



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

Összefoglaló	5
Abstract.....	6
1 Bevezetés	7
2 Felhasznált technológiák	9
2.1 Spring Boot	9
2.2 Gradle.....	10
2.3 JPA és Hibernate.....	10
2.4 Tervezési minták.....	11
2.5 TypeScript.....	12
2.6 React	12
2.7 Material UI.....	13
2.8 Axios	14
2.9 Auth0	14
3 Követelmények.....	18
3.1 Böngészés	19
3.2 Általános felhasználó	20
3.3 Szolgáltató felhasználó	21
4 Architektúra	22
4.1 Frontend architektúra	22
4.2 Backend architektúra	24
4.3 Adatbázis szerkezete.....	27
5 Megvalósítás	31
5.1 Profil	31
5.2 Szolgáltatók listája.....	32
5.3 Időpont foglalás	32
5.4 Foglalt időpontok kezelése	35
5.4.1 Általános felhasználó.....	35
5.4.2 Szolgáltató	35
5.5 Szolgáltatások kezelése.....	35
5.6 Foglalható időpontok kezelése.....	39
6 Összefoglaló	46

7 Irodalomjegyzék.....	47
Függelék.....	48

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 fitness 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 hasonlóak, 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 foglalmazni 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 MySQL 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 2. 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 példát. 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

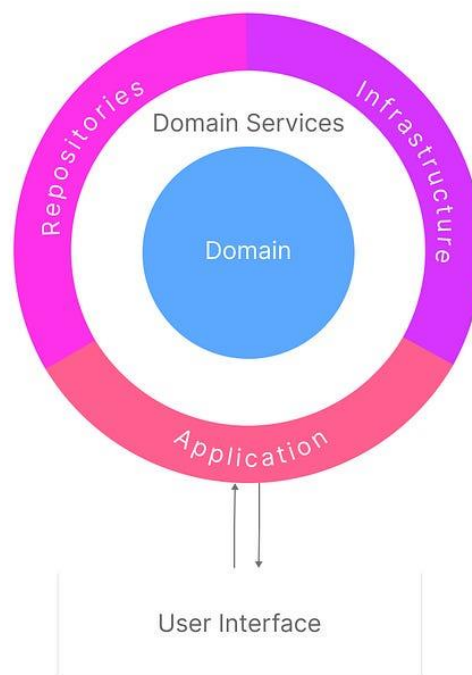
A Jakarta Persistence (JPA) [4] a korábban Java Persistent API-ként ismert Java szabvány az objektumok és relációs adatbázisok közötti adatleképezéshez (ORM). A Hibernate [5] a JPA egyik legelterjedtebb implementációja, amely megvalósítja és kiterjeszti ezt a szabványt. A Spring Data JPA biztosítja ezen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

Az adatbázisban lévő táblák sorait közvetlenül Kotlin osztályokként kezelhetjük. Az @Entity annotációval adhatjuk meg, hogy ez az osztály egy adatbázisban lévő táblát reprezentál, és objektumai a táblán belüli entitás példányokat. 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 megírunk az implementációt, vagy az adatbázis lekérdezést, csak a megfelelő függvény elnevezési konvenciót kell használnunk.

2.4 Tervezési minták



2. 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 (4. 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.

Ez azt jelenti, hogy egy szülő komponens nem csak a benne lévő JSX és HTML tartalmat fogja megjeleníteni, hanem még bármit, amit a gyerek komponense visszaad. A komponens megjelenítése és logikájának megvalósítása 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.

A useState hook a komponens állapotának tárolását és megváltoztatását teszi lehetővé változóban. 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 komponens új állapot értékeivel.

A useEffect hook képes mellékhatásokat rendelni adatokhoz és eseményekhez, konfigurációjától függően automatikusan lefut a benne lévő kód bizonyos események hatására. 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 különösen 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

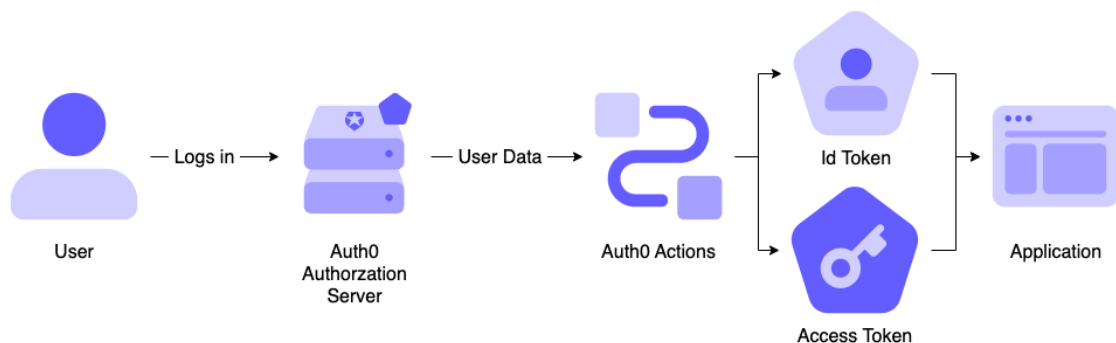
weboldalakat létrehozni, emellett az elemek stílusa rugalmasan módosítható, ezzel teret adva az egyedi megoldásoknak. Mivel ez a React technológiához tervezett UI könyvtár, így hasonlóan JSX leírással megadva egyszerű az elemek használata. A Material UI komponensek nagyon hasonlítanak a sima React komponensekhez, emellett jól kidolgozottak, felparaméterezhetők.

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 adatainak 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 szerveret 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).

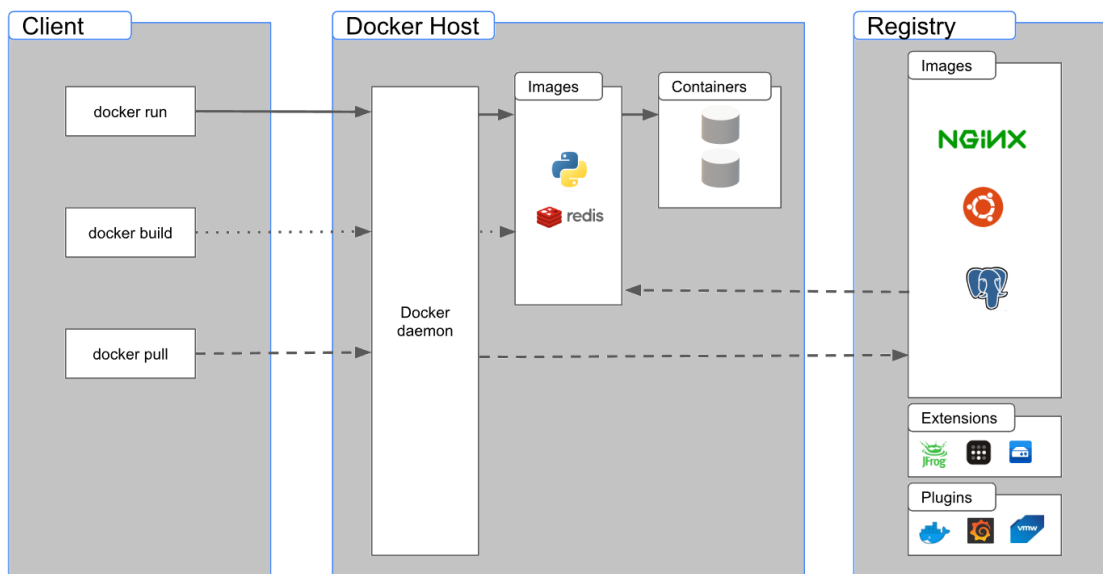
Amennyiben új módszerrel jelentkezik be vagy regisztrál a felhasználó, kap egy új – az eddigiektől eltérő - azonosítót. Ezért fontos, hogyha például össze szeretnénk kapcsolni egy felhasználó Facebook és Google fiókját, akkor tudnunk kell, hogy ezek az azonosítók ugyanarra a személyre vonatkoznak. Erre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

2.10 Docker

A Docker egy nyílt forráskódú konténerizációs platform, amely lehetővé teszi az alkalmazások és azok futtatási környezetének csomagolását könnyen hordozható egységekbe, úgynevezett konténerekbe. A konténerek izolált környezetben futnak, amely biztosítja az alkalmazások megbízható működését különböző környezetekben, legyen szó fejlesztői gépekről, tesztserverekről vagy éles rendszerekről.

A hagyományos virtuális gépekkel ellenben a konténerek nem futtatnak saját operációs rendszert, hanem a host gép kernelét használják. Ezzel a konténerek kevesebb erőforrást igényelnek és gyorsabban indulnak. Emellett a konténerek könnyen másolhatóak és telepíthetőek különböző környezetekbe, így gyakran használják skálázható szoftverek építésére.

Egy konténer létrehozásához szükségünk van egy Docker image nevű sablonra. A Docker image tartalmazza, hogy mik az alkalmazáshoz futtatásához szükséges függőségek, könyvtárak és bináris fájlok. Ez garantálja, hogy akármilyen környezetben futtatjuk programunkat, a felhasznált függőségei nem változnak, ezzel platform függetlenné téve azt. Egy ilyen image létrehozásához szükségünk van egy Dockerfile nevű konfigurációs fájlra, melyben definiálhatjuk a szükséges függőségeket, kijelölt portokat és alkalmazásunk belépési pontját.



2. ábra Docker architektúra [10]

A Docker platform egy komponense a Docker Engine a konténerek kezelését az teszi lehetővé az ábrán látható Docker Client segítségével. Lehetőségünk van egy command-line interfészen (CLI) vagy egy grafikus felhasználói felületen (GUI) interaktálni ezzel a komponenssel. A CLI segítségével a build parancs hatására a Docker Engine elkészíti az image csomagot a Dockerfile alapján, és a run parancs hatására elindítja a konténert az alkalmazással, és annak teljes környezetével. Ezen kívül lehetőségünk van törölni, leállítani és újraindítani konténereket. Egy image példányból akár több konténert is indíthatunk, ezzel skálázhatóbbá téve alkalmazásunkat.

Gyakran nem szeretnénk az összes jellemzőjét egyesével definiálni a konténerünknek. Például egy backend futtatására elég lehet tudnunk, hogy a backend kódját a megfelelő verzióban képes legyen lefordítani a környezet. A Docker Hub egy konténer regisztrációs szolgáltatás, ahonnan előre definiált image szoftvercsomagokat használhatunk, ezzel felgyorsítva a konfigurációt. A Docker Engine a konténer futtatásakor automatikusan egy pull művelettel letölti, és lokálisan eltárolja a futtatni kívánt image csomagot, ha korábban még nem töltöttük le.

Egy modern alkalmazás esetén gyakran több konténerre van szükségünk, melyeknek kezelését nem akarjuk egyenként végezni. Ilyen például egy webalkalmazás frontend, backend és adatbázis szerverekkel. Ehhez gyakran orkesztrációt használnak, mellyel biztosíthatjuk a terheléelosztást, a konténerek közötti kommunikációt és a skálázást. Ilyen orkesztrációs technológiát biztosít a Kubernetes. Kisebb projekteknél

elegendő lehet a Docker Compose használata, melynél egy docker-compose.yml fájlban definiálhatjuk a konténerek közötti kapcsolatot, indítási sorrendet és egy egyszerű skálázást. Ugyan a többi orkesztrációs technológiához képest a Docker Compose funkcionalitása limitált, kisebb projekteknel elegendő lehet ennek használata az egyszerűsége és gyorsan konfigurálhatóságra való tekintettel.

3 Követelmények

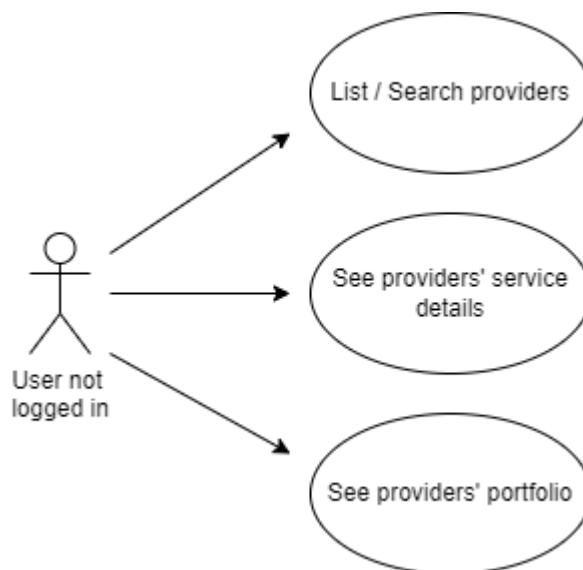
A diplomaterv feladatom egy időpontfoglalásra alkalmas webes alkalmazás elkészítése volt, melynek követelményeit a következőkben részletezem.

TODO: Kép

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.

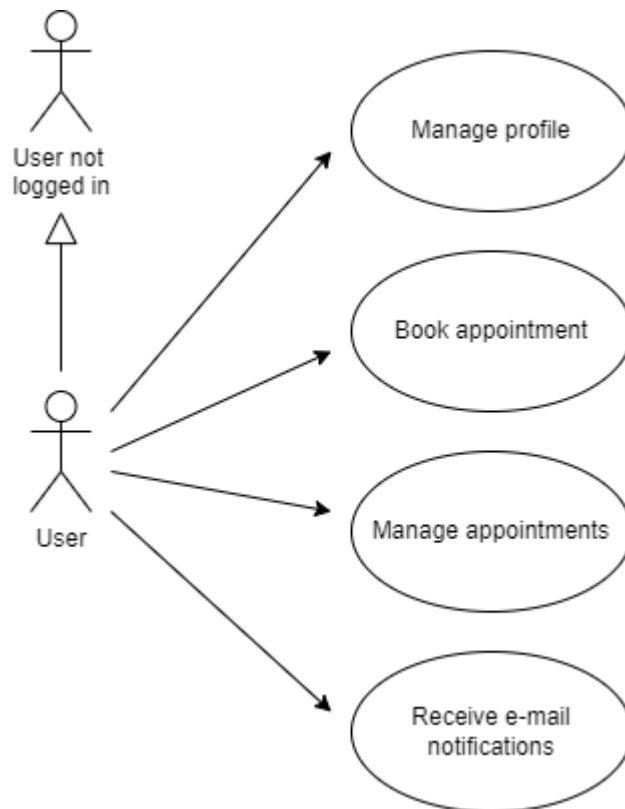
3.1 Böngészés



3. 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.

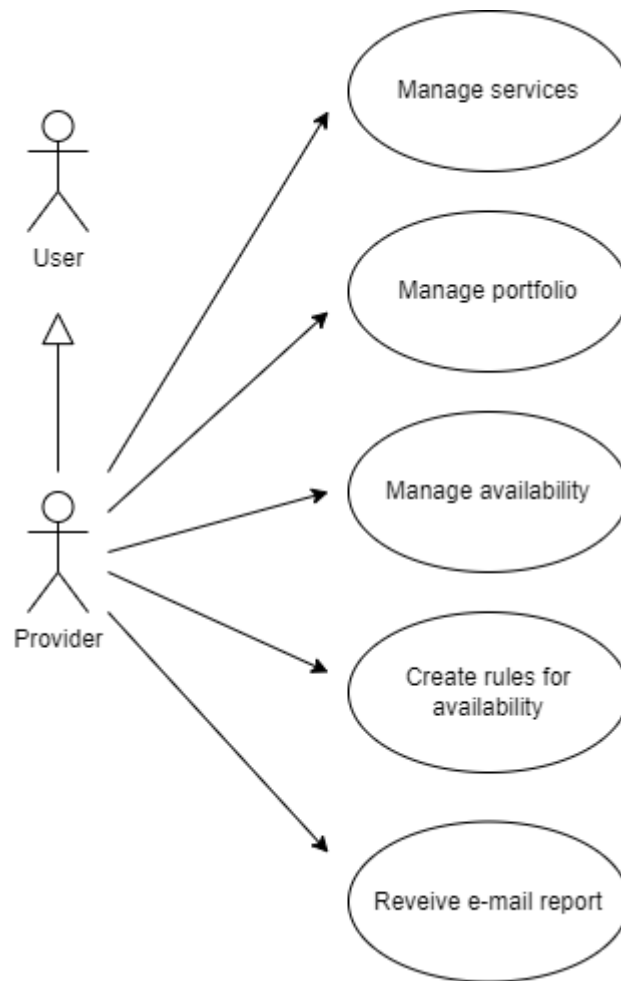
3.2 Általános felhasználó



4. 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.

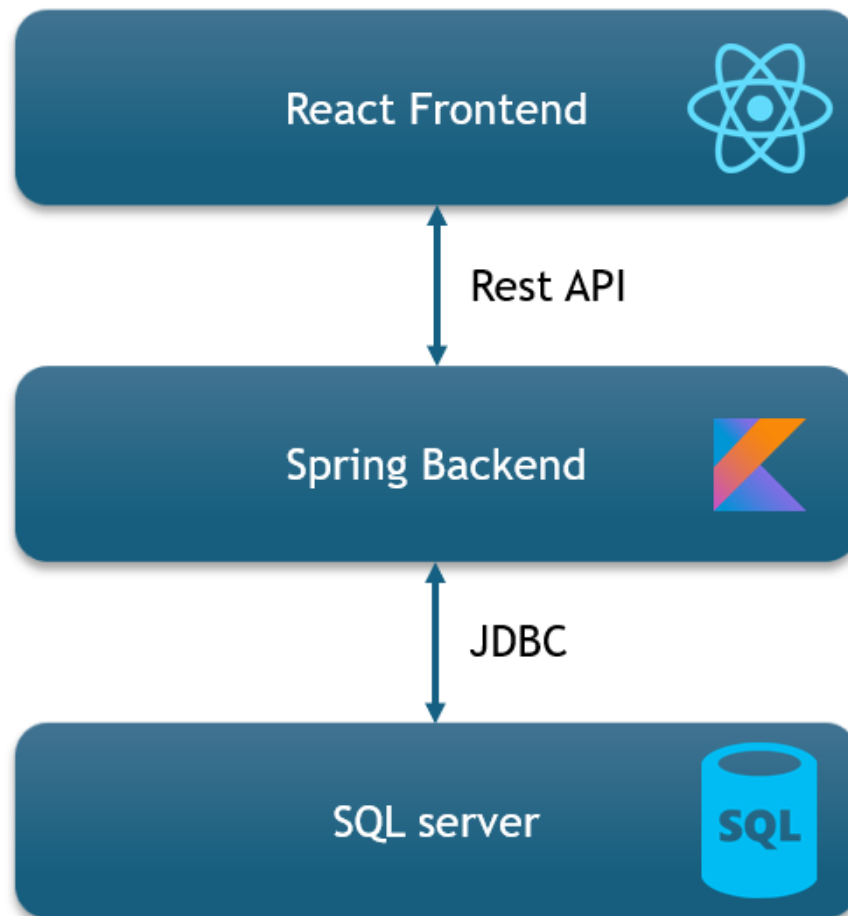
3.3 Szolgáltató felhasználó



5. 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

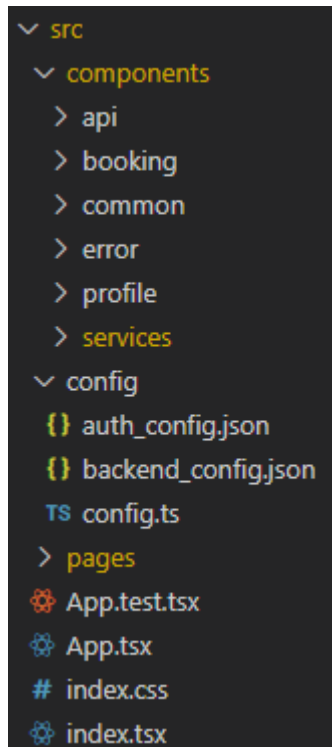


6. ábra Architektúra

Webalkalmazásom három különböző részből épül fel. Az 6. 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 fejlődhetnek megismerésével és elsajátításával.



7. 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 töltsse 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();
```

3. kódrészlet Auth0 hook

Az autentikációra az Auth0 szolgáltatót használok, melyet a weboldalukon felkonfiguráltam a frontend és backend alkalmazásomhoz. A 3. 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 tokenet és felhasználó kezelési műveleteket hajthatunk végre vele.

Mivel a felület lehetőséget ad adatok módosítására és új adatok felvételére, ezért ezek validációja is egy fontos feladat. Amikor egy mező nincs kitöltve, az input validáció megakadályozza a felhasználót, hogy elmentse az adatokat. A speciális formátumú adatoknál, például e-mail cím és telefonszám validáció esetén regex segítségével is ellenőrizzük a megadott adatot..

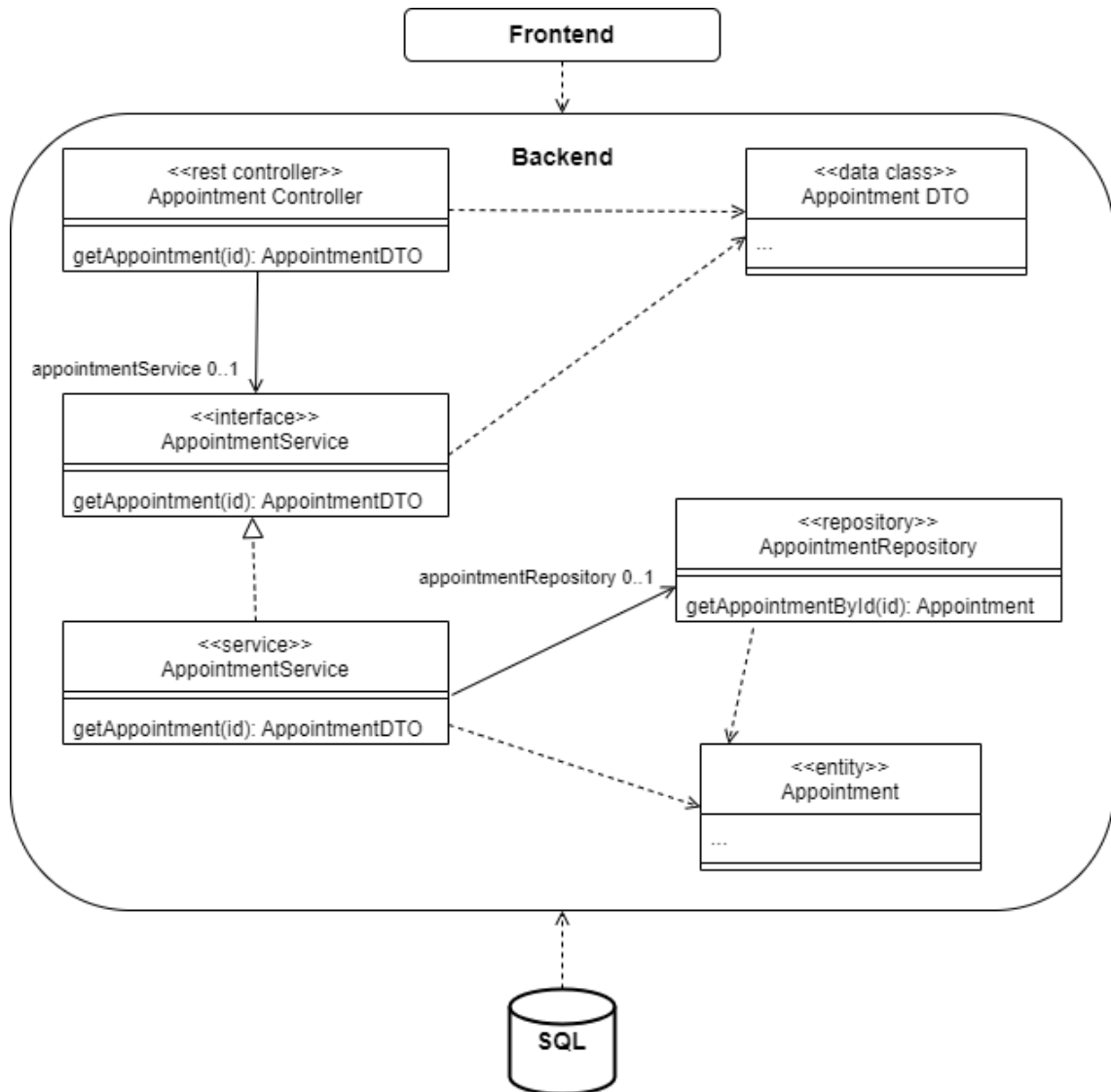
Mivel User Interface tervezéssel még nem volt sok tapasztalatom, ezért fontosnak tartottam egy olyan könyvtár használatát, amellyel könnyen tudok egységes és stílusos komponenseket létrehozni. A Material UI egy ennek megfelelő könyvtár volt, melynek dokumentálhatósága és népszerűsége könnyebbé tette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



8. 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 8. 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.

```
@Repository
interface AppointmentRepository: JpaRepository<Appointment, Long> {
    fun findById(customerId: Long): List<Appointment>
    fun findByIdAndDateGreaterThan(providerId: Long, date: Date):
    List<Appointment>
    ...
}
```

4. kódrészlet Példa repository interfész

A fenti példán (4. kódrészlet) látható egy ilyen AppointmentRepository, amivel az Appointment entitást kezelhetjük. Ez leszámazik a típusos JpaRepository interfészből, így ezen már elérhetőek a Felhasznált technológiák fejezetben említett műveletek, és egyedi lekérdezést is megfogalmaztunk függvény elnevezésével és paraméterezésével. A findByIdCustomerId függvény segítségével kereshetünk az Appointment példányok között a vevő azonosítója alapján. A findByIdAndDateGreaterThan függvény azokat a foglalásokat találja meg, amiknél egy bizonyos azonosítójú felhasználó a szolgáltató, és egy bizonyos dátum után van a foglalt időpont.

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.

```
@ControllerAdvice
class ExceptionControllerAdvice {
    @ExceptionHandler
    fun handleIllegalArgumentException(ex: IllegalArgumentException):
    ResponseEntity<String> {
        return ResponseEntity("Illegal argument: " + ex.message,
        HttpStatus.BAD_REQUEST)
    }
    ...
}
```

5. kódrészlet REST végpontok hibakezelése

A REST végpontok hibakezelése a `@ControllerAdvice` annotáció segítségével egy központi helyen valósul meg. Ezáltal a többi kontrollernek nem szükséges kezelnie a hibás eseteknél a választ, HTTP státusz kódot. A kontrollerek visszatérési értéke lehet az elvárt típusú, nem kell kiegészíteni egy opcionális error típussal. Mivel a hibás esetek központilag vannak kezelve, azonos kódokkal és hibaüzenetekkel, így a kiajánlott interfész is egységesebb lesz.

Ilyen központi hibakezelő az `ExceptionHandlerAdvice` metódus a 5. kódrészletből. Nem csak a frontenden, de a backenden is történik input validáció. Amennyiben egy kontroller hibásnak találja a bemenetet, `IllegalArgumentException` típusú hibát dob, amelyet a korábban említett központi metódus kezel majd le.

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

6. kódrészlet Auth0 security konfiguráció

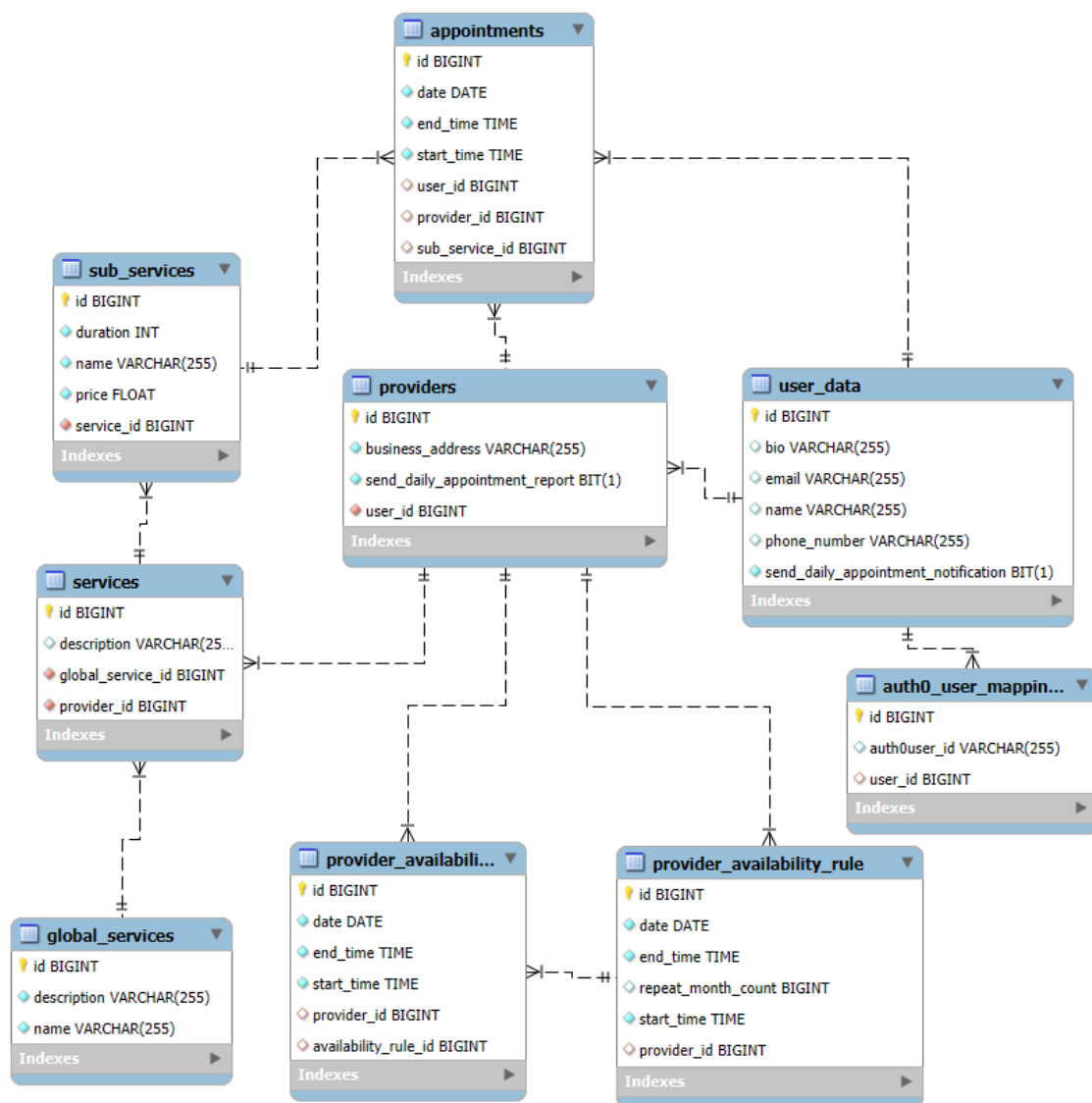
A fenti kódrészletben látható az alkalmazás security konfigurációja. A GET kérések kiszolgálására nincs szükség autentikációra, azonban a módosító műveletekhez már szükséges a bejelentkezés. Az auth0 azonosítás használatához még szükség volt az audience és issuer beállítására az `application.properties` 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 végezni a backend által definiált entitásokban és ismét gördülékenyen működött az összes JPA repository használata, így ez egy tényleg könnyen konfigurálható alternatíva volt.



9. 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 9. 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 időszavok ismétlődő szabályait tartalmazza az egyes szolgáltatókhoz. Ehhez kapcsolódó adat, például hogy hány következő hónapra írja ki az időszavokat, ezek mettől meddig legyenek.
- **provider_availability:** A szolgáltatókhoz tárolja azokat az időszavokat, amikben lehet hozzájuk időpontot foglalni. Ehhez szükség van a dátumra, az időszav 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 által 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


5.1 Profil

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




Name

Max Berger

 **E-mail address**

max.berger@testmail.com

 **Phone number**

*123456789

Enter a valid phone number

Would you like to receive e-mail notifications the day before your appointments? ☒

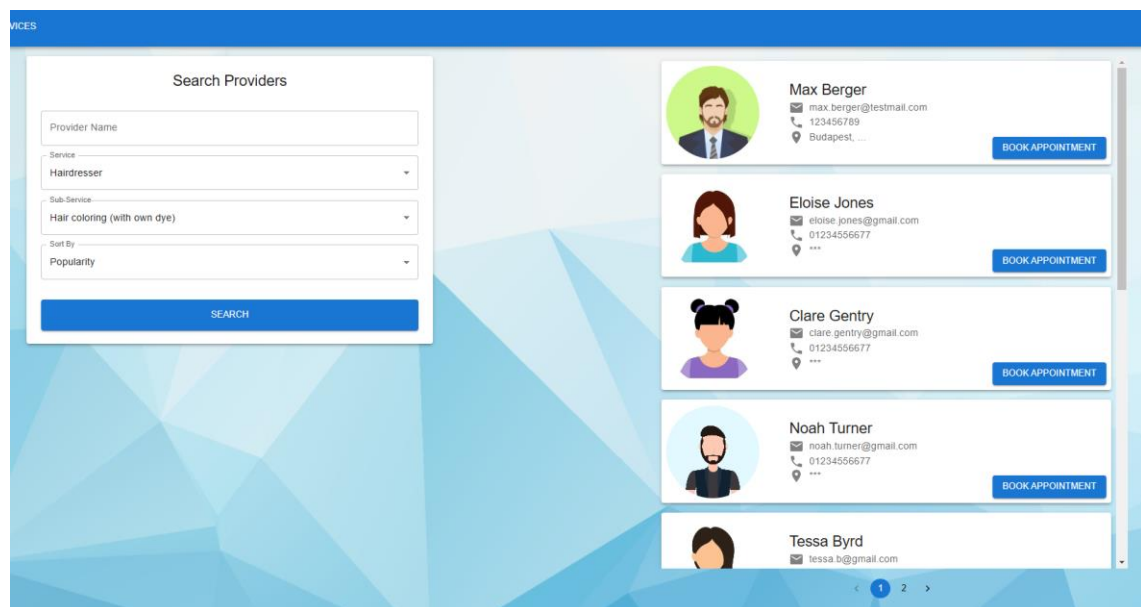
10. ábra Profil oldal

A 10. ábra egy módosítás alatt álló profil oldalt mutat be. A felhasználó a profilja módosításakor módosíthatja nevét, e-mail címét, telefonszámát és beállíthatja preferenciáit. A módosításokat az alsó gombokkal elvetheti vagy elmentheti. Mentés esetén az oldal ellenőrzi, hogy egyik megadott mező sem üres, emellett regex kifejezéssel validálja az e-mail és telefonszám helyes-e. Ha valamelyik adat nem megfelelő, akkor azt az ábrán látható piros hibaüzenettel jelzi. A hibák javítása után a felület sikeresnek tekinti az adatok javítását, és egy PUT HTTP üzenetet küld a backend számára. A backend is ellenőrzi a kapott adatokat, és elmenti a módosításokat az adatbázisba, amennyiben ez sikeres volt, majd visszaküldi a frissített adatokat.

Az adatokon kívül preferenciáit is beállíthatja, jelenleg csak egy ilyen van. Megadhatja, hogy szeretne-e emlékeztető e-mail értesítést kapni a foglalása előtti napon. Az értesítéseket az Értesítések és ütemezett feladatok fejezetben fejtem ki bővebben.

A profil oldal a kezdő állapotában nem módosított állapotban van. Ilyenkor alul egy edit profile gomb található, és az adatok text box helyett label formájában jelennek meg, ezen felül nem jelenik az e-mail értesítés preferencia sor.

5.2 Szolgáltatók listája



11. ábra Szolgáltatók listája

5.3 Időpont foglalás

Időpont foglalást a bejelentkezett felhasználó képes kezdeményezni. Nem csak a React frontenden és backenden van korlátozva ez az út, de maga a gomb sem jelenik meg egy nem bejelentkezett felhasználó számára. Az időpont foglaláshoz a felhasználónak először ki kell választania egy szolgáltatót a szolgáltatók listája oldalról, vagy egy szolgáltató áttekintő oldaláról tud ide navigálni.

BOOK MY SERVICES

Book Appointment

Max Berger
max.berger@testmail.com
123456789
Budapest, ...

1 Select Service 2 Select Date and Time 3 Confirm Booking

Select Service

Service
Hairdresser

Subservice
Hair bleaching

NEXT

12. ábra Időpont foglalás oldal

Az időpont foglalás oldal összképét a 12. ábra mutatja be. Itt láthatjuk a szolgáltató elérhetőségi adatait és profil képét. Alatta található egy folyamatjelző sáv lépésekkel. Az első lépés a szolgáltatás kiválasztása, a második az időpont kiválasztása, és végül az adatok megerősítése és foglalás.

Jelenleg a szolgáltatók csak általánosságban tudják megadni elérhetőségeiket. Ezt a felületet úgy terveztem, hogyha a jövőben szolgáltatásra specifikusan tudnak elérhetőséget megadni, akkor ezt ezen az oldalon is kényelmesen tudjuk kezelni.

A képen látható felső select mezővel kiválaszthatjuk, hogy a szolgáltató melyik fő szolgáltatását szeretnénk igénybe venni. Amint kiválasztottunk egyet, lehetőségünk van alszolgáltatást választani. Ha mindkettőt kiválasztottuk, akkor a next gomb engedélyezve lesz, és áttérhetünk a következő lépésre.

2 Select Date and Time

Select Date and Time

11/29/2024

15:15
15:30
15:45
16:00
16:15
16:30

BACK
NEXT

13. ábra Időpont foglalás oldal dátum kiválasztás

A foglalás időpontjának kiválasztásánál a fenti ábrán látható baloldali dátum választó komponenssel tudjuk kiválasztani a napot. Csak azok a napok vannak engedélyezve, amikre a szolgáltatónak van elérhető időpontja. Ez után az időpont választó komponenssel választunk az engedélyezett opciókból, és mehetünk a következő lépésre.

Select Date and Time

Confirm Booking

Provider name:	Max Berger
Service:	Hairdresser - Hair coloring
Date:	11/29/2024
Time:	15:30
Location:	Budapest, ...

BACK
SUBMIT

14. ábra Időpont foglalás oldal adat összegzés

Az utolsó lépésben ellenőrizhetjük a foglalásunk adatait, és a submit gombbal elküldhetjük. A lépések között visszafele is lehet menni. Az eddig megadott adatokat megjegyzi az oldal.

5.4 Foglalt időpontok kezelése

Todo: írni a daypilot-ról

5.4.1 Általános felhasználó

5.4.2 Szolgáltató

5.5 Szolgáltatások kezelése

A szolgáltató a My Services menüpontra kattintva éri el a szolgáltatásai kezelőfelületét. Itt lehetősége van az általa nyújtott összes szolgáltatást megtekinteni, módosítani és törölni, illetve frissíteni az üzleti profilját. Ezt az oldalt a többi bejelentkezett vagy nem bejelentkezett felhasználó is megtekintheti, azonban az adatokat nem tudják szerkeszteni.

Max Berger
Budapest, ...
123456789
max.berger@testmail.com
I have 10+ years experience in cosmetics and hairdressing.

MANAGE AVAILABILITY

HAIRDRESSER COSMETICS

Below you can see my hairdressing services. The prices may vary based on hair length and hair type. If you bring your own dye, I can give you a discount on coloring.

Subservices

Name	Duration (minute)	Price
Haircut	30	23 \$
Hair coloring (with own dye)	45	26
Hair coloring	45	30 \$
Hair bleaching	45	30 \$
Hair styling	30	26 \$

Portfolio

15. ábra Szolgáltatás kezelés felület

A 15. ábra mutatja be a szolgáltatások oldalt. Az oldal tetején található az üzleti profilja a szolgáltatónak. Itt röviden írhat életrajzáról, tapasztalatairól, és megadhatja a címet, ahol a szolgáltatást nyújtja. A modul jobb alsó sarkában található a manage availability gomb, amellyel a foglalható időpontok kezelése oldalra navigálhat, melyet a következő fejezetben fejték ki.

Az üzleti profil alatt található a szolgáltatásainak listája. Egy szolgáltatás a Szolgáltatók listája fejezetben említett módon 2 részből épül fel: a fő- és a alszolgáltatás. Fő szolgáltatás egyelőre csak 3 van: Fodrászat, Kozmetika és Manikűr. A szolgáltató a jobb oldalon található plusz gombbal tud felvenni új főszolgáltatást a legördülő menüből. Ezután egy új fül megjelenik a kiválasztott főszolgáltatás nevével, és alatta üres tartalommal. Ilyen fül például a képen látható Hairdressing és Cosmetics.

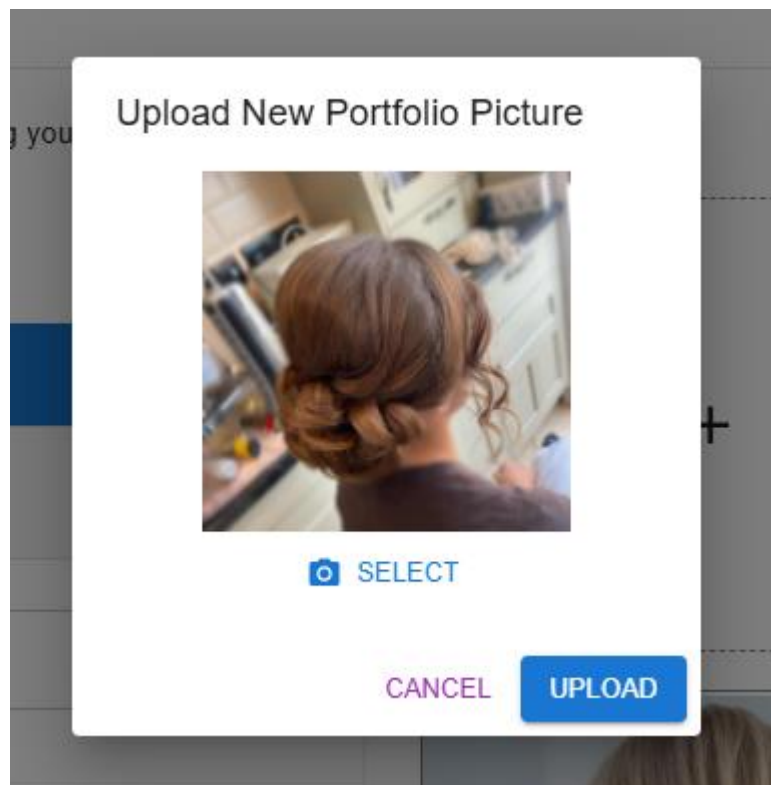
A 15. ábra jobb oldalán a szolgáltató portfóliója van feltüntetve az éppen kiválasztott szolgáltatáshoz. Általában egyszerre 6 kép látható egy oldalon, ettől az első és utolsó oldal térhet el. Amikor az utolsó oldalon kevesebb van, mint 6 kép, akkor a maradék helyeken azonos méretű, fekete szélű dobozok jelképezik a hiányzó képeket. Mivel lehet, hogy sok kép van a portfólióban, így a frontend lapozással kéri el ezeket a backendtől. Ehhez először a backendtől megkérdezi, hogy mennyi kép tartozik ehhez a szolgáltatáshoz. Ezután az oldalszám függvényében kéri azt a pár kép nevét, amennyit éppen meg kell jeleníteni.

```
@GetMapping("/{serviceId}/portfolio-pictures", params = ["from",
"amount"])
fun getPortfolioPictureNames(@PathVariable serviceId: Long, @RequestParam
from: Int, @RequestParam amount: Int): ResponseEntity<List<String>> {
    val uploadDirectory =
portfolioPicturesPath.resolve(serviceId.toString()).toFile()
    if (!uploadDirectory.exists()) {
        return ResponseEntity(ListOf(), HttpStatus.OK)
    }
    val pictureFiles = uploadDirectory.listFiles()
    pictureFiles.sortBy {
        it.lastModified()
    }
    val pictureNames = pictureFiles.map {
        it.name
    }
    return
ResponseEntity(pictureNames.subList(from.coerceAtMost(pictureNames.size),
(from+amount).coerceAtMost(pictureNames.size)), HttpStatus.OK)
}
```

7. kódrészlet Backend portfólió lekérdezés lapozással


A 7. kódrészleten látható a portfólió képek lekérdezésének a backend oldali implementációja. A REST API kérésnek URL query paraméterként adhatjuk meg, hogy hányadik képtől kérjük a kép fájlok nevét és hogy mennyit kérünk. Ha lapozással szeretnénk erőforrásokat kérni, akkor fontos, hogy ezek az erőforrások determinisztikus sorrendben legyenek, azaz ugyanarra a kérésre ugyanazokat a fájl neveket kapjuk vissza.

Ezt úgy oldottam meg, hogy módosítás dátuma szerint rendeztem a képeket, mielőtt kiválasztanánk melyik fájl neveket küldjük vissza a listából.












16. ábra Portfólió kép feltöltés dialógus ablak

A portfólióban az első plusz jeles téglalap csak a bejelentkezett szolgáltató saját oldalán jelenik meg. Erre rákattintva egy dialógus ablak nyílik, ahol újabb képet tölthet fel. A fenti ábrán (16. ábra) látható SELECT gombbal a fájl rendszerből kiválaszthatunk egy képet, aminek megtekinthetjük előnézetét. Feltöltésre csak a kép típusú fájlok vannak engedélyezve. Az UPLOAD gomb megnyomásával meg is jelenik a portfólióban az új kép.

Below you can see my hairdressing services. The prices may vary based on hair length and hair type. If you bring your own dye, I can give you a discount on coloring. 

Subservices

Name	Duration (minute)	Price	
Haircut	30	23 \$	 
<input type="text" value="Name"/> Hair coloring (with own dye)	<input type="text" value="Duration"/> -45 <small>Duration should be greater than 0 minutes</small>	<input type="text" value="Price"/> 26	
Hair coloring	45	30 \$	 
Hair bleaching	45	30 \$	 
Hair styling	30	26 \$	 

Rows per page: 5 ▾ 1-5 of 6 < >

17. ábra Alszoolgáltatások

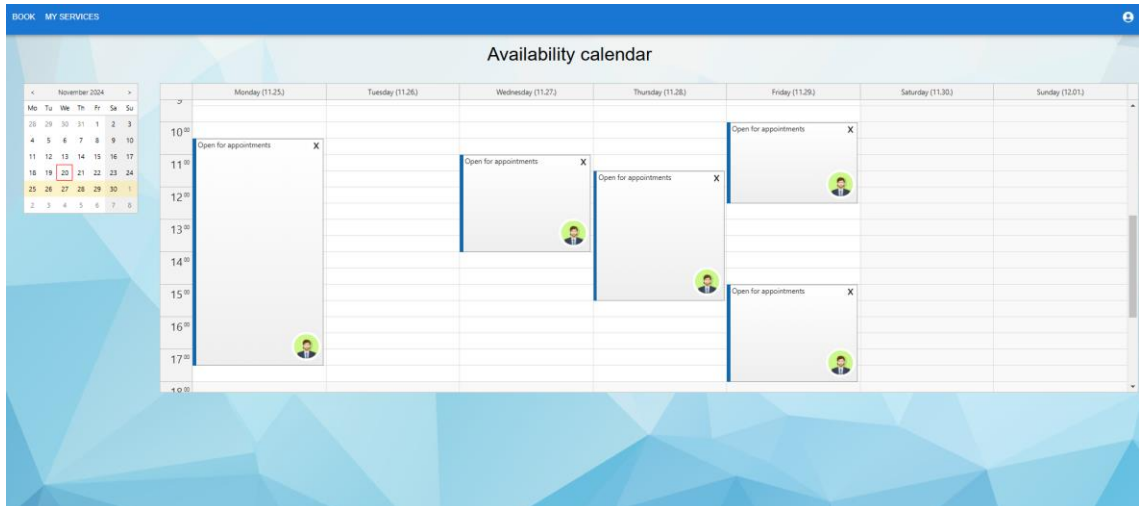
A portfóliótól balra található a kiválasztott szolgáltatás leírása, amivel további fontos információt adhat a vevőnek. Például, hogy a feltüntetett árak függhetnek a hajhossztól vagy a kívánt frizura komplexitásától. Ez alatt található az alszoolgáltatások részletei táblázatba rendezve, melyet a 17. ábra közelebbről vizualizál. Ez a táblázat lapozható, illetve be lehet állítani, hogy hány sor jelenjen meg egyszerre.

Az alszoolgáltatásoknak adhatunk egy leíró nevet, időtartamot és árat. Módosíthatjuk és törölhetjük a létrehozott sorokat a jobb oldalon található gombokkal, egy ilyen módosításra mutat példát a második sor. Új alszoolgáltatás létrehozására a táblázat utolsó sorát használhatjuk. Ez a második sorhoz hasonlóan feliratok helyett szövegdobozokat tartalmaz, melyek alap állapotban üresek. A mentés gombra nyomva a felület ellenőrzi a megadott adatokat, nem engedi a mentést és kiírja a hibát, ha valami problémát talál.

Jelenleg az alkalmazás a szolgáltatások időtartamát és foglalását negyed óránként tudja kezelni. Kisebb intervallumok esetén az időpont foglaláskor a vevőnek túl sok lehetőség jelenne meg. A negyedórás időszavokat használva a foglalási felület letisztult, de mégis elég részletesen meg tudjuk adni, hogy egy-egy szolgáltatás meddig tart. A szolgáltatások árát egyelőre dollárban jelenítjük meg, azonban ez könnyen a kívánt pénznemre szabható.

5.6 Foglalható időpontok kezelése

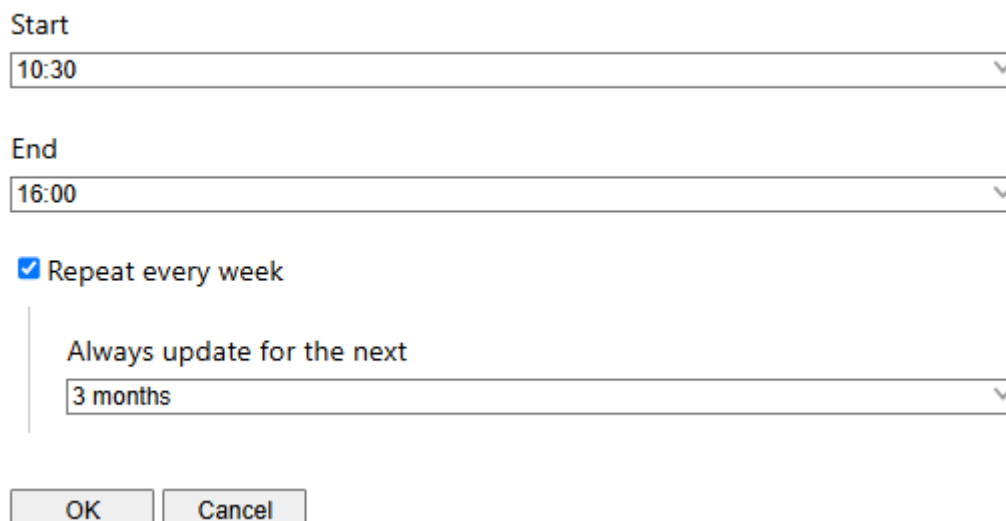
Ahhoz, hogy a felhasználók foglalni tudjanak egy szolgáltatóhoz, először a szolgáltatónak meg kell adnia az elérhető időpontjait. Ez nehézkes lenne egy a foglaláshoz hasonló dátum választóval, így ezt is a DaypilotCalendar komponens segítségével hoztam létre. A szolgáltató a szolgáltatások kezelése oldalról tud idénavigálni.



18. ábra Foglalható időpontok kezelése

Az képen látható heti nézetes naptár ábrázolja a foglalható időpontokat időintervallum formájában. Ezek azt jelzik mikor elérhető a szolgáltató, és a foglalt időpontoknak mindig bele kell férniük ebbe az időszakba. A továbbiakban elérhetőségi eseményként is hivatkozok a foglalható időintervallumra.

A bal oldalon található havi nézetes naptárból a szolgáltató kiválaszthatja, hogy melyik hetet szeretné látni a központi naptárban. Egy-egy esemény jobb alsó sarkában láthatja a szolgáltató a saját profilképét. Eseményt törölni a jobb felső sarokban található X jellel tudunk. Új elérhetőségi esemény felvételéhez a naptárból ki kell jelölnünk egy időintervallumot, és megadni annak részleteit.



Start

10:30

End

16:00

☒ Repeat every week

Always update for the next

3 months

OK Cancel

19. ábra Új elérhetőség dialógus ablak

Az intervallum kijelölése után a fenti ábrán látható dialógus ablak nyílik meg. Itt megadhatunk más kezdetet és véget az intervallumnak, ha nem jól jelöltük ki a sávot elsőre. A szolgáltatók megadhatnak ismétlődő elérhetőségi eseményeket szabályok definiálásával. Ha az ábrán látható Repeat every week kapcsolót bepipáljuk, akkor nem egy eseményt, hanem egy úgynevezett elérhetőségi szabályt hozunk létre. Egy ilyen szabály főbb adatai, hogy mettől meddig tartsanak az események, a hét milyen napján, és a következő hány hónapra lehessen előre foglalni.

Amikor egy új szabályt vesz fel a felhasználó, akkor automatikusan generál elérhetőségi eseményeket az alkalmazás minden megadott időszámba a következő x hónapig. Az események tárolják, hogy melyik szabályhoz tartoznak. A felhasználónak lehetősége van úgy törölni szabály által létrehozott eseményt, hogy a szabály és a többi esemény ettől függetlenül megmarad. Amikor töröl egy szabályt, akkor az összes ahhoz a szabályhoz tartozó esemény is törlődik.

Ezek a szabály alapján generált események nem csak a szabály létrehozásától számított x hónapra keletkeznek, hanem minden héten generálódik egy újabb elérhetőségi esemény. Erről az Értesítések és ütemezett feladatok fejezetben írok bővebben.

5.7 Értesítések és ütemezett feladatok

Mivel a projektem egy időpont foglaló alkalmazás, így fontosnak tartottam, hogy részletes értesítéseket kapjanak a felhasználók a foglalt időpontjaikról. Az értesítések e-mail formájában érkeznek a felhasználó profiljában megadott e-mail címre. Az értesítések

két különböző kategóriába sorolhatók: olyan értesítések, amelyek felhasználói művelet hatására keletkeztek, és olyanok, amik egy-egy ütemezett feladat eredménye. Ezen kívül vannak olyan ütemezett feladatok is, amelyek nem kapcsolódnak az értesítés küldéshez.

5.7.1 E-mail küldés

E-mail értesítés küldéséhez szüksége volt az alkalmazásomnak egy Simple Mail Transfer Protocol (SMTP) szerverre. Egy SMTP szerver képes e-mailt küldeni és továbbítani a felhasználók címére, emellett lehetővé teszi a biztonságos és megbízható kommunikációt a megfelelő konfiguráció mellett.

```
spring.mail.host=smtp.gmail.com
spring.mail.port=587
spring.mail.username=bme.appointment.app@gmail.com
spring.mail.password=***
spring.mail.properties.mail.smtp.auth=true
spring.mail.properties.mail.smtp.starttls.enable=true
```

8. kódrészlet SMTP szerver konfiguráció

A fenti kódrészleten látható az SMTP szerverem konfigurációja az application.properties fájlból. A google által üzemeltetett gmail SMTP szerverét használtam az e-mailek küldésére egy új bme.appointment.app@gmail.com címről. A konfiguráció utolsó két sora biztosítja az adatátviteli titkosítást, és a biztonságos kapcsolathoz szükséges hitelesítést. Egy-egy e-mail küldéséhez szükséges megadni a címzettet, az értesítésre specifikus tárgyat és a tartalmat.

5.7.2 Értesítések

Jelenleg 4 különböző fajta értesítést küld az alkalmazás. Foglalás lemondásakor annak a félnek nem küld e-mailt, aki a lemondást kezdeményezte, de a másik félnek igen. Például, ha a vevő nyomott rá a foglalás törlése gombra, akkor a szolgáltatónak fog értesítést küldeni az alkalmazás. A foglalás módosításakor a vevőnek küld levelet, benne az új módosított adatokkal. A szolgáltatónak ilyenkor soha nem küld, mivel csak ő tudja módosítani a foglalás adatait.

```
@Scheduled(cron = "0 0 20 * * *")
fun sendAppointmentNotifications() {
    val nextDay = java.sql.Date.valueOf(LocalDate.now().plusDays(1))
    val providerAppointments = mutableMapOf<Long,
MutableList<Appointment>>()
    appointmentRepository.findByDate(nextDay).forEach {
        if (it.customer.sendDailyAppointmentNotification) {
            ...
        }
    }
}
```

```

        }
        if (it.provider.sendDailyAppointmentReport) {
            ...
        }
    }
    providerAppointments.forEach {
        val mail =
        emailMessageFactory.createAppointmentReportEmail(it.value)
        if (mail != null) {
            emailService.sendMail(mail)
        }
    }
}

```

9. kódrészlet Ütemezett foglалás értesítés és beszámoló

A 9. kódrészlet mutatja be az ütemezett foglалás értesítést vevők és szolgáltatók számára. A fenti metódus lekéri a holnapi foglалások listáját, és e-mailt küld az összes vevőnek, aki szeretne értesítést kapni erről. A szolgáltatóknak pedig nem egyesével külön e-mailekben küldi ki a holnapi foglалásaikat, hanem egy táblázatos beszámoló jellegű üzenetet kap, ahol áttekintheti a napirendet és a vevők adatait.

A metódus felett található annotáció miatt minden nap este 8 órakor fut le ez a kód. Ezt a cron időzítési formátummal tudtam megadni, azaz 0 másodperckor, 0 perckor, 20 órakor és az összes (*) napon az összes hónapban.

5.7.3 Egyéb ütemezett feladatok

Az értesítés küldésén kívül van két egyéb ütemezett feladat, ami a szolgáltatók elérhetőségeit tartja karban. Minden nap éjfélt után egy perccel az alkalmazás kitörli a szolgáltatók összes elérhetőségi eseményét, ami a mai vagy korábbi időpontra vonatkoznak. Tehát nincs lehetősége a felhasználónak mai, vagy múltbeli időpontra jelentkeznie egy szolgáltatónál. A jelenlegi napra való foglалást azért vettem ki, mert ilyenkor a szolgáltatónak már lehet nem lesz ideje felkészülni erre a foglалásra, és a vevő inkább telefonon vagy e-mailben keresse ez esetben.

A szolgáltatók által megadott elérhetőségi szabályok megmondják, hogy a következő hány hónapra lehessen előre foglалni, így létre kell hozni ezeket az elérhetőségi eseményeket. Minden nap éjfélt után egy perccel összeszedi az összes olyan szabályt, amely a hét erre a napjára vonatkozik. Nem módosít vagy töröl semmilyen eseményt, ami a szabályhoz tartozik, mivel lehetséges, hogy a felhasználó már korábban kitörölt 1-2 alkalmat belőlük. Hozzáadja eseményként - 30 napos hónapokkal számolva - az x hónap múlva következő első alkalmat, amikor ugyanaz a hét napja, mint most. Például, ha a

szabály szerint hétfőnként 3-tól 4-ig elérhető a szolgáltató, akkor hétfőnként hozzáad egy három hónap múlva lévő hétfői elérhetőségi eseményt.

5.8 Tesztelés

5.9 Konténerizáció

A projektem során fontosnak tartottam, hogy könnyen telepíthető és platform független legyen. Ennek megvalósítására a népszerű Docker technológiát használtam. Az alkalmazásom 3 komponensből áll, így szükséges volt ezeket különböző konténerekbe tennem.

```
FROM node:18-alpine
WORKDIR /app
COPY package*.json ./
RUN npm install
COPY . .
RUN npm run build
EXPOSE 3000
CMD ["npm", "serve", "-s", "build"]
```

10. kódrészlet Frontend Dockerfile

A 10. kódrészlet a frontendem futtatásához szükséges konfigurációt írja le. A React alkalmazásom függőségeit, fordítását és futtatását a Node.js keretrendszeren belül a Node Package Manager segítségével kezelem. Ezért is a node:18-alpine előre elkészített image csomagot használtam fel rá, emellett az alpine egy minimalista linux disztribúció, aminek kis mérete miatt gyorsabb az image pull ideje. Ez a Dockerfile fájl a frontend mappájában található. Instalálja a package.json fájlban megadott függőségeket, bemásolja az összes forrás kódot, lefordítja és elindítja a szerver a 3000-es porton.

```
FROM gradle:latest AS BUILD
WORKDIR /app
COPY . .
RUN gradle build

FROM openjdk:21-jdk-slim
ENV JAR_NAME=appointment-app-backend-0.0.1-SNAPSHOT.jar
ENV APP_HOME=/app
WORKDIR $APP_HOME
COPY --from=BUILD $APP_HOME .

EXPOSE 8080
```

```
ENTRYPOINT exec java -jar $APP_HOME/build/libs/$JAR_NAME
```

11. kódrészlet Backend Dockerfile

Az alkalmazást futtató konténer indításakor praktikus, hogyha nem csak az előre elkészített .jar fájlt indítja el, hanem ő maga fordítja le a kódot és hozza létre ezt a fájlt. Így, ha változtatunk a kódon, mindig a legfrissebb változatát indíthatjuk az alkalmazásunknak. A 11. kódrészlet mutatja a backend Docker konfigurációt. Itt látható, hogy a Dockerfile-ban két különböző előre elkészített image csomagot is használtam. Az első gradle:latest képes gradle build parancsot kiadni a forráskódra, aminek hatására elkészül a futtatható jar fájl a build/libs mappában. Ezt felhasználva az open-dk:21-jdk-slim image csomagból kiindulva tudja futtatni a backend szerver a java segítségével a 8080-as porton.

Az SQL adatbázist tartalmazó konténer létrehozásához nem volt szükség külön Dockerfile konfigurációra, mivel az előre elkészített mysql:8.0 image tartalmazta a szükséges beállításokat. Eleinte nem használtam konténert az adatbázis szerver kezelésére, azonban a Docker-es konfigurációja sokkal egyszerűbb volt, mint a host gépen.

```
version: '3.8'

services:
  frontend:
    build:
      context: ./frontend/appointment-app-frontend
    ports:
      - "3000:3000"
    depends_on:
      - backend
    environment:
      - REACT_APP_API_URL=http://localhost:8080

  backend:
    build:
      context: ./backend/appointment-app-backend
    ports:
      - "8080:8080"
    depends_on:
      - db
    environment:
      - SPRING_MYSQL_DATASOURCE_URL=jdbc:mysql://db:3306/docker_mysql
      - SPRING_MYSQL_DATASOURCE_USERNAME=root
      - SPRING_MYSQL_DATASOURCE_PASSWORD=appointment_app_backend

  db:
    image: mysql:8.0
    ports:
      - "3307:3306"
```

```
environment:
  - MYSQL_ROOT_PASSWORD=appointment_app_backend
  - MYSQL_DATABASE=docker_mysql
volumes:
  - mysql_data:/var/lib/mysql

volumes:
  mysql_data:
```

12. kódrészlet docker-compose.yaml fájl

A 12. kódrészlet mutatja be az alkalmazás Docker compose konfigurációját. A különböző komponensek leírásai a services alatt található. A Docker létrehoz egy alapértelmezett hálózatot, ahol ezek a szolgáltatások futnak, így tudnak egymással kommunikálni. Például a backend service a db:3306 címen éri el a MySQL adatbázist, ahol db a service neve, és 3306 a szolgáltatás portja a többi konténer felé. Ezt a backend a SPRING_MYSQL_DATASOURCE_URL környezeti változóként kapja meg az application.properties fájlban.

Az adatbázis service a 3307-es porton teszi elérhetővé a hozzáférést a host gépnek. Erre azért volt szükséges, mivel a host gépen már volt MySQL adatbázisom ezen a porton, amelyet nem szerettem volna törölni, és így a konténeren belüli adatbázist is tudtam kezelni a MySQL workbench segítségével a megadott porton. Létrehoztam egy mysql_data nevű volume-ot, ami lehetőséget adott az adatbázis adatok tartós tárolására. Enélkül az adatok törlődnének a konténer minden egyes leállításakor.

A docker-compose.yaml fájl mappájában a docker compose up parancs kiadásával elindul a teljes alkalmazás mind a három komponensével. Mivel az egyes rétegek közt számít a futási sorrend, így ezt a depends_on paraméter beállításával specifikáltam. Elsőként az adatbázis szerver indul el, mivel tőle függ a backend szerver. Ezután a backend indul el és utána a frontend. Gyakran használtam fejlesztés közben az alkalmazás konténerizált változatát, így legtöbbször a --build kapcsolóval futtattam a docker compose up parancsot, ami kikényszeríti az image csomagok újraépítését. Ezzel ugyan tovább tartott az alkalmazás indulása, de mindig a legfrissebb kódot tartalmazta.

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