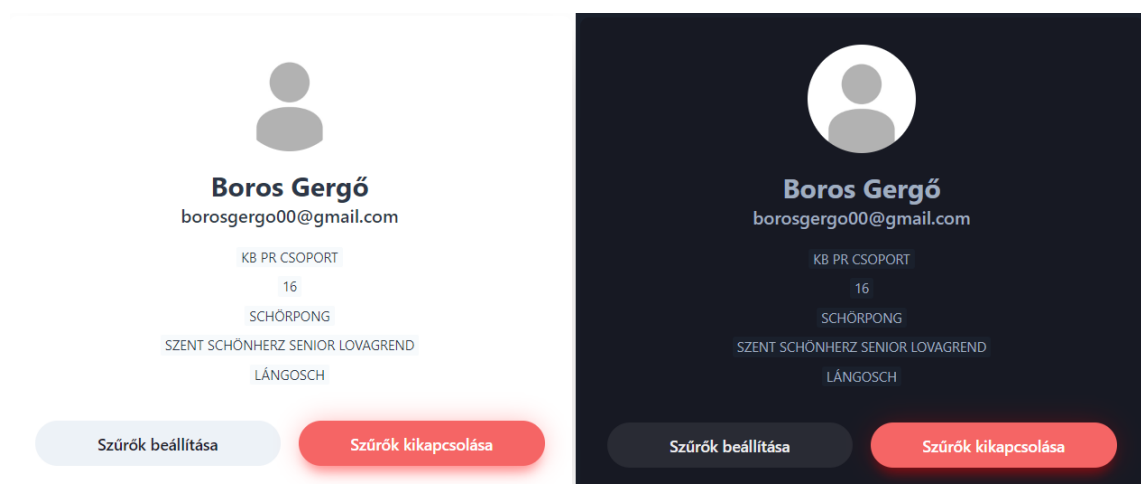
```

A fenti kódrészletben látható, hogy az eseményét végének dátumát miképpen validálja az alkalmazás. Természetesen kötelező megadni valamit, illetve ezen felül nem lehet a jelen időpontnál, vagy a kezdési időpontnál korábbi dátum. Ezen feltételek alapján adjuk vissza dinamikus a FormControl komponensnek a szövegeket, melyek majd megjelenítésre kerülnek a felületen.

Minden helyes beviteli mező esetén, ha az űrlap alján található "Létrehozás" vagy "Módosítás" gombra kattinthatunk. Ezek után a frontend indít egy kérést a backend felé, melyen keresztül a létrehozás vagy módosítás ténye lefut egészen az adatbázisig, így a főoldalra visszairányítás után már látható is az általunk végzett művelet eredménye.

5.10. Éjszakai mód

Az alkalmazásban a Chakra UI segítségével implementáltam éjszakai módot is, amely nincs olyan erős, káros hatással az emberi szemre.



5.8. ábra. A felhasználó profilja világos és sötét módban

A sötét mód minden komponensre megfelelően működik, minden szöveg, gomb, háttér dinamikusan változik attól függően, hogy a felhasználó éppen melyik módot választotta. A kinézetet a navigációs sávon található napocska, vagy hold segítségével lehet állítani. A gombhoz egy kapcsoló van rendelve, melyet felhasználva a Chakra UI `useColorModeValue` metódusának segítségével minden egyes komponensünkre beállíthatjuk, hogy az egyes módok esetében milyen színt vegyen fel.

6. fejezet

Tesztelés

A szoftverfejlesztés ágazatán belül a tesztelés egy érdekes, sokoldalú és komplex feladat. Az egyre fejlődő terület és bonyolultabbnál bonyolultabb alkalmazások esetében manapság elengedhetetlen alkalmazásunk tesztelése. A tesztek segítségével visszajelzést kaphatunk az alkalmazásunk működéséről és megtudhatjuk, hogy éppen az elvárt működés szerint funkcionál-e. A korai hibafelismeréssel növelhetjük az elkészült kód, alkalmazás minőségét.

6.1. Tesztek fajtái

A tesztelés egy dinamikus ellenőrzési forma, amely során a vizsgált forráskód végrehajtásra kerül. A jelen szakdolgozat elkészítése során egység (unit) és integrációs teszteket implementáltam és futtattam. A tesztek megírása során a Given-When-Then (GWT) tesztstruktúrálási módszert alkalmaztam, amely az alábbi sablont határozza meg:

- **Given:** van egy adott állapot
- **When:** végrehajtunk egy tevékenységet
- **Then:** ezt az eredményt kapjuk

Ezeket felül nagy hangsúlyt fektettem arra, hogy a Spring Testing és Mockito által nyújtott tesztelési konfigurációk közül a lehető legtöbbet kipróbáljam a projekten.

6.1.1. Egység tesztek

A unit tesztek a kód egy logikailag könnyen szeparálható egységét tesztelik, amelyek tesztelési szempontból már nem bonthatóak kisebb egységekre, a jelen projekt esetében ezek egy-egy metódust jelképeznek. A teszteseteket vegyesen, a forráskódot és a specifikációt is alapul véve terveztem meg. A tesztesetekre általánosságban az alábbiak jellemzőek:

- bonyolultságuk alacsony
- bemenő értékeik száma minél alacsonyabb
- izoláltak a többi egységtől
- egyetlen egy viselkedést tesztel

A Java világában az egység tesztek írására a legelterjedtebb keretrendszer a JUnit, amelyet az általam használt Spring Testing modul is tartalmaz.

6.1.1.1. Repository rétegbeli teszt

Egy repository rétegbeli osztály teszteléséhez először is létrehoztam egy közös osztályt, amelyet az összes tesztelendő repository beimportálhat:

```
@TestConfiguration
public class RepositoryTestConfig {
}
```

A `@TestConfiguration` annotáció segítségével módosíthatjuk a Spring-es applikációnk kontextusát futási időben. Segítségével felülírhatunk különböző Bean definíciókat, vagy helyettesíthetjük őket mock-olt másolataikkal. A jelen projekt esetében erre nem volt szükség a Mockito miatt, de lehetőséget ad későbbi bővítéseknek.

Egy repository rétegbeli osztály tesztosztálya az alábbiak szerint került beállításra:

```
@DataJpaTest
@Import({RepositoryTestConfig.class})
@AutoConfigureTestDatabase(replace = <teszt adatbázis>)
class ExampleRepositoryTest {
    @Autowired
    private ExampleRepository exampleRepository;
}
```

A következőket érdemes észrevenni az osztály deklarációján:

- A `@DataJpaTest` annotáció segítségével csak az érintett entitások és repository-k fognak betöltődni a kontextusba.
- Az `@Import` segítségével behúzhatjuk a korábban említett tesztkonfigurációs osztályunkat.
- Az `@AutoConfigureTestDatabase` segítségével pedig meghatározhatunk egy teszt adatbázist, nem kell az alkalmazásét használni, erre szolgálhat például egy memóriabeli H2 adatbázis is.

- Az @Autowired annotációval injektálhatjuk a tesztelendő repository osztályunkat a tesztosztályba.

Az esemény entitásokon keresztül bemutatva pedig az alábbiak szerint alakult egy unit teszt:

```
@Test
void testSaveAndFindAll() {
    // given
    int numberOfEventsInDatabase = eventRepository.findAll().size();
    Event event = new Event();
    event.setName("Test Event");
    eventRepository.save(event);

    // when
    List<Event> eventList = eventRepository.findAll();

    // then
    assertEquals(numberOfEventsInDatabase + 1, eventList.size());
}
```

Jól látható a teszten a kommentekkel jelzett GWT sablon. A tesztben egy egyszerű esemény mentést, továbbá az események lekérdezését teszteltük az eventRepository osztály által. A módszer végén assert metódussal ellenőrzi a végkimenetelt.

6.1.1.2. Egyszerű kontroller teszt

A kontroller réteg unit tesztelése során a munkát egy RepositoryTestConfig osztályhoz hasonló WebMvcConfig osztály deklarálásával kezdtem, amely pontosan ugyan azokkal a funkciókkal rendelkezik mint a RepositoryTestConfig osztály. Létezésének az oka, hogy egy kontroller rétegbeli unit tesztelése esetében általában más konfigurációra lehet szükségünk. A jelen projekt esetében nem igazán került használatba, de további lehetőséget nyújt a tesztek testreszabására:

```
@TestConfiguration
public class WebMvcTestConfig {
}
```


Egy egyszerű controller teszt osztály példa deklarációja az alábbiak szerint nézhet ki:

```
@SpringBootTest(classes = WebMvcTestConfig.class)
@AutoConfigureMockMvc
class ExampleControllerTest {

    @Autowired
    private MockMvc mockMvc;

    @MockBean
    private ExampleService exampleServiceMock;
}
```

A `@SpringBootTest` annotáció használatával megteremtjük a megfelelő alkalmazás kontextust a tesztosztálynak az `@AutoConfigureMockMvc`-vel pedig a Spring előre bekonfigurálja az `@Autowired` által beinjektált mock-olt objektumunkat, amely a controller felé intézi majd az API hívásokat a teszt során.

A kör entitás controllerén keresztül nagyszerűen bemutatatható az így felkonfigurált tesztet:

```
@Test
void testGettingCircles() throws Exception {
    // given
    List<CircleDto> circles = List.of(new CircleDto(), new CircleDto());

    // when
    when(circleServiceMock.findAll()).thenReturn(circles);

    // then
    this.mockMvc.perform(get("/api/circle")
        .contentType(MediaType.APPLICATION_JSON))
        .andExpect(status().isOk())
        .andExpect(content().contentType(MediaType.APPLICATION_JSON))
        .andExpect(jsonPath("$", hasSize(2)));

    verify(circleServiceMock, times(2)).findAll();
    verifyNoMoreInteractions(circleServiceMock);
}
```

A teszt során létrehozunk két új kört a **Given** részben. A körökhöz tartozó mock-olt Service osztály működését a Mockito keretrendszerrel szimuláljuk a **When** részben, így ténylegesen csak a controller osztály működését figyelhetjük meg. Az autokonfigurált mockMvc objektummal meghívjuk az API végpontot és különféle eredményeket, formátumokat várunk el a kapott értéktől. Legvégül pedig verify metódusokkal ellenőrizzük, hogy a mockolt szolgáltatás osztály is helyesen működött-e.

6.1.1.3. Kontroller WebMvc teszt

A WebMvc-s tesztelés során lehetőségünk van elhagyni az `@AutoConfigureMockMvc` annotációt is, ez a következő kódot eredményezi:

```
@WebMvcTest(ExampleController.class)
@ContextConfiguration(classes = {ExampleControllerWebMvcTest.ControllerConfig.class})
class ExampleControllerWebMvcTest {

    @Autowired
    private MockMvc mockMvc;

    @MockBean
    private ExampleService exampleServiceMock;

    @TestConfiguration
    static class ControllerConfig {
        @Bean
        public ExampleService exampleServiceMock() {
            return mock(ExampleService.class);
        }
    }
}
```

A `@WebMvcTest` betölti a kontroller osztály számára szükséges összes Bean-t, és egy előkonfigurált `MockMvc` környezetet kapunk. A `ControllerConfig` osztályban pedig a Mockito segítségével beállítjuk a teszteléshez szükséges szolgáltatás Bean szimulálását. Ebben az esetben egy teszteset teljesen megegyezik az előző bekezdésben bemutatott példával.

6.1.1.4. Önálló kontroller WebMvc teszt

Lehetőségünk van olyan WebMvc alapú kontroller tesztet is írni, amely során egyáltalán nem töltjük be a Spring kontextusát, azonban ebben az esetben saját magunknak kell felépíteni a környezetet és injektálni a mock-okat:

```
@ExtendWith(MockitoExtension.class)
class ExampleControllerStandaloneWebMvcTest {

    private MockMvc mockMvc;

    @Mock
    private ExampleService exampleServiceMock;

    @BeforeEach
    public void setUp() {
        ExampleController exampleController = new
        ↪ ExampleController(exampleServiceMock);
    }
}
```

```

        mockMvc = MockMvcBuilders.standaloneSetup(exampleController)
            .setControllerAdvice(new GlobalExceptionHandler())
            .build();
    }

    @AfterEach
    public void validate() {
        validateMockitoUsage();
    }

```

A `setUp` metódusban kedvünkre bekonfigurálhatjuk a `mockMvc` osztályunkat, és ez minden teszteset előtt le fog futni. A tesztesetek után a `validateMockitoUsage` explicit ellenőrzi a keretrendszer állapotát, hogy kiszűrje a Mockito helytelen használatát.

6.1.2. Integrációs tesztek

Az egység teszteléssel ellentétben az integrációs tesztelés egy olyan módszertan, amely során a kódban található modulokat egyesítjük és egy csoportként teszteljük őket. A folyamat során meggyőződünk róla, hogy a komponenseink nem csak külön-külön, hanem együtt, egy rendszert alkotva is helyesen működnek, megfelelnek a funkcionális követelményeknek. A jelen dolgozat esetében többféle integrációs tesztet is implementáltam, ebben az alfejezetben ezek kerülnek bemutatásra.

6.1.2.1. Service rétegbeli integrációs teszt

A jelen projekt esetében talán ez a módszer volt a legegyszerűbb tesztelési technika. Egy `services` osztályhoz az alábbi példakód segítségével készíthetünk egy tesztosztályt:

```

@SpringBootTest
@Transactional
class ResortServiceTestIT {

    @Autowired
    private ResortService resortService;
}

```

Itt is megjelenik a korábban már tárgyalt `@SpringBootTest`, továbbá a `@Transactional` annotációval érhetjük el, hogy az összes teszteset tranzakcionális műveletként fusson le. Ez azt eredményezi, hogy a teszt lefutása után minden adatbázissal kapcsolatos művelet alapértelmezés szerint rollback-elésre kerül.

```

@Test
void testGettingEventByDay() {
    // given
    CreateEventDto event = new CreateEventDto();
    event.setCircle("KB PR Csoport");
    event.setResort("Egyéb");
    event.setStart("2022-12-02 12:00");
    event.setEnd("2022-12-03 13:00");
    event.setName("Teszt Esemény");
    event.setPlace("Teszt Helyszín");
    event.setDescription("Teszt Részletek");

    // when
    EventDto savedEvent = eventService.createEvent(event);

    // then
    assertNotNull(savedEvent);
    assertNotNull(eventService.findEventsByDay("2022-12-02"));
}

```

Megfigyelhetjük, hogy a roppant egyszerű példa teszt metódusban nem kerül semmi mock-olásra, az igazi injektált objektum metódusát hívjuk meg, majd assert-ekkel ellenőrizzük a kapott eredményt.

6.1.2.2. Egyszerű kontroller integrációs teszt

Lehetőségünk van olyan integrációs tesztek írására is, mely futtatásakor egy teljesen inicializált Spring webalkalmazást kapunk.

```

@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
class ResortControllerIT {
    private static final String BASE_URL = "/api/event";
    @Autowired
    private WebTestClient webTestClient;
}

```

Ezt a `@SpringBootTest` annotáció segítségével tehetjük meg. A tesztesetek minden esetben egy REST alapú webes *klienset* konfigurálnak fel, amely meghívja, majd a kontroller egy bizonyos végpontját.

```
@Test
void testGettingResorts() {
    // given, when
    ParameterizedTypeReference<List<Resort>> responseType = new
    ↪ ParameterizedTypeReference<>() {};

    EntityExchangeResult<List<Resort>> response =
    ↪ webTestClient.get().uri(BASE_URL)
        .exchange().expectStatus().isOk()
        .expectBody(responseType).returnResult();
    List<Resort> responseBody = response.getResponseBody();

    // then
    assertEquals(HttpStatus.OK, response.getStatus());
    assert responseBody != null;
    assertFalse(responseBody.isEmpty());
}
```

A kiértékelés ebben az esetben a HTTP válasz megvizsgálásával egyenlő. Asszertálásokkal ellenőrizhetjük az adott válaszkódot, illetve a válasz body tartalmát is.

6.1.2.3. Kontroller MockMvc integrációs teszt

Tudunk a már jól bevált, korábban bemutatott MockMvc-t használó integrációs tesztet is írni, amely teljes Spring kontextussal teszteli a kontroller réteget. A függőségek nincsenek mock-olva, továbbá megengedjük, hogy az összes metódushívás az összes rétegen keresztül menjen.

```
@SpringBootTest
@Transactional
@AutoConfigureMockMvc
class ResortControllerMockMvcIT {

    private static final String BASE_URL = "/api/resort";

    @Autowired
    private MockMvc mockMvc;

}
```

A tesztesetek kísértetiesen hasonlítanak a már korábban bemutatott MockMvc egységtesztekhez, annyi különbséggel, hogy a metódusban semmilyen mock-olásra nincs szükség, egyből munkának indíthatjuk a mockMvc objektumunkat.

7. fejezet

Összegzés

A legutolsó fejezetben röviden összefoglalom a szakdolgozat elkészítése közben megfogalmazódott továbbfejlesztési ötleteimet, majd befejezem a dolgot a záró gondolataimmal.

7.1. Továbbfejlesztési lehetőségek

A fejlesztés során természetesen az előre lefektetett követelményeken túl is jutottak eszembe olyan plusz elemek, melyeket később bele lehet építeni a jelenlegi projektbe. Ezek az ötletek a szakdolgozati munkafolyamat vége felé körvonalazódtak meg bennem, amikor már volt lehetőségem átlátni az egész alkalmazást. Mivel az alkalmazás felhasználói felülete véleményem szerint egészen letisztult és felhasználóbarát lett, ezért inkább az új funkciók implementálásán gondolkodtam. Asztali számítógépek esetében meg vagyok elégedve a felület kinézetével, azonban egy fejlesztési lehetőség, hogy a projekt UI-ja reszponzív legyen. A legtöbb modul megfelelően néz ki telefonos vagy tabletes képernyőn is azonban vannak olyanok, mint például a navigációs sáv, melyek kinézetén és élvezhetőségén lehetne javítani. Ezen felül megjelenhetnének különböző pop-up, toast üzenetek bizonyos felhasználói akciók után.

Egy bizonyos időpontnál régebbi események számára létrehoznék egyfajta archív események oldalt is, ahol nem kellene a naptárban visszalépkedve visszakeresni az összes eseményt, hanem egy helyen láthatóak lennének az események, amelyek a múltban megrendezésre kerültek.

A kollégisták egy esemény népszerűsítése közben rengeteg időt ölnek bele a plakátkészítésbe. Személy szerint mindig szeretek megállni a kollégiumban és megnézegetni őket, észrevenni kis részleteket bennük, csodálkozni rajtuk, hogy milyen igényesek. Mindenképpen létrehoznék egy plakát "Hall-Of-Fame" oldalt, ahol a legszébb plakátok találhatók. Az esemény képernyőjén lehetne egy upvote-ot, vagy downvote-ot adni a plakátnak. A legtöbb upvote-ot szerző plakátok szerepelnének ezen az új képernyőn. Ezeken felül a projekt továbbfejlesztéseként bevezetnék minden esemény esetén egyfajta számlálót is, mely számon tartja, hogy hány ember

tekintette meg az esemény hirdetését. Ezt a számlálót az adott öntevékeny kör tagjai tekinthetnék meg, ezzel könnyebben tudnának tervezni a szervezési folyamatok során. Ennek kiegészítéseként pedig a Facebook-on található eseményekhez hasonlóan a bejelentkezett felhasználóknak meg lenne a lehetősége, hogy bejelöljék egy egytől háromig terjedő skálán mennyire valószínű az, hogy megjelennek az adott eseményen.

7.2. Végszó

Összességében visszatekintve a projektet sikeresnek könyveltem el magamban. Sikertült megfelelni az előre felállított funkcionális és nem funkcionális követelményeknek is. Az összes elképzelt funkcionalitást sikerült a gyakorlatba is átültetnem, és reményeim szerint tényleg sikerült egy igényes munkát kiadnom a kezeim közül.

Bátran mondhatom, hogy a feladathoz igen motiváltam álltam hozzá, hiszen tényleg úgy éreztem rengeteget tanulhatok belőle. A tervezési és korai fejlesztési folyamat során végig azt a központi elvet tűztem ki magamnak, hogy hogyan kell rendesen megcsinálni, implementálni, használni az igénybe vett technológiát, nem pedig azt, hogy a lehető leghamarabb kész legyen a feladattal. E szemlélet teljesítésének rengeteg irodalomkutatás volt az előkövetelménye, azonban ez a folyamat nagyon sok mindenre megtanított.

Először is az igényességre, mert ha valamit csinált az ember, azt csinálja rendesen. Nem szabad megelégedni a nem teljesen helyes megoldásokkal, melyeket "majd később átír" az ember. Rengeteget code debt-et tudunk így felhalmozni, melyek később egy saját magunk ellen vétett gonosztettekként köszönhetnek vissza.

Megtanított a türelemre, hiszen egy nagyobb volumenű feladat esetében semmi sem megy mindig gördülékenyen. Megtanultam kezelni a hibákkal járó stresszt, továbbá nagyban fejlesztette azt a képességemet, hogyan álljak neki egy probléma, nem működő funkció, rejtélyes bug megfejtésének.

A projekt megtanította azt a fajta asszertív gondolkodásmódot, mely segítségével rendszerezni tudom a már elsajátított tudást, a még hátralevő feladatokat. Egy személyként viselni a projektvezetői, üzleti szervezői és fejlesztői sapkákat egy teljes fejlesztési munkafolyamaton átívelő tapasztalatokat adott. Ez a fajta rendszerszintű látásmód úgy gondolom, nagyon hasznos lesz számomra jövőbeli projektet során, továbbá az ipar világában.

A feladat megvalósítása előtt csak felületes ismereteim voltak a Spring keretrendszerről, nem éreztem magam igazi nagyigényű felhasználónak. A frontend technológiák túlnyomó része teljesen idegen vizek voltak számomra, ezért is örülök nagyon utólag a választott eszközöknek. Csak pozitívan tudok nyilatkozni róluk, minden percét élveztem a velük való közös munkának. Mind a Spring és React esetében is kellemes meglepetésként ért a hihetetlenül felhasználóbarát és részletes hivata-

los dokumentáció, továbbá rendkívüli elterjedtségük miatt, szinte mindig azonnal választ találtam a kérdéseimre.

Legvégül pedig a megszerzett tervezési és fejlesztési tapasztalatokat kifejezetten hasznosnak tartom a jövőre nézve, melyek akármilyen másik projektre nézve relevánsak, de nem szabad elfelejteni, hogy a tanulási folyamatnak egy szoftverfejlesztő számára sosincs vége, biztos vagyok benne, hogy még rengeteg új tudással fogok gazdagodni a dolgozatban körüljárt technológiák esetében.

Köszönetnyilvánítás

Végezetül pedig szeretném megköszönni két embernek, akik segítséget nyújtottak a szakdolgozatom elkészítésében.

Először is, Szabó Gergely hallgatótársamnak az architektúrával kapcsolatos tanácsait, továbbá hogy segített kiválasztani a dolgozatom témáját és tippeket adott, útmutatást nyújtott az AuthSCH implementálása során. Sok álmatlan éjszakát spórolt meg nekem.

Legvégül pedig Imre Gábor konzulensemnek, akivel már témalabor óta együtt dolgozunk, és minden önálló tárgy esetében nagy szabadságot adott nekem. Minden félévben a témalabort, az önálló labort és a szakdolgozat-készítést élveztem a legjobban.

Irodalomjegyzék

- [1] Atlassian Bitbucket: What is version control (2022.11.25). <https://www.atlassian.com/git/tutorials/what-is-version-control>.
- [2] D. Hardt, Ed., Microsoft: Rfc 6749: The oauth 2.0 authorization framework (2022.11.27). <https://www.rfc-editor.org/rfc/rfc6749#section-1>.
- [3] Docker: Hivatalos dokumentáció (2022.11.24). <https://docs.docker.com/get-started/overview/>.
- [4] FullCalendar: Hivatalos dokumentáció (2022.11.24). <https://fullcalendar.io/docs>.
- [5] GitHub: Understanding github actions (2022.11.25). <https://docs.github.com/en/actions/learn-github-actions/understanding-github-actions>.
- [6] IBM Cloud Education: What is java spring boot? (2022.11.25). <https://www.ibm.com/cloud/learn/java-spring-boot>.
- [7] IntelliJ IDEA: Hivatalos dokumentáció (2022.11.25). <https://www.jetbrains.com/help/idea/discover-intellij-idea.html>.
- [8] KSZK: Authsch (2022.11.27). <https://auth.sch.bme.hu/>.
- [9] Mapstruct: Hivatalos dokumentáció (2022.11.25). <https://mapstruct.org/>.
- [10] npm: Hivatalos dokumentáció (2022.11.24). <https://docs.npmjs.com/>.
- [11] PostgreSQL: Hivatalos dokumentáció (2022.11.25). <https://www.postgresql.org/docs/current/intro-what-is.html>.
- [12] React: Introducing hooks (2022.11.28). <https://reactjs.org/docs/hooks-intro.html>.
- [13] React: A javascript library for building user interfaces (2022.11.28). <https://reactjs.org/>.
- [14] Sonarcloud: Hivatalos dokumentáció (2022.11.25). <https://docs.sonarcloud.io/>.

- [15] Spring: Spring aop (2022.11.27). <https://docs.spring.io/spring-framework/docs/current/reference/html/core.html#aop>.
- [16] Spring: Spring boot hivatalos dokumentáció (2022.11.25). <https://spring.io/projects/spring-boot>.
- [17] Spring: Spring data jpa (2022.11.27). <https://spring.io/projects/spring-data-jpa>.
- [18] Spring: Spring framework hivatalos dokumentáció (2022.11.23). <https://docs.spring.io/spring-framework/docs/current/reference/html/core.html#beans>.
- [19] Spring: Spring security (2022.11.27). <https://spring.io/projects/spring-security>.
- [20] Spring Data JPA: Query methods (2022.11.27). <https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#jpa.query-methods>.
- [21] Spring Initializr: Spring initializr (2022.11.25). <https://start.spring.io/>.
- [22] Srirangan: *Apache Maven 3 Cookbook*. 2011, Packt Publishing.
- [23] Swagger: About swagger specification (2022.11.27). <https://swagger.io/docs/specification/about/>.
- [24] Takahiko Kawasaki: Diagrams and movies of all the oauth 2.0 flows (2022.11.27). <https://darutk.medium.com/diagrams-and-movies-of-all-the-oauth-2-0-flows-194f3c3ade85>.
- [25] WebStorm: Hivatalos dokumentáció (2022.11.25). <https://www.jetbrains.com/help/webstorm/meet-webstorm.html>.