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

Jankó Júlia

ONLINE IDŐPONTFOGLALÓ FULLSTACK WEBALKALMAZÁS

KONZULENS

Kövesdán Gábor

BUDAPEST, 2024

Tartalomjegyzék

HALLGATÓI NYILATKOZAT

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

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

Kelt: Budapest, 2024. 12. 05

...…….
Jankó Júlia

Összefoglaló

A szolgáltatóiparban manapság már alapvető elvárás, hogy a szolgáltató rendelkezzen egy webes felülettel. A szolgáltatóipar minden területén találkozhatunk rengeteg különböző megoldással, melyek ugyanolyan vagy nagyon hasonló funkciókat látnak el, akár kozmetikai, egészségügyi, vagy fitnesz területen. Ezeken az oldalakon tájékozódhatunk a nyújtott szolgáltatásokról és időpontot foglalhatunk ezekre. Így rengeteg szolgáltató kénytelen megoldani olyan problémákat, amit az összes előtte levő szolgáltató már egyszer megoldott. A vevők szempontjából is előnytelen ez a jelenség, mivel ahány szolgáltató, annyi féle különböző weboldallal találkozhat. Az oldalakon a funkciók hasonlók, viszont az oldal megjelenése és navigáció különböző, nehézkes.

Dolgozatomban egy olyan időpont foglaló alkalmazást mutatok be, mely megoldást ad ezekre a nehézségekre. Webalkalmazásomban lehetőségük van a különböző szolgáltatóknak regisztrálni és szolgáltatást nyújtani anélkül, hogy saját weboldalt fejlesztenének. Emellett a felhasználók egy egységes felületen kezelhetik foglalásaikat, valamint válogathatnak a különböző szolgáltatók között az egyéni preferenciájuk szerint.

Megoldásomban a felhasználóknak lehetőség van keresni több szempont alapján a szolgáltatók között. Időpontot foglalhatnak az elérhető szolgáltatóknál, menedzselhetik foglalásaikat és saját profiljukat. Szolgáltatóként az általános felhasználói funkciók mellett lehetőség van az általuk nyújtott szolgáltatások menedzselésére és elérhetőségeik megadására.

Abstract

In the service industry, it is now a basic requirement that the service provider has a web interface. In all areas of the service industry, we can find many different solutions that perform the same or very similar functions, whether in the fields of cosmetics, health, or fitness. On these pages, you can find out about the services provided and book an appointment for them. Thus, many service providers are forced to solve problems that all the service providers before them have already solved once. This phenomenon is also disadvantageous from the customers' point of view, since there are as many different websites as there are service providers. The functions on the pages are similar, but the page appearance and navigation are different and difficult.

In my thesis, I present an appointment booking application that provides a solution to these difficulties. In my web application, different service providers have the opportunity to register and provide services without developing their own website. In addition, users can manage their reservations on a single interface and choose between different service providers according to their individual preferences.

In my solution, users have the opportunity to search among service providers based on several criteria. They can book an appointment with the available service providers, manage their reservations and their own profile. As a service provider, in addition to general user functions, it is possible to manage the services they provide and provide their contact information.

1 Bevezetés

Az internetnek köszönhetően akár mindennapi ügyeinket és teendőinket végezhetjük otthonunk kényelméből. Ilyen például a bankolás, home-office, utazások, események szervezése. Mivel sokan a telefonálás vagy személyes ügyintézés helyett interneten végzik teendőiket, így a szolgáltatási szektor is alkalmazkodott ehhez a változáshoz, ritka az olyan szolgáltató, akinek nincsen weboldala. Ezeken a weboldalakon megtekinthetjük a szolgáltató és a szolgáltatás részleteit, árakat, és gyakran foglalhatunk időpontot is ezekre a szolgáltatásokra.

Mivel rengeteg szolgáltató van a világon, ezért rengeteg különböző weboldallal találkozhatunk. Különbözhetnek stílusukban, elrendezésben, az oldal navigációjában, de általában mindegyiknek az a célja, hogy a látogató tájékozódhasson a szolgáltatásokról, és tudjon időpontot foglalni a szolgáltatásra. Mivel majdnem minden szolgáltatónak szüksége van egy ilyenre, ezért gyakran megoldják újra és újra ugyanazokat a feladatokat, ami idő és pénz igényes.

Ha igénybe szeretnénk venni egy szolgáltatást, a nagy választék miatt gyakran optimalizálni kezdünk a saját prioritásaink alapján. Általában szeretnénk minél olcsóbban szeretnénk szolgáltatást kapni. Gyakran fontos, hogy minél közelebb legyen ez otthonunkhoz, milyen minőségű munkát végez a szolgáltató, vagy hogy mennyire szimpatikus. Ehhez igyekszünk sok szolgáltatót összevetni, hogy kiválasszuk az optimálist, azonban ehhez először találnunk kell ilyen szolgáltatókat. Navigálnunk és tájékozódnunk kell a sok különböző oldalon, összevetni a különböző információkat.

Megoldásomban egy olyan webes alkalmazást készítettem, amely lehetővé teszi a felhasználóknak a szolgáltató keresést és időpont foglalást egy egységes felületen. A szolgáltatóknak nem kell saját webalkalmazást készítenie, hanem beregisztrálhatják szolgáltatásaikat, és jobban megtalálhatóvá válnak a felhasználók számára.

Ebben az alkalmazásban szükségem volt egy weboldalra, amely esztétikusan és könnyedén teszi elérhetővé ezeket a funkciókat a felhasználó számára. Ezt a nagyon elterjedt React frontend technológiával oldottam meg. Ezelőtt még nem fejlesztettem React technológiában, ezért fontosnak tartottam, hogy megismerjem és elsajátítsam. A felhasználói felület mellett szükségem volt egy szerveroldali alkalmazásra, mely kiszolgálta a weboldal kéréseit, és meg tudtam benne fogalmazni az oldalon található

funkciók logikáját. Erre a Spring Java alapú keretrendszert használtam, azonban Java helyett a fejlett és népszerű Kotlin programozási nyelven. A szerveroldali alkalmazásom mellé szükségem volt egy adatbázisra is, amelyben tároltam a szükséges információkat. Erre a Microsoft SQL Server által nyújtott relációs adatbázist használtam.

Az időpont foglaló alkalmazásomnál a fodrászati vagy kozmetikai szolgáltatás foglalást tartottam szem előtt, így a dolgozatomban a szolgáltató szó használatánál első sorban fodrászokra vagy kozmetikusokra utalok.

Dolgozatom Felhasznált technológiák fejezetében bemutatom a projektem elkészítéséhez használt technológiákat. A 3. Követelmények fejezetben bemutatom a megoldás elvárt követelményeit. A 4. Architektúra fejezetben röviden áttekintem az elkészült alkalmazás architektúráját. Az 5. Megvalósítás fejezetben az elvárt funkcionális követelmények alapján mutatom be az alkalmazásomat. A 6. Összefoglaló fejezetben pedig röviden összefoglalom tapasztalataimat és az alkalmazás fejlesztési lehetőségeit.

2 Felhasznált technológiák

2.1 Spring Boot

A Spring framework egy népszerű és rugalmas Java keretrendszer, amely leegyszerűsíti a webes alkalmazások fejlesztését. Célja, hogy megoldást kínáljon a Java nagyvállalati alkalmazásokban gyakran felmerülő problémákra. Fő jellemzője a modularitás, és az Inversion of Control (IoC) és Dependency Injection (DI) tervezési elvek, mely során az objektumok (úgynevezett bean-ek) életciklusa és a függőségek feloldása és injektálása a keretrendszerre van bízva.

A Spring Boot [1] a Spring Framework egyik kiterjesztése, amelynek célja, hogy leegyszerűsítse a Spring alapú alkalmazások fejlesztését és telepítését. Mivel a Spring Framework rendkívül rugalmas és személyre szabható, a konfigurálás gyakran hosszan tartó manuális beállításokat és komplex döntéseket igényel. A Spring Boot ezt a komplexitást egyszerűsíti előre beállított technológiákkal és komponensekkel. Egy új projekt létrehozására használhatjuk a Spring Initializr [2] weboldalt, melynél a felületen kiválaszthatjuk a kezdő projektünk technológiáit és függőségeit, majd a generált projektet letöltve futtathatjuk is az új alkalmazásunkat.

A Spring keretrendszer számos annotációval rendelkezik, amelyek különböző célokra szolgálnak, és megkönnyítik a fejlesztők számára a konfigurálást. A rétegzési annotációk, mint például a @Service, @Repository és @Controller, segítenek a különböző rétegek elkülönítésében. A konfigurációs annotációk lehetővé teszik az alkalmazás konfigurációját XML fájl helyett Kotlin osztályokból. Ezeken kívül lehetőségünk van webes, adatbázis, életciklus, scope és injektálási annotációk használatára is.

```
@Service
class UserService(private val userRepository: UserRepository) {

    fun getUserCount(): Long {
        return userRepository.count() // visszaadja a felhasználók számát
az adatbázisban
    }
}
```

1. kódrészlet Spring Dependency Injection példa

Az 1. kódrészlet egy üzleti rétegben található bean leírására ad pédát. Mivel nem kell foglalkoznunk az objektum életciklusával, így nincsen boilerplate kód, pár sorból létrehozhatunk egy ilyen osztályt. A Spring a konstruktorként megadott típust kikeresi a regisztrált bean-ek közül, példányosítja és létrehozza vele a UserService objektumot. A keretrendszer kikényszeríti a fa jellegű függőségeket, mivel két objektum egymásra függése esetén (akár tranzitívan) egyiket sem tudja létrehozni, hiszen ehhez szüksége van a másik objektumra az injektáláshoz. Ezzel nem alakulnak ki körkörös, kibogozhatatlan függőségek, ezzel átláthatóbbá és könnyebben kezelhetővé téve a kódot.

2.2 Gradle

A Gradle [3] egy futtatás automatizáló eszköz Java, Android és Kotlin alapú projektekhez. Segít a projekthez szükséges függőségek telepítésében, projekt konfigurációban. Rendelkezik beépített futtatható taszkokkal, például build és clean, illetve akár általunk definiált taszkokkal is testre szabhatjuk alkalmazásunk futtatási folyamatait. Ez nem csak fejlesztés során lehet kényelmes, de akár a DevOps feladatokat is megkönnyíthetik a konzolból hívható Gradle taszkok.

A Gradle a deklaratív DSL (Domain Specific Language) nyelv használatával könnyen olvasható és karban tartható build fájlokat eredményez, továbbá egyszerűen integrálható a Kotlin nyelvvel.

2.3 JPA és Hibernate

A Jakarta Persistence (JPA) [4] a korábban Java Persistent API-ként ismert Java szabvány az objektumok és relációs adatbázisok közötti adatleképezéshez (ORM). A Hibernate [5] a JPA egyik legelterjetteb implementációja, amely megvalósítja és kiterjeszti ezt a szabványt. A Spring Data JPA biztosítja ezen technológiák szoros integrációját a Spring Boot keretrendszerbe, ezzel minimalizálva a szükséges konfigurációt.

```
@Entity
@Table(name = "users")
data class User(
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    val id: Long? = null,
    @Column(nullable = false)
    val username: String,

    @Column(nullable = false)
```

```
    val email: String

    @Column(nullable = false)
    val birthDate: LocalDate
)
```

2. kódrészlet Példa entitás

Az adatbázisban lévő táblák sorait közvetlenül Kotlin osztályokként kezelhetjük, melyre a 2. kódrészlet ad példát. Az @Entity annotációval adhatjuk meg, hogy ez az osztály egy adatbázisban lévő táblát reprezentál, és objektumai a táblán belüli entitás példányokat. A fent látható módon megadhatjuk például a tábla nevét, elsődleges kulcsot és annak generálási módját, oszlopokat és ezek tulajdonságait. Emellett lehetőséget ad entitások közötti komplexebb relációk kezelésére, például egy-a-sokhoz (One-to-Many) vagy sok-a-sokhoz (Many-to-Many). A kapcsolatok betöltésének típusát is beállíthatjuk Lazy vagy Eager stratégiára. A Kotlin nyelvi funkciói, például a data class és null biztonság tovább növelik az adatbázis kezelés hatékonyságát.

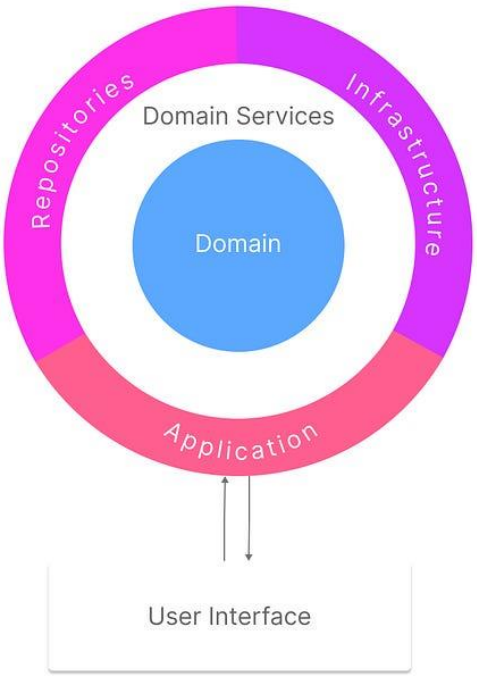
Az entitások kezelésére a JPA egy előre definiált, típusosan paraméterezhető repository interfészt nyújt. Így a leggyakoribb adatbázis műveleteket, mint a mentés, keresés, törlés, frissítés, előre megírt függvényekkel végezhetjük el, mely kevesebb boilerplate kódot eredményez. A beépített funkciókon kívül készíthetünk egyedi lekérdezéseket is. A repository képes a függvény deklarációból lekérdezést generálni, amely során nem szükséges megírnunk az implementációt, vagy az adatbázis lekérdezést, csak a megfelelő függvény elnevezési konvenciót kell használnunk.

```
@Repository
interface UserRepository : JpaRepository<User, Long> {
    fun findByUsername(username: String): Optional<User>
    fun findAllByBirthDateAfter(date: LocalDate): List<User>
}
```

3. kódrészlet Példa repository interfész

A fenti példán (3. kódrészlet) a User entitás kezelésére egy UserRepository interfészt hozunk létre, mely leszármazik a User típusú JpaRepository interfészből. Ezen már elérhetőek a korábban említett műveletek, és két új lekérdezést is megfogalmaztunk függvény elnevezésével és paraméterezésével. A findByUsername függvény segítségével kereshetünk a User példányok között felhasználónév alapján, a findAllByBirthDateAfter függvény pedig azokat találja meg, akik egy bizonyos dátum után születtek.

2.4 Tervezési minták



4. kódrészlet Domain Driven Design [6]

A Domain-Driven Design (DDD) [6] egy tervezési megközelítés, amelynek célja, hogy az üzleti logika köré építsük fel az alkalmazás architektúráját (3. ábra). A DDD alapelvei szerint a szoftver tervezése során a valódi üzleti problémák modelljei (domain-ek) kerülnek előtérbe. Egy általános monolitikus alkalmazáshoz képest szükséges, hogy az üzleti logika független maradjon az adatbázistól, amelyet a függőség inverziójával oldhatunk meg. A Repository minta segítségével egy közös absztrakciós réteget, a egy repository interfészt tehetünk a kettő közé, ezzel megvalósítva az inverziót.

A Domain Driven Design megközelítésben a függőségek inverziója mellett szükségünk van Data Transfer Object-ekre (DTO), melyekkel minimalizálhatjuk a rétegek közötti függőségeket. Rétegenként egyedi adatstruktúrát építhetünk, és megóvhatjuk adatainkat a lehetséges inkonzisztens állapotoktól. A Kotlin nyelvi elemek leegyszerűsítik az entitások és adatátviteli objektumok közti leképezést, ezzel csökkentve a boilerplate kód mennyiségét.

2.5 TypeScript

A JavaScript egy prototípus-alapú programozási nyelv, amelyet elsősorban weboldalak interaktivitásának és dinamikus viselkedésének biztosítására használnak. A TypeScript a JavaScript egy típuskiterjesztése, amely statikus típusosságot vezet be a JavaScript dinamikus típusrendszere fölé. Már a kód írásakor típusellenőrzést végez, így hibamentesebb, átláthatóbb és könnyebben karbantartható kódot eredményez.

2.6 React

A React egy nyílt forráskódú JavaScript könyvtár, amelyet a Meta (korábban Facebook) fejlesztett ki a webes felhasználói felületek építésére. Segítségével könnyen készíthetünk Single-page alkalmazásokat. Egy Single-page alkalmazás egy olyan webes alkalmazás, amelynél a felhasználói interakció során a weboldal dinamikusan újraírja tartalmát, szemben a klasszikus megoldással, melynél új oldalak betöltésére lenne szükség. Ezáltal gyorsabban képes reagálni a felhasználó kéréseire.

A React fő célja, hogy a felhasználói felület elemeit komponensekké bontva megkönnyítse az újrafelhasználhatóságot és az alkalmazások kezelhetőségét. Ezeket a komponenseket deklaratív módon JavaScript vagy a típus kiterjesztett TypeScript használatával tehetjük meg. A komponensek olyan változók, melyek tartalmazzák a működésükhöz szükséges logikát, megjelenítést és állapotot. Komponenst osztályokkal vagy függvényekkel hozhatunk létre, melyeknek visszatérési értékét JSX (JavaScript Syntax Extension) [7] kifejezésekkel adhatjuk meg.

A JSX a JavaScript egy kiterjesztése, mely lehetővé teszi, hogy HTML-hez hasonló formátumban írjunk közvetlenül a JavaScript kódban. A JSX egy XML leírás, mely JavaScript függvényekre fordul, így a kód egyszerűbb és átláthatóbb, mintha függvény hívásokat használnánk. A JSX a HTML elemek mellett emellett lehetővé teszi a komponensek egymásba ágyazását is.

```
function ChildComponent() {
  return <p>Hello from the Child Component!</p>;
}

function ParentComponent() {
  return (
    <div>
      <h1>This is the Parent Component</h1>
      <ChildComponent />
    </div>
  );
```

```
}
```

5. kódrészlet React példa komponens

Ahogy az 5. kódrészlet mutatja, a szülő komponens nem csak szöveget fog megjeleníteni, hanem még bármit, amit a gyerek komponens visszaad. A komponens megjelenítése és logikájának megvalósítása ilyen függvények használatával erősen limitált, mivel nem rendelkeznek belső állapottal. A React Hooks [8] egy 16.8-as verzióban bevezetett funkció, amely lehetővé teszi a funkcionális komponensekben olyan funkciók használatát, mint a state és a lifecycle metódusok, amelyek korábban csak osztály alapú komponensekben voltak elérhetők. A funkcionális komponensek használata népszerűbbé vált, különösen a React Hooks bevezetése óta, mivel egyszerűbb és tisztább szintaxisa van, nem kell törődnünk a this kontextussal, vagy a constructor és lifecycle metódusokkal. Ilyen Hook például a useState vagy a useEffect.

```
function Counter() {
  const [count, setCount] = useState(0);
  useEffect(() => {
    console.log(`The counter has changed. New value: ${count}`);
  }, [count]);

  return (
    <div>
      <button onClick={() => setCount(count + 1)}>Increment</button>
    </div>
  );
}
```

6. kódrészlet UseState és useEffect hook példa

A 6. kódrészlet egy példát mutat a hook használatra. A useState hook a komponens állapotának tárolását és megváltoztatását teszi lehetővé változókban. Egy ilyen változónak megadhatjuk az alapértelmezett értékét, van egy neve, amivel lekérhetjük a jelenlegi értékét, és egy set metódusa, mellyel beállíthatjuk azt. Amennyiben egy állapotérték megváltozik, az oldal újból kirajzolja a komponenst az új állapot értékeivel.

A useEffect hook képes mellékhatásokat rendelni adatokhoz és eseményekhez, konfigurációjától függően automatikusan lefut a benne lévő kód bizonyos események hatására. Ahogy a korábbi példában látható, megadhatjuk, hogy milyen adatok változásánál fusson le a logika, jelen esetben a count változó változásakor. Minden esetben amikor a count megváltozik, a konzolba kiírjuk, hogy mennyi az értéke. Használatakor vigyázunk kell, mivel könnyen végtelen ciklusba juthatunk, ha a

függvényben egy olyan változót változtatunk, amely a függősége is. Az oldal renderelése után mindig lefut egyszer a useEffect belsejében található kód, ez külnösen hasznos, ha adatlekérést szeretnénk végrehajtani, amikor a komponens frissül.

A komponensek és állapotok kezelésén kívül szükségünk lehet egyéb könyvtárakra, melyek kiegészítik a funkcionalitást. Például a React Router [9] segítségével megadhatjuk, hogy melyik URL melyik komponenshez navigáljon, vagy az Axios amivel webes kéréseket küldésében segít.

2.7 Material UI

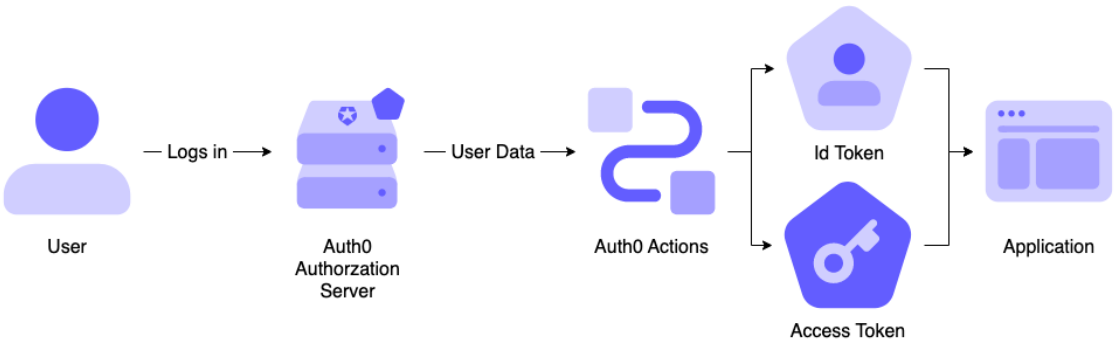
A Material UI egy UI fejlesztői eszköz gyűjtemény, mely egységes és könnyen kezelhető alkotóelemeket nyújt a webes alkalmazások fejlesztéshez. A Material Design stílusirányelvek miatt az elemek hasonló stílusúak, így egyszerűbb az esztétikus weboldalakat létrehozni, emellett az elemek stílusa rugalmasan módosítható, ezzel teret adva az egyedi megoldásoknak. Mivel ez a React technológiához tervezett UI könyvtár, így hasonlóan JSX leírással megadva egyszerű az elemek használata. A Material UI komponensek nagyon hasonlítanak a sima React komponensekhez, emellett jól kidolgozottak, felparaméterezhetőek.

2.8 Axios

Az Axios egy HTTP kliens, mely lehetővé teszi az aszinkron HTTP kéréseket. Támogatja a JavaScript Promise technológiáját, mely megkönnyíti az aszinkron kérések használatát az async és await kulcsszavakkal.

2.9 Auth0

Az Auth0 egy OpenID protokoll alapú autentikációt biztosít. Az OpenID egy autorizációs szerver segítségével azonosítja a felhasználót, és adataiknak lekérését. Meg szerettem volna ismerni egy biztonságos és megbízható felhasználó kezelési módot, melynél a jelszó és regisztráció kezelés delegált, és lehetővé tesz egyéb bejelentkezési módszereket, például Google vagy Facebook fiókokkal.



1. ábra Auth0 bejelentkezés

Az Auth0 esetén egy alkalmazás szervert kell létre hoznunk a webes felületen, amely kezeli a beérkezett autentikációs kéréseket. Itt lehetőség van a már regisztrált felhasználók kezelésére és aktivitásuk megtekintésére. Amikor egy felhasználó be szeretne jelentkezni (1. ábra), akkor a frontend elnavigál az Auth0 által nyújtott bejelentkező felületre. Itt a felhasználó azonosítja magát, és a megadott címre visszairányítja az Auth0. Ekkor a frontendünkön el tudjuk érni a felhasználó adatait (ID Token) és a használatra jogosító hozzáférési tokent (Access token).

Amennyiben új módszerrel jelentkezik be vagy regisztrál a felhasználó, kap egy új – az eddigiektől eltérő - azonosítót. Ezért fontos, hogyha például össze szeretnénk kapcsolni egy felhasználó Facebook és Google fiókját, akkor tudnunk kell, hogy ezek az azonosítók ugyanarra a személyre vonatkoznak. Erre például a jó megoldás lehet, hogyha már regisztráltak ezzel az e-mail címmel felhasználót, akkor megadjuk a lehetőséget, hogy összekapcsolja a két fiókot.

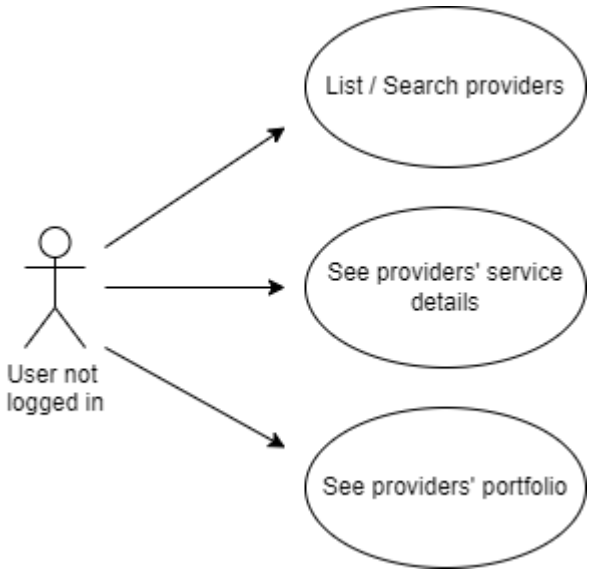
3 Követelmények

A diplomaterv feladatom egy időpontfoglalásra alkalmas webes alkalmazás elkészítése volt, melynek követelményeit a következőkben részletezem.

Az időpont foglaláshoz szükséges, hogy legyen olyan felhasználó, aki szolgáltatást és ehhez elérhető időpontokat tud megadni. Kell egy olyan felhasználó is, aki foglalni tud ezekre az időpontokra, és az alkalmazás azt is támogatja, hogy bejelentkezés nélkül is meg lehessen tekinteni ezeket a foglalható szolgáltatásokat. Így az alkalmazás felhasználóit három csoportra bontottam. Az első csoport a nem bejelentkezett felhasználók, akik pár limitált funkciót használhatnak. Mivel a szolgáltatóknak külön felület kell, ahol felvihetik adataikat a szolgáltatásaikról, ezért ezt a két felhasználói típust vettem még fel: általános felhasználó és a szolgáltató felhasználó.

Eleinte teljesen elkülönülő szerepkörökként akartam létrehozni a vevő és szolgáltató felhasználókat, azonban ekkor a szolgáltató szerepű felhasználóknak nem lenne lehetősége időpontot foglalni más szolgáltatókhoz. Készíteniük kellene egy különálló profilt csak a foglalásra. A szolgáltatói és vevői profil közti váltogatás erősen negatívan hatna a felhasználói élményre. Végül úgy döntöttem, hogy a szolgáltatók is használhatnak minden funkciót, mint egy általános felhasználó, és emellett képesek egyéb extra funkciók igénybevételére is.

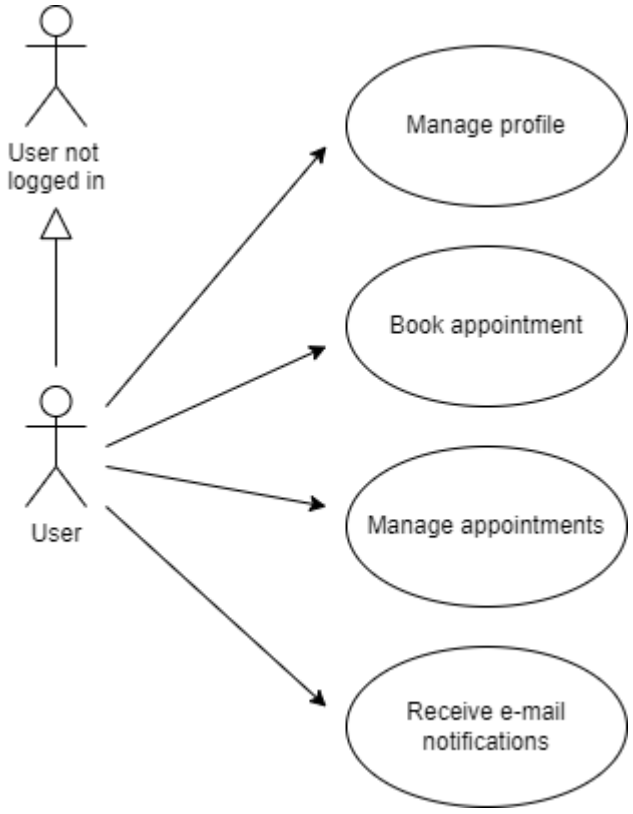
3.1 Böngészés



2. ábra Be nem jelentkezett felhasználó

A be nem jelentkezett felhasználók megtekinthetik a regisztrált szolgáltatókat. Lehetőségük van kilistázni őket és megtekinteni adataikat. Ebben a listában tudnak név alapján keresni, illetve szűrni az eredményeket szolgáltatások alapján. Emellett megnézhetik az egyes szolgáltatók által nyújtott szolgáltatások részleteit. Ilyen részlet lehet például, hogy meddig tart vagy milyen árban van.

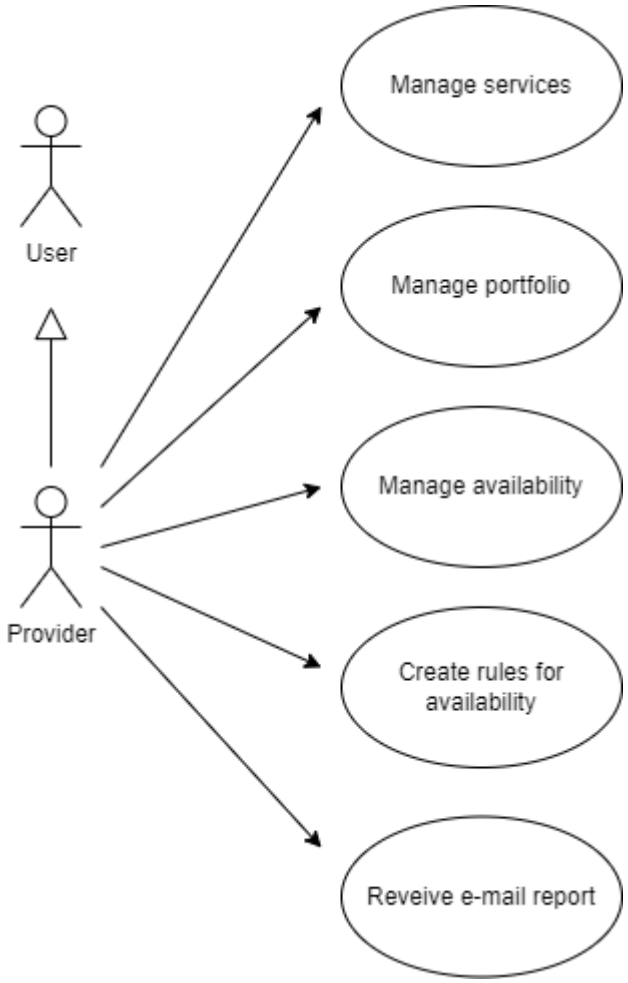
3.2 Általános felhasználó



3. ábra Általános felhasználó

Bejelentkezés után hozzáférhetünk az általános felhasználóknak elérhető funkciókhoz. Ez a szerep a be nem jelentkezett felhasználóhoz hasonlóan képes kilistázni, keresni a szolgáltatók között. Megtekintheti egy szolgáltató által nyújtott szolgáltatások részleteit és az ezekhez kapcsolódó portfólióját. Miután talált egy szimpatikus szolgáltatót, elindíthatja az időpont foglalás folyamatát. Ehhez ki kell választania a szolgáltatást és az időpontot is. Az időpont foglaláskor és az előtti este kap egy emlékeztető e-mailt a foglalás adataival. Emellett képes megnézni a profilját, ezen módosításokat végezni és menedzselni a már lefoglalt időpontjait.

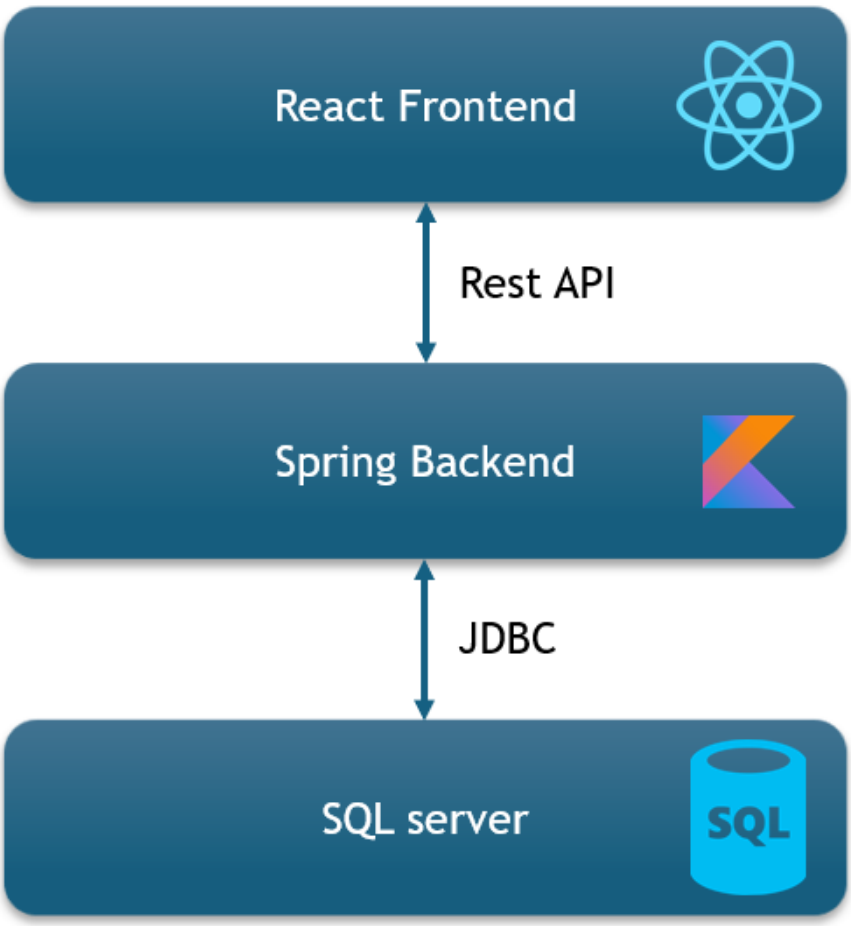
3.3 Szolgáltató felhasználó



4. ábra Szolgáltató felhasználó

A szolgáltatók számára elérhető az összes általános felhasználói funkció, továbbá képesek felvenni és menedzselni általuk nyújtott szolgáltatásokat és ezeknek portfólióit. Meg tudják adni, hogy mikor elérhetők időpont foglalásra. Ezt nem csak egyesével tudják felvinni, hanem képesek ismétlődő eseményeket készíteni szabályokkal. Például a következő hány hónapra legyenek kiírva ezek az események, mettől meddig tartsanak. A rendes felhasználókhoz képest a profiljuk szerkesztésekor megadhatják és módosíthatják üzleti profiljuk adatait, például a szolgáltatás helyét, és a saját foglalásaik mellett a hozzájuk lefoglalt időpontokat is kezelhetik.

4 Architektúra

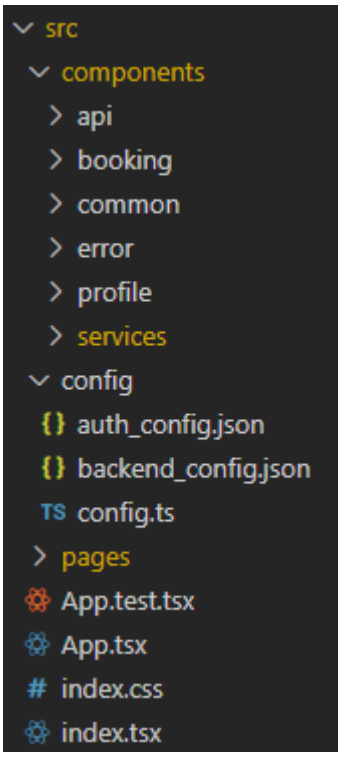


5. ábra Architektúra

Webalkalmazásom három különböző részből épül fel. Az 5. ábra tetején látható a React alapú frontend szerverem, mely biztosítja az oldal megjelenítését, a felhasználók autentikációját és a kérések delegálását a Spring Boot backend szerverem felé. Ez a kommunikáció egy REST API interfészen keresztül megy. A backendre beérkező kérések átmennek az üzleti logikán, és a szükséges adatokat a JPA segítségével elkéri az SQL adatbázis szervertől.

4.1 Frontend architektúra

Az alkalmazásom frontendjét React-ban valósítottam meg. Azért választottam ezt a technológiát, mivel reaktív, nagyon népszerű és elterjedt, emellett szakmailag sokat fejlődhetek megismerésével és elsajátításával.



6. ábra Frontend mappastruktúra

A fejlesztés során igyekeztem újra felhasználható, jól paraméterezhető komponenseket készíteni, hogy ne kelljen ugyanazt a feladatot újból megoldani. Ehhez úgy strukturáltam a forráskódot, hogy a hasonló elemek egymás mellé kerüljenek. Az ábrán látható mappa szerkezetet alakítottam ki a fejlesztéshez. Az index.tsx a frontend alkalmazásom belépési pontja. Ebben a komponensben adom hozzá az Auth0 konfigurációt a config mappában található fájlok segítségével. Az oldalakat a pages mappában tárolom, és az App.tsx írja le, hogy milyen útvonal esetén melyik oldalt töltse be. Ezt a React Router könyvtár segítségével tudtam megvalósítani.

A components mappában találhatóak a közös és az oldal specifikus komponenseim. Az ezen belüli api mappában tárolom a backend API modelleket, és a backend API hívásához szükséges függvényeket, melyet az Axios kliens segítségével tudok megtenni.

```
const {
  user,
  isLoading,
  isAuthenticated,
  loginWithRedirect,
  getAccessTokenSilently,
  logout,
} = useAuth0();
```

7. kódrészlet Auth0 hook

Az autentikációra az Auth0 szolgáltatót használom, melyet a weboldalukon felkonfiguráltam a frontend és backend alkalmazásomhoz. A 7. kódrészlet mutatja be a React alkalmazásoknál használt Auth0 hook használatát. Ezt az auth0-react könyvtárral vehetjük igénybe. Megtekinthetjük vele a bejelentkezett felhasználó adatait, elkérhetjük a backend hívásokhoz szükséges access tokent és felhasználó kezelési műveleteket hajthatunk végre vele.

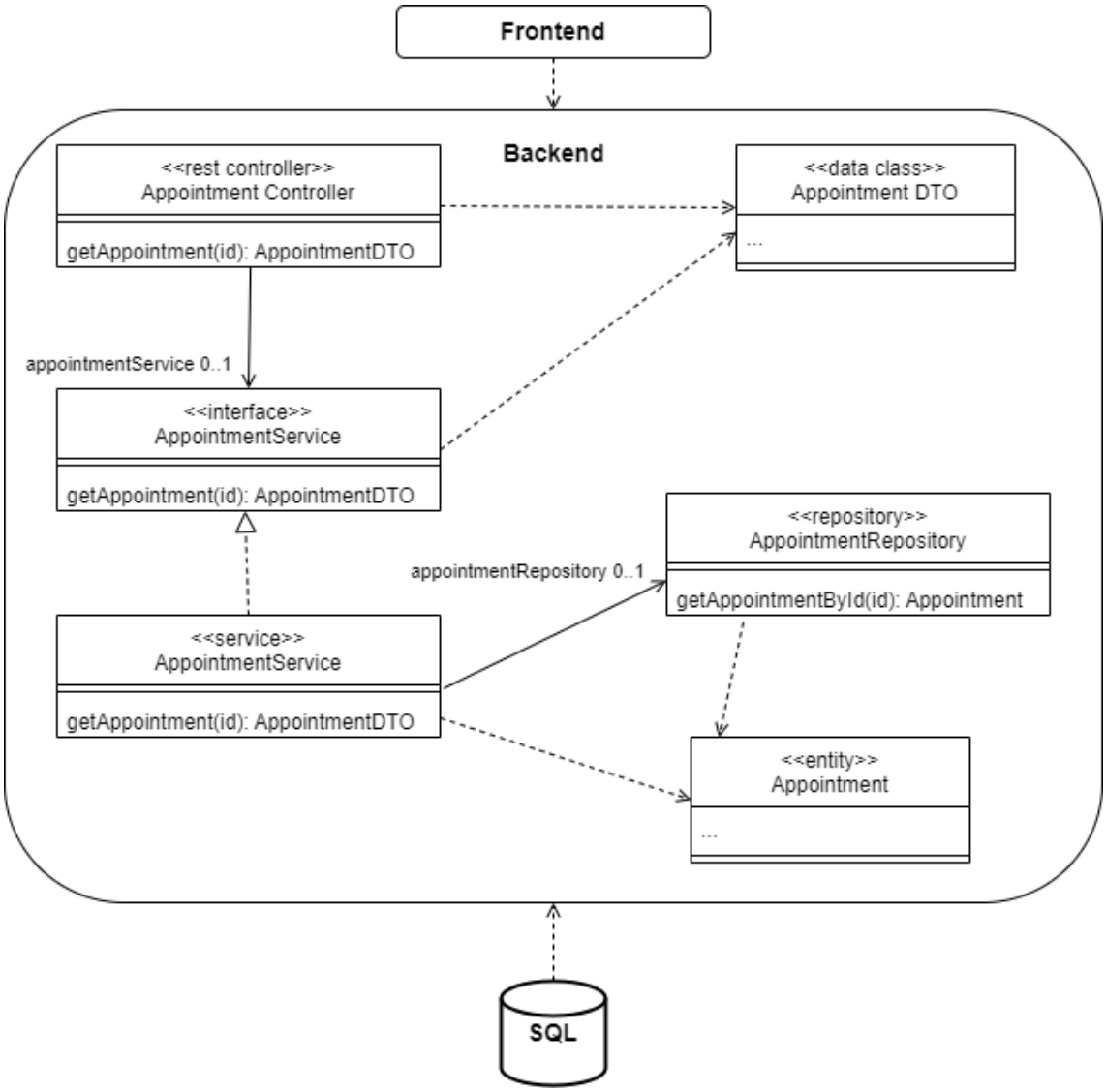
Mivel User Interface tervezéssel még nem volt sok tapasztalatom, ezért fontosnak tartottam egy olyan könyvtár használatát, amellyel könnyen tudok egységes és stílusos komponenseket létrehozni. A Material UI egy ennek megfelelő könyvtár volt, melynek dokumentálhatósága és népszerűsége könnyebbé tette a fejlesztést.

4.2 Backend architektúra

A backend alkalmazásom egy kotlin alapú Spring Boot alkalmazás, mely a Gradle build automatizáló eszközt használja. A Spring keretrendszer népszerűsége, széles körű funkciói és a Spring Boot adta egyszerű konfigurálhatóság miatt döntöttem a technológia mellett. A kotlin a java programozási nyelvhez hasonlóan JVM (Java Virtual Machine) alapú, azonban modernebb és használatakor kevesebb programozási overhead. Fejlett nyelvi funkcióival hatékonyabban megy a fejlesztés, rövidebb és olvashatóbb kódot írhatunk és a nyelv alapértelmezetten megakadályozza a null pointer hibákat, így biztonságosabbá és kiszámíthatóbbá téve a programot.

A Spring projektek esetén a Gradle egy népszerű alternatívája a Maven, amely XML alapú konfigurációs fájlokat használ. A Gradle build fájlok olvashatósága ehhez képest jobb, és a Gradle sokkal rugalmasabb újabb build feladatok írására, amelyek a jövőben például DevOps feladatoknál jól tudnak jönni. A Maven esetén is lehet build feladatokat írni, azonban ez limitált és gyakran plugin használat szükséges hozzá. Ezek a szempontok miatt végül a Gradle mellett döntöttem.

Alkalmazásomat a korábban ismertetett Domain Driven Design alapján terveztem meg, így az üzleti logika által használt domaint helyeztem a középpontba. Ehhez szükség volt az üzleti és adatbázis rétegek közti függőség megfordítására, amelyet a korábban bemutatott Spring Data JPA segítségével a Repository tervezési minta szerint valósítottam meg.



7. ábra Backend architektúra vizualizáció

Az adatbázis táblák a backenden definiált @Entity annotációval rendelkező entitásoktól függenek, az adatbázis műveleteket pedig a @Repository annotációs előre definiált JPA interfészekkel végezhetőek. Az entitások a bounded context elv szerint elkülönülnek az alkalmazás többi részéről. Például a Rest API kontrollerek más adatszerkezetű DTO objektumokat küldenek és fogadnak és nem az üzleti entitásokat használják erre.

Erre mutat egy példát a 7. ábra. A képen látható Appointment Controller fogadja a kéréseket, és delegálja a Spring keretrendszer által injektált AppointmentService osztálynak. Az AppointmentService lekérdezi az AppointmentRepository segítségével a keresett Appointment entitást, ezt átalakítja a DTO reprezentációra, majd visszaküldi az adatot a kontrollernek. Mivel a kotlin data class nyelvi eleme egyszerűvé teszi az adatok

transzformációját, ezért nem használtam külön technológiát erre, a service osztályok maguk gondoskodnak erről.

Az entitásokat lekérdező REST végpontok és az őket kezelő üzleti logikák gyakran saját osztályokban, egymástól elkülönült utat járnak be. Így a service osztályok kevés függőséggel és decentralizált kiszolgálással tudnak válaszolni a frontend kéréseire.

```
@Configuration
@EnableWebSecurity
class SecurityConfig {
    @Bean
    fun filterChain(http: HttpSecurity): SecurityFilterChain {
        http {
            authorizeHttpRequests {
                authorize(GET, "/**", permitAll)
                authorize(anyRequest, authenticated)
            }
            cors {  }
            oauth2ResourceServer {
                jwt {  }
            }
        }
        return http.build()
    }
}
```

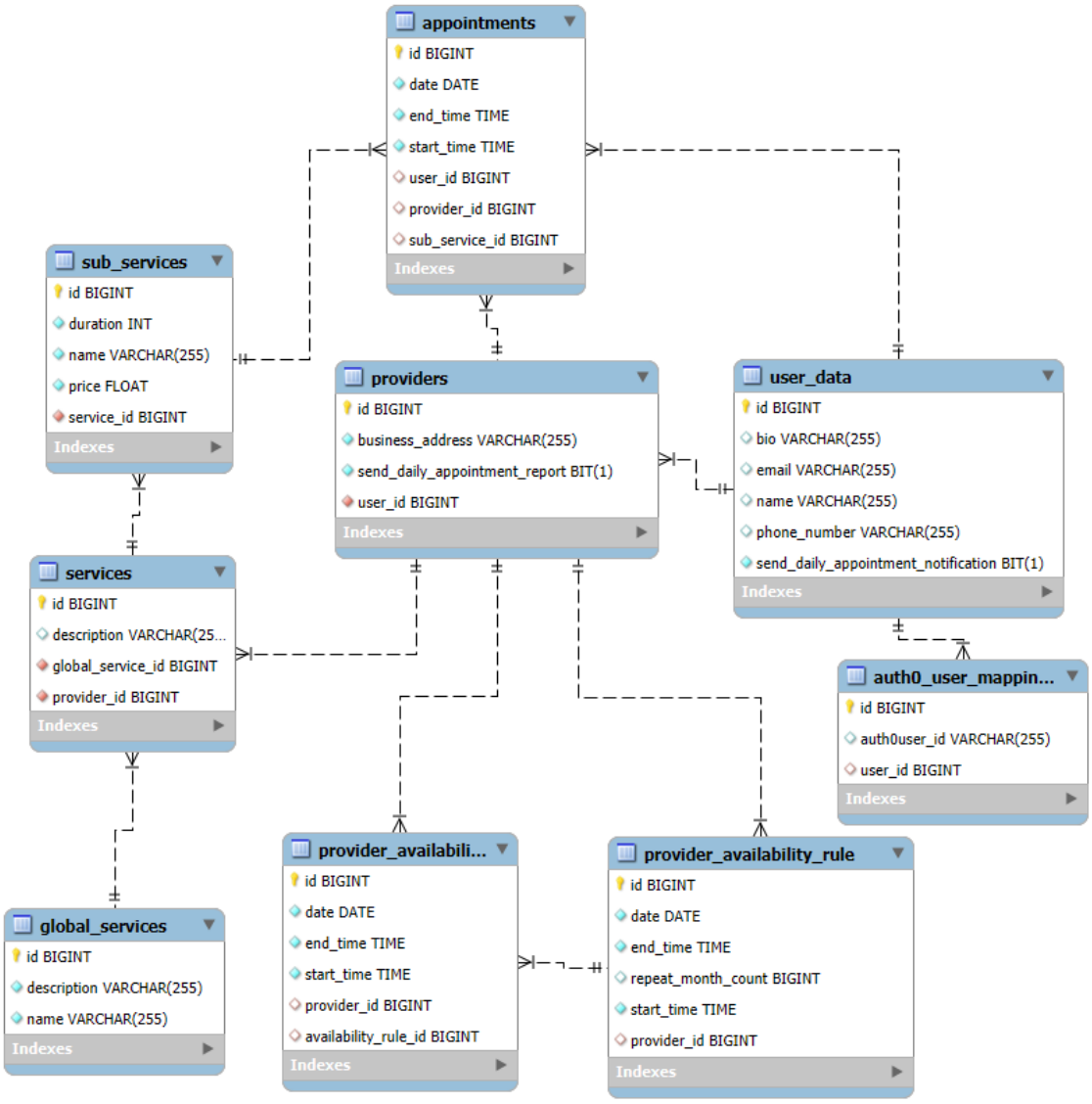
8. kódrészlet Auth0 security konfiguráció

A fenti kódrészletben látható az alkalmazás security konfigurációja. A GET kérések kiszolgálására nincs szükség authentikációra, azonban a módosító műveletekhez már szükséges a bejelentkezés. Az auth0 azonosítás használatához még szükség volt az audience és issuer beállítására az application.properties fájlban. A bejelentkezett felhasználó auth0 adatait az access tokenen keresztül érhetjük el, melyet a SecurityContextHolder Spring osztálytól kérhetünk el.

4.3 Adatbázis szerkezete

A felhasználók, foglalások és szolgáltatók adatainak tárolására szükségem volt egy adatbázisra. Eleinte a H2 relációs adatbázist használtam, mivel a kiinduló Spring Boot projekt már konfigurálva volt ennek használatára, ezzel felgyorsítva a projekt kezdetleges fejlesztését. Ez egy könnyen használható memória adatbázis, azonban ezt főképp fejlesztő és tesztelő környezetekben használják. Ki akartam próbálni egy olyan adatbázis konfigurációt, melyet az éles környezetben használnak. Mivel erősen strukturált adatokat szerettem volna tárolni, ezért a Spring keretrendszerrel gyakran használt relációs adatbázisokat hasonlítottam össze.

A MySQL adatbázis mellett döntöttem, mivel ez platform független, kiemelkedően hatékony az olvasási műveletek esetén és egy széles körben használt technológia. Emellett nyílt forráskódú és egyszerűen konfigurálható, ami kifejezetten alkalmassá teszi egy kis vagy közepes méretű webalkalmazás kiszolgálására. Az adatbázis váltáshoz egy függőséget kellett felvennem, és a JDBC kapcsolat paramétereit kellett konfigurálnom. Miután ezzel végeztem, néhány módosítást kellett végeznem a backend által definiált entitásokban és ismét gördülékenyen működött az összes JPA repository használata, így ez egy tényleg könnyen konfigurálható alternatíva volt.



8. ábra Adatbázis szerkezete

Az adatbázis szerkezetét annotált osztályokból generáltam a Spring Data JPA segítségével. Az osztályokból táblák, az attribútumokból oszlopok keletkeztek. A végső

adatstruktúrát a 8. ábra vizualizálja. Az alábbiakban bemutatom az adatbázis fő és azok kapcsolatát.

- **user_data**: Ez a tábla tárolja a rendszer felhasználóinak alapvető adatait, mint például a nevüket, e-mail címüket és telefon számukat.
- **auth0_user_mapping**: A felhasználók regisztrálásakor létrejött azonosítókat fordítja le az adatbázisunkban használt **user_data** azonosítókra. Ez lehetőséget ad a felhasználók különböző bejelentkezési módú fiókjainak összekötésére.
- **providers**: A szolgáltatók szolgáltatással kapcsolatos adatait tárolja. Ez jelenleg a szolgáltatás címe, és egy preferencia opció, hogy szeretne-e e-mail értesítést kapni. Ezen kívül tárolja, hogy melyik felhasználóra vonatkoznak ezek a szolgáltatói adatok.
- **provider_availability_rule**: A foglalható idősávok ismétlődő szabályait tartalmazza az egyes szolgáltatókhoz. Ehhez kapcsolódó adat, például hogy hány következő hónapra írja ki az idősávokat, ezek mettől meddig legyenek.
- **provider_availability**: A szolgáltatókhoz tárolja azokat az idősávokat, amikben lehet hozzájuk időpontot foglalni. Ehhez szükség van a dátumra, az idősáv kezdetére és végére. Egy **provider_availability** sor lehet egy visszatérő esemény része, vagy egy önálló. Ezt onnan tudhatjuk meg, hogy van-e kulcsa a **provider_availability_rule** táblára, vagy nem. Ez a tulajdonság különösen az esemény módosításnál vagy törlésnél lényeges.
- **global_services**: A szolgáltatások fő kategóriái, amelyeket a szolgáltatók nem módosíthatnak, csak választhatnak közülük. Például fodrászat vagy kozmetika. Erre azért volt szükség, hogy részben egységesítse a szolgáltatókat. Így amikor szolgáltatást keres a felhasználó, ne jelenjen meg több máshogy fogalmazott szolgáltatás, amelyek ugyanarra a fő szolgáltatásra vonatkoznak, például fodrászat és fodrász módon.
- **services**: A szolgáltató általtal nyújtott fő kategóriákat tárolja, melyekre készíthetnek egyéni leírást.

- **sub_services**: A szolgáltatók által nyújtott fő szolgáltatáson belüli alszolgáltatások adatait tároló tábla. Ezek teljesen személyre szabhatók. Tartalmazzák az alszolgáltatás nevét, árát és idejét percben.

5 Megvalósítás

Mindenhova kép a végleges feature-ökről

5.1 Profil

A profil megtekintéséhez először a bejelentkezett felhasználónak a jobb felül található a profil ikonra kell kattintania. A legördülő menüből kiválasztva a profil menüpontot kiválasztva tud navigálni saját profil oldalára.

5.2 Szolgáltatók listája

5.3 Időpont foglalás

insert picture here

Itt látha

Időpont foglalást a bejelentkezett felhasználó képes kezdeményezni. Nem csak a React frontendemben van korlátozva ez az út, de maga a gomb sem jelenik meg egy nem bejelentkezett felhasználó számára.

5.4 Foglalt időpontok kezelése

5.4.1 Általános felhasználó

5.4.2 Szolgáltató

5.5 Szolgáltatások kezelése

5.6 Foglalható időpontok kezelése

6 Összefoglaló

7 Irodalomjegyzék

[1] „Spring Boot,” VMware Tanzu, [Online]. Available: https://spring.io/projects/spring-boot. [Hozzáférés dátuma: 20 10 2024].

[2] „Spring Initializr,” Broadcom Inc., [Online]. Available: https://start.spring.io/. [Hozzáférés dátuma: 20 10 2024].

[3] „Gradle Build Tool,” Gradle Inc., [Online]. Available: https://gradle.org/. [Hozzáférés dátuma: 20 10 2024].

[4] „Jakarta Persistence (JPA),” JetBrains s.r.o., [Online]. Available: https://www.jetbrains.com/help/idea/jakarta-persistence-jpa.html. [Hozzáférés dátuma: 22 10 2024].

[5] „Hibernate ORM,” Red Hat Inc., [Online]. Available: https://hibernate.org/orm/. [Hozzáférés dátuma: 20 10 2024].

[6] „Domain-Driven Design (DDD): A Guide to Building Scalable, High-Performance Systems,” Medium Corporation, [Online]. Available: https://romanglushach.medium.com/domain-driven-design-ddd-a-guide-to-building-scalable-high-performance-systems-5314a7fe053c. [Hozzáférés dátuma: 20 10 2024].

[7] „Writing Markup with JSX,” Meta Open Source, [Online]. Available: https://react.dev/learn/writing-markup-with-jsx. [Hozzáférés dátuma: 20 10 2024].

[8] „React,” Meta Open Source, [Online]. Available: https://react.dev/reference/react/hooks. [Hozzáférés dátuma: 20 10 2024].

[9] „React Router,” Remix, [Online]. Available: https://reactrouter.com/en/main. [Hozzáférés dátuma: 20 10 2024].

Függelék