SAPIENTIA ERDÉLYI MAGYAR TUDOMÁNYEGYETEM
MAROSVÁSÁRHELYI KAR,
INFORMATIKA SZAK

SAPIENTIA
ERDÉLYI MAGYAR
TUDOMÁNYEGYETEM

Vitalink Mobilalkalmazás: Pulzusmérés PPG technológia alkalmazásával

TUDOMÁNYOS DIÁKKÖRI KONFERENCIA

Témavezető:
Novák Pálma-Rozália,
Egyetemi adjunktus

Végzős hallgató:
Bús Gergő-Vince, Orbán Botond

2025

Kivonat

A szív- és érrendszeri betegségek világszerte a vezető elhalálozási okok közé tartoznak. A modern életmód – a mozgásszegény életvitel, a helytelen táplálkozás és a fokozott stressz – jelentősen növeli ezen betegségek kialakulásának kockázatát. Ennek ellenére a népesség jelentős része nem fordít hangsúlyt az egészségük megőrzésére, mivel a hagyományos orvosi vizsgálatok időigényesek és nem mindig elérhetők.

Dolgozatunk célja a mobileszközök kamerájával végzett szívritmusmérés lehetőségeinek vizsgálata, valamint a módszer hatékonyságának és megbízhatóságának értékelése. Az okostelefonok alkalmazása egy könnyen elérhető és felhasználóbarát megoldást kínálhat az egészségmonitorozásra, hozzájárulva a szív- és érrendszeri állapot folyamatos nyomon követéséhez és fenntartásához.

Munkánk során a fotopletizmográfia (PPG) elvét alkalmaztuk, amely az iparban széles körben elterjedt szívritmus-meghatározási technika. Népszerűségét non-invazív jellege és könnyű elérhetősége biztosítja, mivel egyszerűen alkalmazható mind orvosi, mind fogyasztói eszközökben. Az alkalmazásunk a React Native keretrendszerben készült, míg a szerveroldalt a Java Spring biztosítja. A mérések pontosságát különböző külső eszközökkel, például okosórával és pulzoximéterrel végzett összehasonlító vizsgálatokkal ellenőriztük.

Kulcsszavak: szív, fotopletizmográfia, okostelefon, pulzus, mérés, egészség

Tartalomjegyzék

Bevezető

A mai világban az okostelefonok gyakorlatilag mindig kéznél vannak, hiszen ezek segítségével tájékozódunk, vásárolunk, kommunikálunk, és ezáltal kifejezetten hasznos eszközeivé váltak a mindennapjainknak. Ez a felgyorsult és rohanó világ azonban azt is maga után vonja, hogy az emberek egyre kevesebb figyelmet fordítanak saját egészségükre, nagyobb figyelmet kap a kényelem és a problémák könnyű megoldása. Ennek következménye például a gyorsételek elterjedése, amely sok személy számára elengedhetetlen, hiszen nem marad idejük főzni a napi teendők mellett, ami az étkezés minőségének romlását eredményezi.

Ez a probléma észlelhető az országos és világadatok között is, hiszen a szívbetegségek a vezető halálokok közé tartoznak. Ezt statisztikai adatok is alátámasztják. Az Egészségügyi Világszervezet (WHO) adatai szerint ezek a betegségek évente több millió ember életét követelik, és gyakran megelőzhetőek lennének időben történő felismeréssel és életmódváltással [Wor23].

Ennek okán különösen fontossá válik az, hogy könnyen elérhető módszerei valósuljanak meg az egészség monitorozásának. Ez azonban a hagyományos módszerekkel nem működhet, hiszen egy orvosi látogatás gyakran képes a fél napunkat felemészteni. Az okostelefonok használatával azonban ezt a problémát orvosolni lehetne, és egy lehetséges megoldást kínálna a felmerülő megbetegedések korai észlelésére.

Fontos azonban hangsúlyozni, hogy az ilyen jellegű technológia nem helyettesítheti az orvosi vizsgálatokat, és nem célja azok kiváltása. Az okostelefonos alkalmazás inkább egy kiegészítő eszköz, amely segíthet abban, hogy tudatosabbak legyünk saját testünkkel és állapotunkkal kapcsolatban. Arra szeretnénk ösztönözni mindenkit, hogy figyeljen oda jobban önmagára, és használja ki azokat az eszközöket, amelyek támogatást nyújthatnak ebben.

Az alkalmazás, amelyet erre a célra fejlesztettünk, a Vitalink nevet kapta. Célja, hogy az okostelefonok kamerájával egy gyors és egyszerű megoldást nyújtson a pulzusmérésre, amelyből fontos adatokat nyerhetünk ki. React Native és Expo használatával valósítottuk meg az alkalmazást, annak érdekében, hogy több platformon is elérhető legyen, viszont az iOS operációs rendszerrel fennakadásokba ütköztünk, amelyeket a korlátozott idő és erőforrás miatt nem volt lehetőségünk megoldani. A mérések megvalósításához a PPG technológiát használtuk fel, amelyről a későbbiekben fog szó esni.

A dolgozatban vizsgáljuk, hogy mennyire képes egy ilyen fajta alkalmazás pontosan tükrözni a pulzusszámot, illetve kimutatni ennek a normalitását. Úgy gondoljuk, hogy egy ilyen alkalmazás, a megfelelő pontosság elérésével, egy nagyon hasznos mindennapi eszközzé válhat mindenki számára.

1. fejezet

Szakirodalmi áttekintő

Annak érdekében, hogy megfelelően átlátható és érthető legyen a dolgozat témája, szükség van néhány kulcsfontosságú technológia és fogalom megértésére. A következőkben ezek az eszközök illetve alapfogalmak kerülnek bemutatásra.

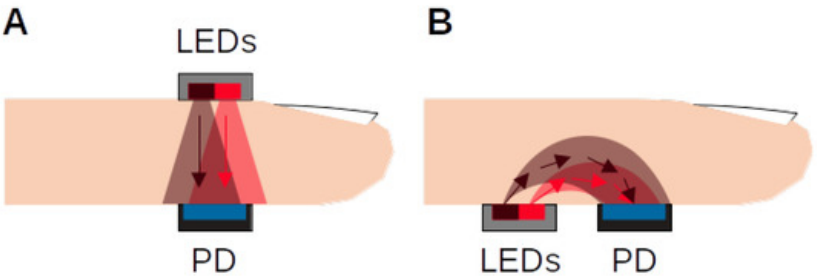
1.1. A PPG technológia

A megvalósítás központi eleme a **Fotopletizmográfia (PPG)** amely egy nem invazív módszere a szívritmus rögzítésének. Ez a technika lehetővé teszi, hogy egy okostelefon kamera segítségével észleljük a felhasználó ujjaiban levő fényintenzitás-változásokat. Ezek a változások eredményezik a jelet, amely a szívritmust képviseli.

A technológia működésének alapja, hogy a bőr alatti állományok különböző módokon tudnak elnyelni bizonyos hullámhosszúságú fényeket. Az ujjban nem található olyan anyag, amely nagyobb képességgel rendelkezik a fény elnyelésére, mint a vér. Fontos azonban, hogy hogyan használjuk a rendelkezésünkre álló fényforrást. A videókamerák általában RGB formátumban rögzítik a tartalmat (piros, zöld, kék), de van egy plusz csatorna is (az úgynevezett alfa, ami az átlátszóságot jelzi) viszont számunkra ez a csatorna nem jelentős. Előzetes kutatások kimutatták, hogy a vér hemoglobinja a legjobban a zöldhöz közeli fényeket nyeli el – pontosabban azokat, amelyek 500-600 nanométer intervallumban levő hullámhosszal rendelkeznek. A zöld fény (550 nm) pont ebbe a tartományba esik, és ígéretesebbnek bizonyult mind a piros, mind a kék fényeknél, ezért erre esett a választás [JZS21].

A fotopletizmográfia jelenleg széleskörben használt az iparban, mivel az okosórák és pulzoximéterek működése is ezen alapul. Két módszert különböztetünk meg a jelek felfogására, amelyek különböznek mind hatékonyságban, mind a szükséges utólagos feldolgozásban (lásd: 1.1-es ábra). Az első az *áteresztő módszer* amely keretén belül a szöveteken átszűrődő fény erősségét mérjük. Ebben az esetben, amikor a hajszálerek nagy mennyiségű vért tartalmaznak, akkor a jelünk gyengébb, a vértérfogat csökkenésével arányosan pedig erősödni fog a jel. Ez a megközelítés akkor optimális, hogyha a fényforrás és a szenzor az ujj két külön oldalára elhelyezhető, mint például a pulzoximéternél. A második megközelítés a *visszaverő módszer*. Ennek működése ellentétes az előzőével, hiszen a növekedő vértérfogat ebben az esetben a jel erősödésével jár. Ez már könnyebben felhasználható, mivel elegendő egy eszközt ráhelyezni az ujj felületére, és nincsen szük-

ség csíptető mechanizmusra. Az okosórákban ez a megoldás található, és a dolgozatunk keretén belül is ezt a megközelítést alkalmaztuk. [TMSY14]



1.1. ábra. PPG jel felfogási módszerei

1.2. Jelfeldolgozás

A pontos és releváns végeredmények meghatározásához elengedhetetlen az adatok feldolgozásának helyes végrehajtása és optimalizálása. Az olyan típusú jeleknél, mint a PPG, nagy figyelmet kell fordítani arra, hogy az adatok feldolgozásakor ezek ne torzuljanak, hiszen a legapróbb változás is különböző eredményeket tud előidézni.

Eleinte egy egyszerűbb megközelítést implementáltunk és teszteltünk, ez a **Mozgó átlagolás**. Ennek működése rendkívül egyszerű: a szomszédos adatpontok átlagértéke alapján mozgatja el az aktuális adatpontot. Ez a megoldás nem bizonyult túl hatékonynak, hiszen különböző paraméterek tesztelése után sem sikerült megkapni az általunk elvárt eredményt.

Számos weboldal és kutatás böngészése után rátaláltunk egy, a miénkhez hasonló dolgozatra, amelyben szerepelt a **Savitzky-Golay** szűrő. Ez a szűrő nem egyszerűen elsimítja a paraméterként átadott adathalmazt, hanem ügyel arra, hogy a részletek megmaradjanak (a mi esetünkben ilyenek a pulzus hullám-völgyek pontos pozíciói). Az algoritmus paramétereit tesztelve, megtaláltuk a számunkra optimális beállítást, amely képes kellően feldolgozni a nyers adatokat, melyeket a későbbiekben a percenkénti szívverések számának, illetve más információk meghatározására tudunk felhasználni [ANP23].

1.3. Kihívások, nehézségek

A munkánk során számos kihívásba ütköztünk, tekintve, hogy nagyon érzékeny adatokkal dolgoztunk, amelyekben rendkívül könnyen tud hiba képződni. Ebben az esetben a legapróbb hibák is tudnak jelentős eltéréseket okozni a végső eredményekben, ami nem elfogadható.

Elsősorban a PPG hátrányait szeretnénk megemlíteni, mivel ezeknek a kiküszöbölése jelentette a pontosabb eredményeket. Mivel a jelet a fényintenzitás változása okozza, ezért a külső fényhatások is erős befolyással tudnak lenni a rögzített jelre. Bár nagyobb gyártók már kifejlesztettek módszereket ennek a kiküszöbölésére, a mi esetünkben korlátozottak a lehetőségek. Ahhoz, hogy minél pontosabb és jobb eredményt tudjunk elérni, kiemelten figyeltünk az adatok feldolgozására és a zajcsökkentésre.

A megfelelő zajcsökkentési algoritmus(ok) kiválasztása is egy jelentős kihívást okozott. Számos ilyen módszer létezik, és mindegyik más szituációban okoz jobb eredményt. Annak érdekében, hogy a célunknak legmegfelelőbb módszert kiválasszuk, több tudományos kutatást is elemeztünk. Ezekben a kutatásokban bizonyos teszteknek voltak alávetve az algoritmusok. Az igazi nehézség itt abban rejlett, hogy nagyon sok opció közül egy algoritmust kellett kiválasztanunk, amely optimális volt a projektünk számára. Ez azonban nem volt egy egyértelmű döntés, mivel több szempont alapján is mérlegelnünk kellett, mint például a pontosság vagy az adatveszteség.

2. fejezet

Hasonló alkalmazások

2.1. Népszerű alkalmazások a piacon

A pulzusmérő alkalmazások piacán több elismert és széles körben használt megoldás érhető el. A piackutatás során részletesen megvizsgáltunk néhány ilyen alkalmazást, köztük az Instant Heart Rate: HR Monitor és a Cardiograph - Heart Rate Meter. Ezek az alkalmazások főként a pulzus mérésére és monitorozására szolgálnak, azonban bizonyos területeken hiányosságokkal küzdenek, amelyeket a Vitalink megoldani kíván, így új szintre emeli a pulzusmérő alkalmazások funkcionalitását és használhatóságát.

2.1.1. Instant Heart Rate: HR Monitor

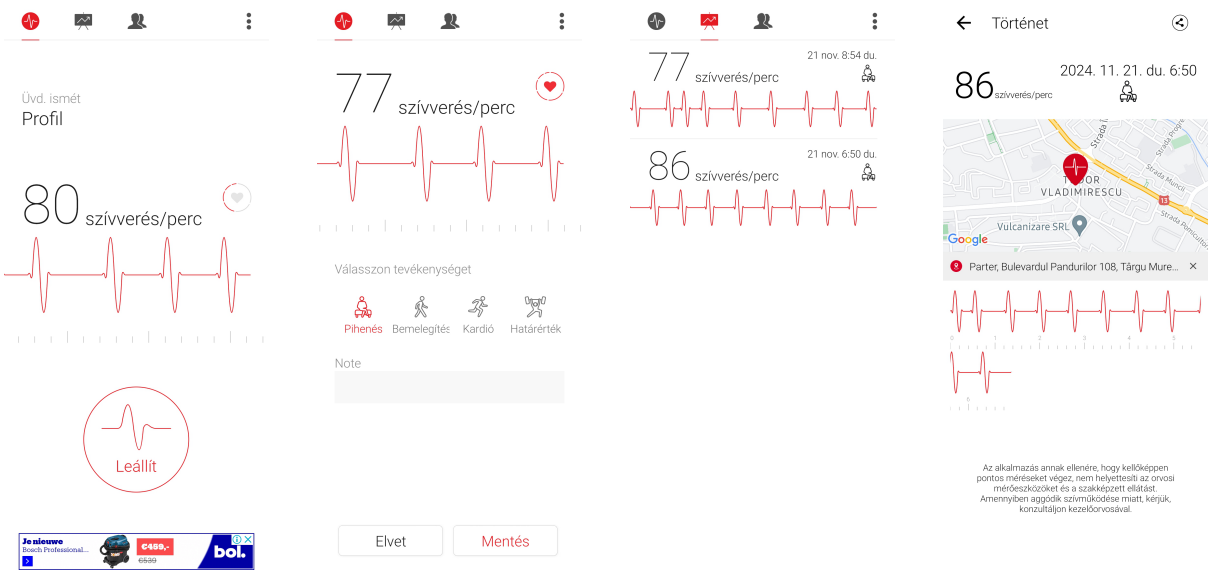
1. Erősségek:
    - Egyszerű használat, intuitív felhasználói felület.
    - Nagy pontosságú pulzusmérés a kamera és LED használatával.
    - Grafikonos megjelenítés az eredmények vizualizálására.
    - Különböző aktivitási szintek figyelése (pl. nyugalmi pulzus, edzés közben mért pulzus).
2. Gyengeségek:
    - Nem tartalmaz lehetőséget a szívritmus adatok megosztására vagy családi együttműködésre.
    - Korlátozott kiegészítő funkciók az alap pulzusmérésen túl.
    - Előfizetéshez kötött további funkciók.

(a) mérés **(b)** mérés után **(c)** mérések visszanaplózása

2.1. ábra. Instant Heart Rate: HR Monitor

2.1.2. Cardiograph - Heart Rate Meter

1. Erősségek:
   - Több felhasználói profil támogatása, így egy eszközön több családtag pulzusa is nyomon követhető.
   - Nagy pontosságú pulzusmérés a kamera és LED használatával.
   - Grafikonos megjelenítés az eredmények vizualizálására.
   - Különböző aktivitási szintek figyelése (pl. nyugalmi pulzus, edzés közben mért pulzus).
2. Gyengeségek:
   - Nem tartalmaz lehetőséget a szívritmus adatok megosztására vagy családi együttműködésre.
   - Korlátozott kiegészítő funkciók az alap pulzusmérésen túl.
   - Előfizetéshez kötött további funkciók.



(a) mérés **(b)** mérés után **(c)** visszanaplózás **(d)** részletes adatok

2.2. ábra. Cardiograph - Heart Rate Meter

2.2. Összehasonlítás a Vitalink alkalmazással

A piacon található hasonló alkalmazások funkciói és technológiái alapján a Vitalink alkalmazás nem csupán a pulzusmérésre fókuszál, hanem egy átfogó egészségügyi és biztonsági megoldást kínál, amely új szintre emeli a felhasználói élményt. Az alábbi kulcsfontosságú területeken nyújt kiemelkedő többletértéket:

2.2.1. Orvosi együttműködés

A Vitalink kiemelt funkciója, hogy integrálja az orvos szerepkörét, amely lehetővé teszi, hogy a rendszer azonnali értesítést küldjön az orvos számára, ha rendellenességet észlel, a felhasználó pulzusmérésében. Az orvos ezután hozzáférhet a felhasználó mérési adataihoz, így gyors szakmai segítséget nyújthat kritikus helyzetekben. Ez a funkció biztosítja a megelőzés lehetőségét és az egészségügyi kockázatok minimalizálását, amit egyetlen másik piaci szereplő sem kínál ilyen szinten.

2.2.2. Adatmegosztás ismerősökkel

A Vitalink másik forradalmi funkciója a valós idejű adatmegosztás. A felhasználók hozzárendelhetik közeli családtagjaikat vagy barátaikat, akik értesítéseket kapnak, ha rendellenességet észlel az alkalmazás a felhasznó pulzusmérése után. Ez egyedülálló biztonsági hálót biztosít, amely lehetővé teszi, hogy a felhasználók és szeretteik figyelemmel kísérjék egymás egészségi állapotát.

Ez a fajta valós idejű együttműködés nemcsak a felhasználó közvetlen egészségügyi támogatását teszi lehetővé, hanem erősíti a közösségi támogatást is, amely különösen fontos idősek vagy krónikus betegek számára.

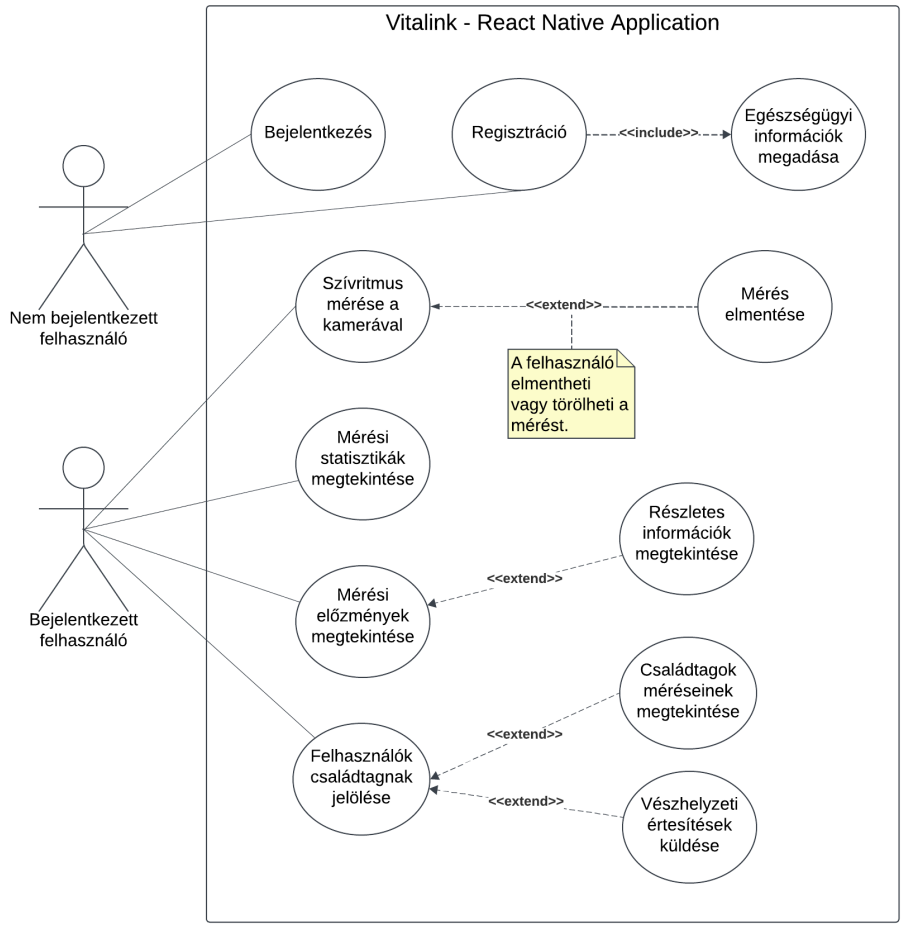
2.2.3. Többletérték a felhasználók számára

A Vitalink egy olyan átfogó ökoszisztémát kínál, amely ötvözi a piacon jelen lévő technológiai megoldásokat és az egészségügyi biztonságot. A funkciók kombinációja – mint például az orvosi integráció, a családi adatmegosztás és az értesítési rendszer – egyedi helyet biztosít az alkalmazás számára a piacon, amely túlszárnyalja a hagyományos pulzusmérő alkalmazások lehetőségeit.

3. fejezet

Rendszerspecifikációk

3.1. Felhasználói követelmények



3.1. ábra. Használati eset diagram

3.1.1. Szereplők és azok leírása

- Páciens/Felhasználó (Patient/User): A fő célcsoport, aki az alkalmazást aktívan használja. A páciens regisztrál, belép az alkalmazásba, méréseket végez, és hozzáfér az előző mérési adataihoz. Lehetősége van családtagok vagy orvosok meghívására is a kapcsolatok közé.

3.1.2. Használati esetek leírása

1. **Regisztráció**: Az alkalmazás elsődleges belépési pontja, ahol a felhasználók három fázison keresztül adják meg adataikat. Először a fiókkal kapcsolatban (email, jelszó), utána a személyes adataikat (név, születési dátum, stb.), majd legvégül az egészségügyi információikat (vértípus, rendellenességek). A regisztráció során a rendszer kommunikál a szerveroldallal az adatok validálásáért és tárolásáért.

2. **Bejelentkezés**: A már regisztrált felhasználók számára lehetőség a fiókjukhoz való hozzáférésre. Ezen kívül azonban lehetőség van Google fiók használatával is bejelentkezni, amely lehetővé teszi a regisztráció folyamatának átugrását. A bejelentkezés során a hitelesítési folyamat biztosítja, hogy csak a jogosult felhasználó férjen hozzá az adatokhoz.

3. **Szívritmus mérése kamerával**: Az alkalmazás fő funkcionalitása. A felhasználó ráhelyezi az ujját a kamerára és elindítja a mérést. Ezalatt láthatja a valósidejű fluktuációkat a fényintenzitásban, egy diagram segítségével, amely a mérés alatt is jelen van, majd a mérés után teljes egészében is látható lesz.

4. **Mérés mentése**: A rögzített méréseket menteni és törölni is tudja a felhasználó.

5. **Mérési statisztikák megtekintése**: A felhasználónak lehetősége van megtekinteni az jelen hónap méréseit. Ez két részre bomlik. Rendelkezésére állnak nagyobb intervallumokra való összesített adatok, ezen kívül pedig napokra osztva is megtekintheti a statisztikákat.

6. **Mérési előzmények megtekintése**: A rögzített méréseket listázva kapja meg a felhasználó, illetve ezek megnyomásával az adott mérés információit tudja megtekinteni. Ez a funkció szintén csak a bejelentkezett felhasználók számára érhető el.

7. **Felhasználók családtagnak jelölése**: Az alkalmazáson belüli kapcsolat rendszer lehetővé teszi, hogy a felhasználók bejelöljék egymást mint családtagok. Ez vészhelyzeti esetekre lehet a leghasznosabb, mivel a bejelölt családtagok értesítést kapnak az egymás méréseiről, ha azok veszélyt jeleznek.

Használati eset	**Szereplő**	**Előfeltétel**	**Utófeltétel**
Bejelentkezés	Nem bejelentkezett felhasználó	A felhasználó rendelkezik egy létező fiókkal.	A felhasználó átirányítódik a kezdőoldalra, már bejelentkezett állapotban.
Regisztráció	Nem bejelentkezett felhasználó	A felhasználó rendelkezik egy működő email címmel	A felhasználó átirányítódik a kezdőoldalra, már bejelentkezett állapotban.
Szívritmus mérése a kamerával	Bejelentkezett felhasználó	A felhasználó eszköze rendelkezik egy kamerával és egy villanófénnyel.	Átirányítódik a mérést összegző oldalra.
Mérés elmentése	Bejelentkezett felhasználó	A felhasználó sikeresen rögzített egy mérést.	A mérés bekerül az adatbázisba.
Mérési statisztikák megtekintése	Bejelentkezett felhasználó	A felhasználó rendelkezik lementett mérésekkel.	Különböző metrikák alapján tekinthetőek meg grafikonok az elmentett mérésekről.
Mérési előzmények megtekintése	Bejelentkezett felhasználó	A felhasználó rendelkezik lementett mérésekkel.	Megtekinthetőek listázva a mérések és azok információi.
Részletes információk megtekintése	Bejelentkezett felhasználó	A felhasználó rendelkezik lementett mérésekkel.	Megtekinthetőek az adott mérés információi.
Felhasználók családtagnak jelölése	Bejelentkezett felhasználó	Nincsen előfeltétel	A megjelölt felhasználó kap egy értesítést és a profil oldalon listázódni fognak egymásnak.
Családtagok méréseinek megtekintése	Bejelentkezett felhasználó	A felhasználó rendelkezzen legalább egy applikáción belüli kapcsolattal.	Láthatóvá válnak minimális információk a bejelölt felhasználó méréseiről.
Vészhelyzeti értesítések küldése	Bejelentkezett felhasználó	A felhasználó rendelkezzen legalább egy applikáción belüli kapcsolattal.	Veszélyt jelző eredmények esetén, a kapcsolatok értesítést kapnak.

3.1. táblázat. Használati esetek specifikációi

3.2. Rendszerkövetelmények

3.2.1. Funkcionális követelmények

Autentikáció

A felhasználónak regisztrálhat egy új fiókot e-mail címével, amellyel ezután lehetősége van bejelentkezni. Emellett bejelentkezhet Google-fiókjával is. Ha úgy dönt, hogy nem szeretne regisztrálni, vagy bejelentkezni, akkor lehetősége van vendégként kipróbálni az alkalmazást. Ebben az esetben a felhasználó csak a pulzusmérést érheti el.

Profil

A bejelentkezés után a felhasználó a Profil oldalra kerül, ahol megtekintheti fiókja adatait. Ezen az oldalon tudja kezelni a családtagokat, akiket hozzáadott a listájához. Navigáció szempontjából a felhasználó rendelkezésére áll egy sáv a képernyő alján, aminek a segítségével a különböző oldalak között tud navigálni.

Statisztika

A statisztikai oldalra érkezve, különböző paraméterek alapján létrehozott diagramokat és információkat tekinthet meg a felhasználó a saját méréseivel kapcsolatban.

Mérés

A mérés oldalra navigálva az applikáció biztosít egy útmutatót a felhasználó számára, annak érdekében, hogy minél pontosabb mérést tudjon végrehajtani. A mérés gomb megnyomásával a felhasználó elindítja a mérési folyamatot, amely 15 másodpercig tart. Az idő leteltével láthatja szívritmusát szívverés/perc formátumban. Az eredmény elmentésre kerül az adatbázisban, amit később a statisztikai oldalon lehet elérni.

Beállítások

A beállítások oldalon a felhasználónak lehetősége van visszajelzést küldeni a fejlesztőcsapat számára annak érdekében, hogy az alkalmazás gyorsabban és hatékonyabban fejlődhessen, illetve a hibák minél hamarabb kiküszöbölődjenek. Itt található a kijelentkezés is.

3.2.2. Nem-funkcionális követelmények

Tekintve, hogy az alkalmazás React-Native-vel készült, iOS és Android platformon is elérhető. A méréshez szükséges `react-native-vision-camera` csomag miatt szükséges az alább meghatározott operációs rendszer verziók használata. A mérések precizitása érdekében az alkalmazás csak olyan eszközökkel engedélyezi a mérést, amelyek rendelkeznek vakuval.

Minimális rendszerkövetelmények:

- Androidot vagy iOS-t futtató eszköz

- Android verzió: 8.0+
- iOS verzió: 12+
- Kamera és hozzá tartozó villanófény

4. fejezet

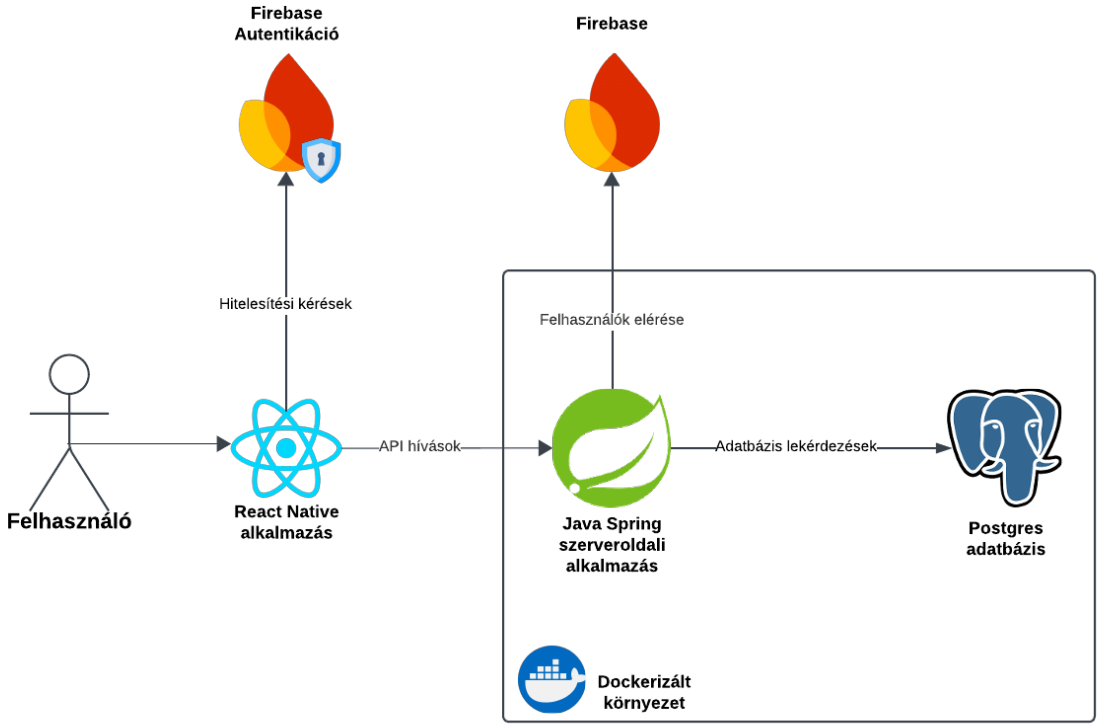
Tervezés

4.1. Architektúra

A projekt architektúrája egy modern, robosztus, skálázható, könnyen karbantartható rendszert valósít meg. Az architektúra kialakításakor figyelembe vettük a korábbi tapasztalatainkat, különösen a nyári gyakorlat során használt technológiákat, ahol nem rendelkeztünk saját backenddel vagy adatbázissal. Akkor a Firebase szolgáltatásait használtuk mind hitelesítéshez, mind az adatkezeléshez, mivel gyorsan implementálható és könnyen integrálható megoldást kínált.

A jelenlegi projekt során azonban saját backend fejlesztése mellett döntöttünk, amelyet a Spring Boot keretrendszer segítségével valósítottunk meg. Ez lehetővé teszi az üzleti logika rugalmasságát, a komplex funkcionalitások beépítését. Az adatbázist egy PostgreSQL adatbázis biztosítja, amely a megbízhatóságáról, a magas teljesítményéről és a skálázhatóságról ismert.

Bár áttértünk egy saját backend és adatbázis használatára, a Firebase-t továbbra is megtartottuk bizonyos funkciókhoz. Elsősorban a felhasználók hitelesítését segít megoldani, valamint az értesítések kezelését segíti, mivel ezekre a szolgáltatásokra a Firebase egyszerű és megbízható megoldást kínál.



4.1. ábra. Architektúra

A 4.1 ábrán látható rendszer architektúrája három fő részből áll: a frontend-ből, a backend-ből és az adatbázisból. A következő komponensek és azok kapcsolatai alkotják a rendszert:

1. Felhasználó (User): A rendszer elsődleges használója, aki a React Native alkalmazáson keresztül interakcióba lép a backenddel. A felhasználói interakciókat a frontend oldalon kezeli a React Native alkalmazás.

2. React Native: A frontend komponens, amely biztosítja a mobilalkalmazás felhasználói felületét és kommunikációját a backenddel. Az alkalmazás REST API-n keresztül kapcsolódik a backendhez, így biztosítva az adatcserét a felhasználói interakciók és a backend között. Ezen kívül a Firebase SDK-t használja a hitelesítéshez és az értesítésekhez.

3. Firebase: A Firebase szolgáltatásokat két fő célra használjuk:

   - Felhasználói hitelesítés: A Firebase Authentication biztosítja a felhasználók egyszerű és biztonságos bejelentkezését.
   - Értesítések: A Firebase Cloud Messaging (FCM) segítségével küldünk push értesítéseket a felhasználóknak.

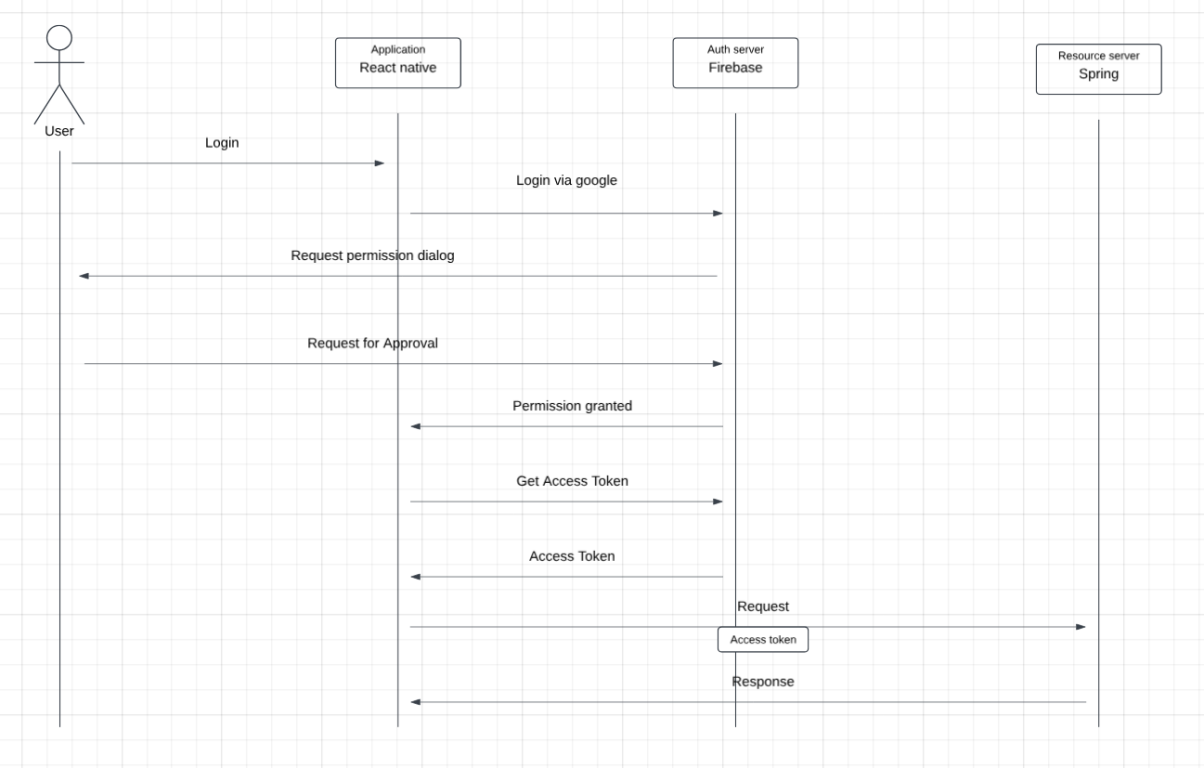
4. Spring Boot (Backend): A backend komponens a Spring Boot keretrendszerrel valósult meg, és felelős az üzleti logikáért, az adatok feldolgozásáért, valamint a REST

API biztosításáért, amelyet a frontend használ. A backend az API-n keresztül kommunikál a React Native alkalmazással és a PostgreSQL adatbázissal is.

5. PostgreSQL adatbázis: Az adatbázis tárolja a felhasználói adatokat és a rendszer által kezelt információkat. A Spring Boot backend a JDBC vagy ORM (Object-Relational Mapping) interfészen keresztül kapcsolódik a PostgreSQL adatbázishoz, így biztosítva az adatok kezelését és tárolását.

6. Docker: Az infrastruktúra részeként Docker konténert használunk a backend és az adatbázis futtatására. A Docker lehetővé teszi a környezet könnyű kezelhetőségét és skálázhatóságát, így az alkalmazás könnyen deployálható és karbantartható.

4.2. Autentikáció



4.2. ábra. Autentikáció

A szerepkörök az autentikáció során:

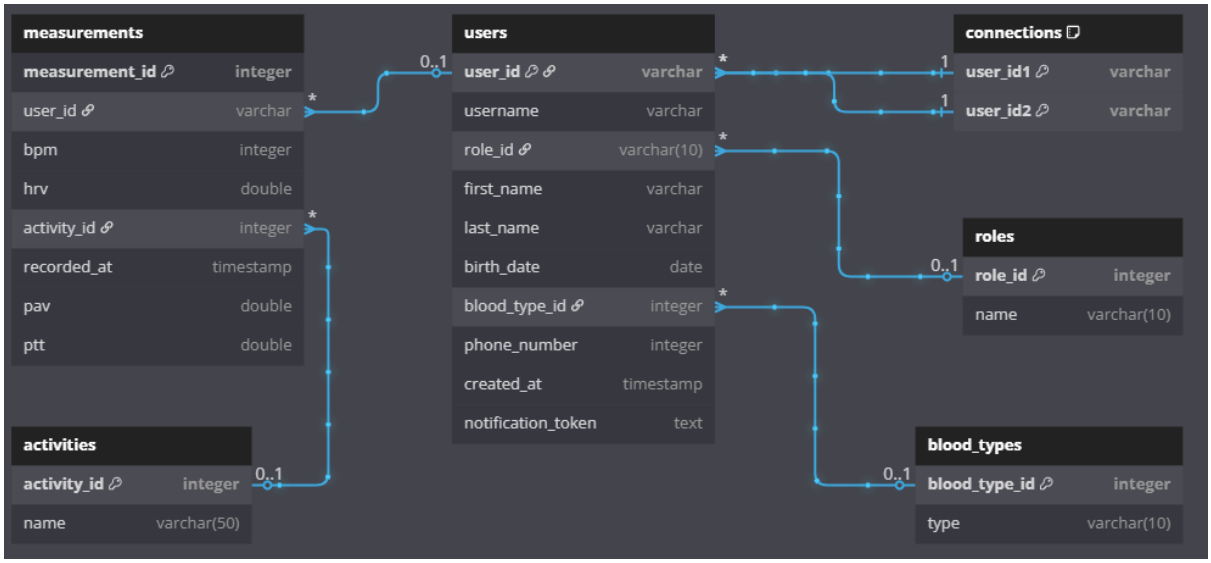
- Resource owner - A felhasználó (User)
- Resource server - Spring backend
- Client - React Native
- Auth Server - Firebase

Ez a diagram (4.2) az alkalmazás hitelesítési folyamatát ábrázolja, amely Google bejelentkezésen keresztül történik:

1. Felhasználó (User):
    - A felhasználó elindítja a bejelentkezési folyamatot az alkalmazásban a „Login” opcióval.
2. Alkalmazás (React Native):
    - Miután a felhasználó a Google-en keresztül szeretne bejelentkezni, az alkalmazás egy Google bejelentkezési párbeszédpanelt kér (a Request permission dialog lépés), amely lehetővé teszi számára, hogy engedélyezze a Google fiók használatát az alkalmazásba való belépéshez.
    - Az alkalmazás a felhasználó jóváhagyását kéri a hitelesítési folyamat elindításához (Request for Approval).
3. Firebase Auth Server (Firebase):
    - Ha a felhasználó jóváhagyja a kérést (kiválaszt egy email fiókot), akkor a Firebase hitelesítési szerver (Auth server) elküldi az applikációnak az engedélyezést (Permission Granted).
    - Ezután a Firebase Auth szerver egy hozzáférési tokent (Access token) generál, amely az alkalmazás számára biztosítja a hozzáférést a felhasználó adataihoz.
4. Alkalmazás (React Native):
    - Az alkalmazás megkapja a hozzáférési tokent a Firebase Auth szervertől. Ez a token a felhasználó hitelesítési információit tartalmazza, és biztosítja, hogy a későbbi kéréseket az alkalmazás a felhasználó nevében indíthassa.
5. Erőforrás Szerver (Spring):
    - Az alkalmazás összes endpoint-ja le van védve, minden endpoint-ra érkező kérés fejlécéből a tokent kiveszi, és a Firebase hitelesíti, csak hitelesített, érvényes tokennel rendelkező kéréseket szolgál ki a backend.

4.3. Adatbázis terv

A Spring backend mellé mindenképpen egy relációs adatbázist kellett válasszunk, mivel a relációs adatbázisok jól illeszkednek a Spring JPA által biztosított objektum-relációs-leképzési (ORM) lehetőségekhez. A PostgreSQL mellett döntöttünk, mert nyílt forráskódú, rendkívül stabil és jól skálázható.



4.3. ábra. Adatbázis szerkezet

4.3.1. Adatbázis terv leírása

Az adatbázisban az alábbi táblák találhatók:

1. Felhasználók (users):
    - Ez a tábla a felhasználók adatait tárolja, és a hozzájuk kapcsolódó információkat rögzíti. A felhasználók egyedi azonosítója (id) a Firebase szolgáltatásból származik.
2. Mérések (measurements)
    - Ebben a táblában tárolódnak a mérési adatok, amiből egy, egy felhasználóhoz tartozik, itt megvalósul egy egy az többhöz kapcsolat.
3. Tevékenységek (activities)
    - A méréshez egy tevékenység (activity) tartozik, amely leírja, hogy a mérés során a személy éppen milyen tevékenységet végzett. Minden méréshez egy konkrét activity kapcsolódik, például pihenés, sport, munka stb., így minden méréshez rögzíthető, hogy a mérés pontosan melyik tevékenység alatt történt.
4. Vércsoport (blood_types)
    -

4.4. Gitlab

4.4.1. Feladatkezelés és Tervezés

A tervezési szakaszban a GitLab Issues funkcióit használtuk, egy külön repository-t hoztunk létre az issue-k kezelésére:

- Issues: A projekt feladatait issuekra bontottuk, amelyek tartalmazzák:
  - Cím: A feladat rövid összefoglalása.
  - Leírás: A feladat részletes bemutatása, például a végrehajtandó kódmódosítások vagy az elvárt eredmények.
  - Címkék (Labels): Kategorizálják a feladatokat (pl. backend, bug, feature).
  - Assignee: A felelős csapattag kijelölése.

4.4.2. Verziókezelés

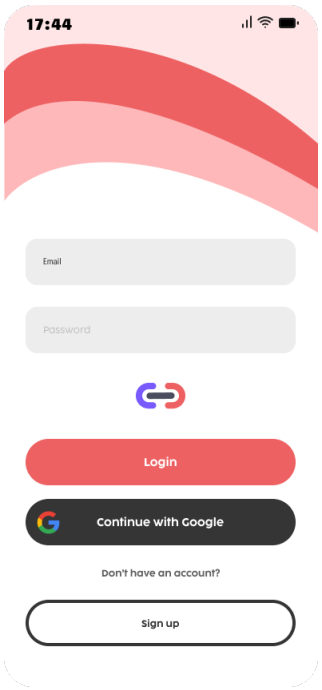
A GitLab verziókezelésének segítségével nyomon követjük a kód módosításait:

- Branching stratégia: A main ág tartalmazza a stabil kódot, míg a különböző featureök fejlesztése saját ágakon történik.
- Merge Requestek (MR-ek): A kód áttekintése és jóváhagyása:
  - Kód átnézése (Code Review) kötelező, mielőtt egy merge request-et jóváhagyunk és a main ágba összevonjuk.
  - Automatikus tesztek futtatása a CI/CD pipeline segítségével.

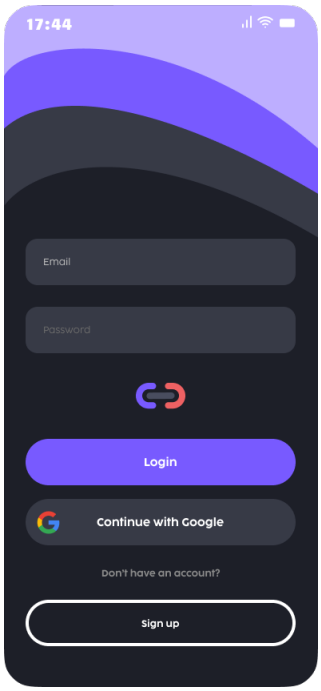
4.5. UI Terv

A UI-nak kétféle kinézetet terveztünk, egy világos témát (light theme) és egy sötét témát (dark theme). A téma színe automatikusan alkalmazkodik a telefonnak a gyárilag beállított témájához. A tervezéshez Figma-t használtunk.

4.5.1. Login



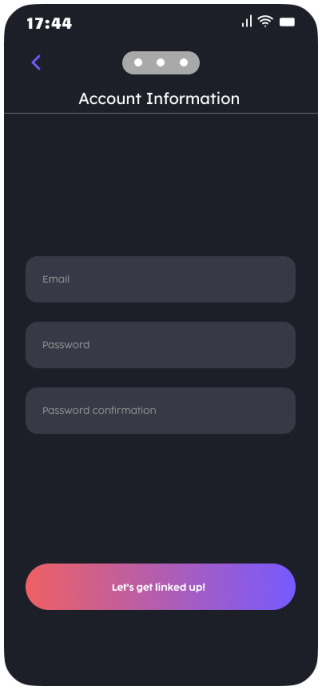
4.4. ábra. Bejelentkezés



4.5. ábra. Bejelentkezés (Sötét)

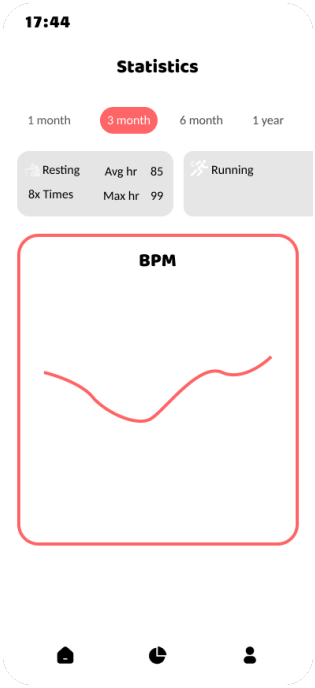
4.5.2. Register

4.6. ábra. Regisztráció

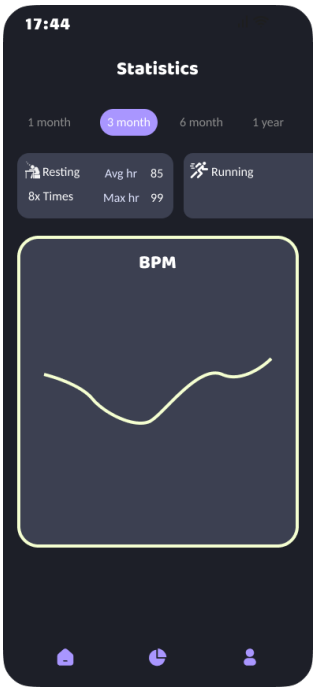


4.7. ábra. Regisztráció (Sötét)

4.5.3. Tracking

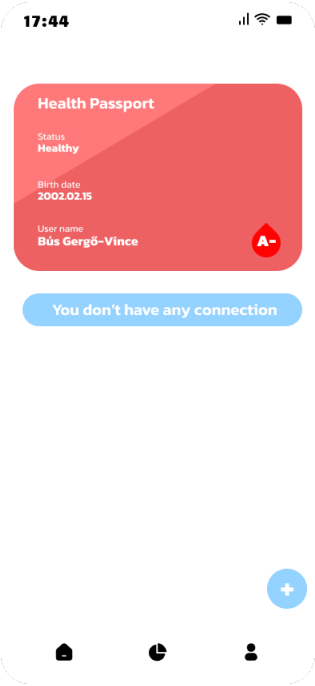


4.8. ábra. Statisztikák

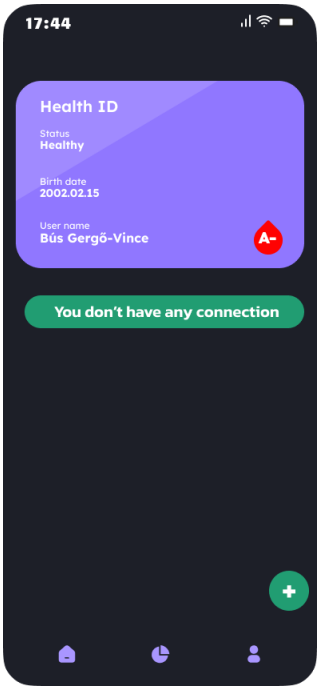


4.9. ábra. Statisztikák (Sötét)

4.5.4. Profile



4.10. ábra. Profil



4.11. ábra. Profil (Sötét)

5. fejezet

Felhasznált technológiák

Az alkalmazás három fő részből tevődik össze: a kliensalkalmazásból (frontend), a szerverből (backend) és az adatbázisból.

5.1. Frontend

- **React Native** – Keresztplatformos mobilalkalmazás-fejlesztési kerektrendszer, amely lehetővé teszi a natív IOS és Android alkalmazások fejlesztését.
- **Expo** – Nyílt forráskódu keretrendszer React Native fejlesztéshez, amely megkönnyíti az alkalmazások építését, tesztelését és telepítését.
- **TypeScript** – Típusbiztonság és jobb kódbázis-karbantarthatóság érdekében.

5.2. Backend

- **Java Spring Boot** – Könnyen konfigurálható keretrenedszer robosztus és skálázható szerveroldali alkalmazások fejlesztéséhez.
- **Spring Data JPA** – Absztrakciós réteget biztosít a JPA fölött, leegyszerűsítve az adatbázis-műveleteket. Automatikus lekérdezésgenerálást, tranzakciókezelést és integrációt kínál különböző perzisztenciarétegekkel.
- **Gradle** – Build és függőségkezelés

5.3. Adatbázis

- **PostgreSQL** – Megbízható, skálázható relációs adatbázis.

5.4. Infrastruktúra & Deployment

- **Docker** – Konténerizált futtatókörnyezet a backend és adatbázis számára
- **GitLab CI/CD** – Automata tesztelés

5.5. Egyéb

- **Firebase** – Hitelesítés és push értesítések

5.6. Felhasznált könyvtárak

- **react-native-vision-camera** – Nagy teljesítményű kamera API, amely lehetővé teszi a valós idejű képfeldolgozást és a kamera közvetlen elérését, például a pulzusméréshez szükséges zöld csatorna intenzitásának kiolvasásához.
- **react-native-gifted-charts** – Interaktív és testreszabható diagramok megjelenítésére szolgáló könyvtár, amelyet az alkalmazás pulzusdiagramjainak vizualizálásához használunk.
- **react-native-svg** – SVG grafikák megjelenítéséhez és manipulálásához használt könyvtár, amelyet egyedi vizuális elemek, például egyéni ikonok és grafikák rajzolására használunk.

6. fejezet

Megvalósítás

6.1. Adatbázis

Az adatok kezelését egy PostgreSQL adatbázis biztosítja, amely az adatok tárolásáért és kezeléséért felel. Az adatbázis docker container-ben fut.

A rendszer az alábbi főbb táblákat tartalmazza: mérések (measurements), felhasználók (users), címek (addresses) és a kapcsolatok (connections).

A users tábla tartalmaz egy felhasználóhoz tartozó adatokat, alapvető információkat. A measurements tábla egy a többhöz kapcsolatban (1:N) van a users táblával, mivel egy mérés egy felhasználóhoz tartozik, és egy felhasználónak lehet több mérése is.

Az activities, roles és blood_types táblák backend szintű enum-ként vannak tárolva, mivel ezek az adatok statikusak, előre definiált értékeket tartalmaznak.

6.2. Adatbázis integráció a szerveroldalon

6.2.1. Liquibase

A liquibase-et az adatbázis állapotának verziókezelésére és migrációk kezelésére használjuk. A lényege, hogy az adatbázis szerkezeti változásait nyomon kövessük és könnyen kezeljük egyes változások között az átmeneteket. A migrációkat YAML formátumban, SQL szkriptek segítségével definiálunk. A fő változásnapló fájl, a db.changelog-master.yaml (6.1 kódrészlet), tartalmazza az összes migrációs lépést, amelyek között megtalálhatóak táblák, oszlopok módosítása, létrehozása, törlése.

A rendszer indításakor a liquibase automatikusan végrehajtja az új változtatásokat az adatbázison. Ezzel egyszerűsítve a fejlesztési folyamatot, mivel minden szükséges migráció automatikusan végrehajtódik, ha szükséges.

6.1. kódrészlet. db.changelog-master.yaml

```yaml
databaseChangeLog:
  - changeSet:
      id: 1
      author: orbanbotond2002
      changes:
        - sqlFile:
            path:
    001_main_schema.sql

  - changeSet:
      id: 2
      author: orbanbotond2002
      changes:
        - sqlFile:
            path:
    002_add_notification_token.sql

  - changeSet:
      id: 3
      author: orbanbotond2002
      changes:
        - sqlFile:
            path:
    003_notificationToken_rename.sql

  - changeSet:
      id: 6
      author: orbanbotond2002
      changes:
        - sqlFile:
            path:
    006_make_bigger_notification.sql
```

6.2. kódrészlet. 001_main_schema.sql

```sql
CREATE TABLE users (
    user_id VARCHAR PRIMARY KEY,
    username VARCHAR NOT NULL,
    role INTEGER REFERENCES
        roles(role_id),
    first_name VARCHAR,
    last_name VARCHAR,
    birth_date DATE,
    blood_type INTEGER REFERENCES
        bloodtypes(bloodtype_id),
    phone_number INTEGER,
    created_at TIMESTAMP DEFAULT
        CURRENT_TIMESTAMP
);

CREATE TABLE activities (
    activity_id SERIAL PRIMARY KEY,
    name VARCHAR NOT NULL
);

CREATE TABLE measurements (
    measurement_id SERIAL PRIMARY KEY,
    user_id VARCHAR REFERENCES
        users(user_id),
    bpm INTEGER,
    hrv FLOAT,
    activity INTEGER REFERENCES
        activities(activity_id),
    recorded_at TIMESTAMP DEFAULT
        CURRENT_TIMESTAMP
);
```

6.2.2. JDBC az adatbázis kapcsolódásához

Az alkalmazásban a JDBC kapcsolatot (Java Database Connectivity) a Spring Data JPA-n keresztül valósítottuk meg, ez teszi lehetővé, hogy a backend könnyedén kommunikáljon a PostgreSQL adatbázissal. A JDBC beállítások és az adatbázis elérési adatai az `application.properties` (6.11 kódrészlet) fájlba kerülnek definiálásra. Az adatbazis URL-je, felhasználóneve és jelszava környezeti változókként vannak tárolva, a biztonságos hozzáférés érdekében.

6.3. kódrészlet. `application.properties`

```
spring.application.name=Vitalink_backend
spring.datasource.username=${DB_USERNAME}
spring.datasource.password=${DB_PASSWORD}
spring.datasource.url=${DB_URL}
spring.datasource.driver-class-name=org.postgresql.Driver
spring.liquibase.change-log=/db/changelog/db.changelog-master.yaml
firebase.url=https://pulse-app.firebaseio.com
spring.jackson.serialization.fail-on-empty-beans=false
```

6.3. Firebase Integráció

A Firebase integrációra backend szinten azért van szükségünk, hogy tudjuk hitelesíteni a felhasználókat, akik a kliens alkalmazáson keresztül bejelentkeztek. Valamint értesítések küldését is a Firebase szolgáltatja.

A Firebase inicializálása egy Spring *@Configuration* osztályban történik, ahol a Firebase szolgáltatások, mint a FirebaseApp és a FirebaseMessaging, beállításra kerülnek. Az alábbi kódot használjuk a Firebase inicializálásához: FirebaseInitialization (6.4 kódrészlet).

```
1  @Configuration
2  public class FirebaseInitialization {
3
4      @Value("${firebase.url}")
5      private String firebaseUrl;
6
7      @Bean
8      public FirebaseApp firebaseApp() {
9          try {
10             if(FirebaseApp.getApps().isEmpty()) {
11                 String serviceAccountPath =
                       System.getenv("FIREBASE_SERVICE_ACCOUNT_KEY_PATH");
12                 FileInputStream serviceAccount =
13                         new FileInputStream(serviceAccountPath);
14
15                 FirebaseOptions options = new FirebaseOptions.Builder()
16                         .setCredentials(GoogleCredentials.
17                                         fromStream(serviceAccount))
18                         .setDatabaseUrl(firebaseUrl)
19                         .build();
20                 return FirebaseApp.initializeApp(options);
21             } else {
22                 return FirebaseApp.getInstance();
23             }
24         } catch (IOException e) {
25             throw new
                   RuntimeException(ErrorMessages.FAILED_TO_INITIALIZE_FIREBASE_APP, e);
26         }
27     }
28
```

```
    @Bean
    public FirebaseMessaging firebaseMessaging(FirebaseApp firebaseApp) {
        return FirebaseMessaging.getInstance(firebaseApp);
    }
}
```

6.4. kódrészlet. Firebase inicializálása

Itt is jól látható, hogy a service key mivel érzékeny információkat tartalmaz, ezért környezeti változóként van a projekthez hozzáadva.

```
{
  "type": "service_account",
  "project_id": "pulse-app-6f8eb",
  "private_key_id": "<private-key-id>",
  "private_key": "<private-key>",
  "client_email": "<client-email>",
  "client_id": "<client-id>",
  "auth_uri": "https://accounts.google.com/o/oauth2/auth",
  "token_uri": "https://oauth2.googleapis.com/token",
  "auth_provider_x509_cert_url":
  "https://www.googleapis.com/oauth2/v1/certs",
  "client_x509_cert_url":
  "https://www.googleapis.com/robot/v1/metadata/x509/
    firebase-adminsdk-dwq8t@pulse-app-6f8eb.iam.gserviceaccount.com",
  "universe_domain": "googleapis.com"
}
```

6.5. kódrészlet. Firebase service key

6.4. Biztonság és Hozzáférés-ellenőrzés

A kliensalkalmazás közvetlenül a Firebase-el kommunikál a hitelesítés érdekében. A felhasználó bejelentkezése után a Firebase egy hitelesítési tokent generál, amelyet a kliensalkalmazás minden kéréséhez csatol, amelyek a backend felé történnek.

Az alkalmazás biztonságát a Firebase alapú token-validálás biztosítja. Az endpointok védettek, mivel csak érvényes Firebase tokennel rendelkező felhasználók férhetnek hozzá. A filter ellenőrzi a „Bearer” típusú Authorization fejlécet, ha a token érvénytelen vagy lejárt, elutasítja a kérést. Ez a szűrő minden beérkező HTTP-kérést ellenőriz.

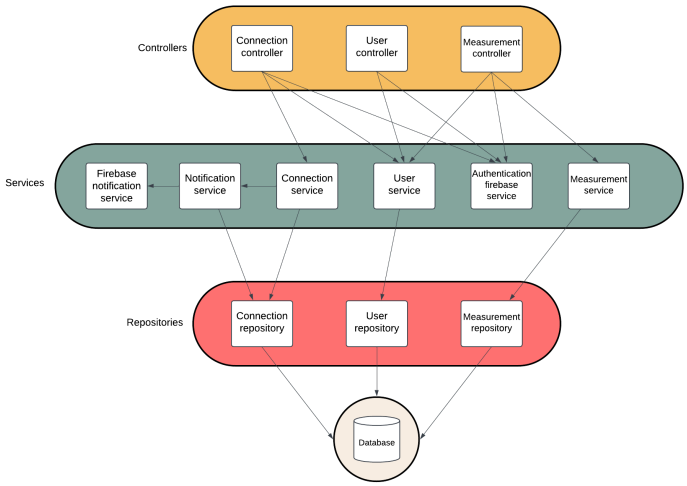
```
@Override
    protected void doFilterInternal(HttpServletRequest request, HttpServletResponse
        response,
            FilterChain filterChain) throws ServletException, IOException {

        String authorizationHeader = request.getHeader("Authorization");

        if (authorizationHeader != null && authorizationHeader.startsWith("Bearer ")) {
            String token = authorizationHeader.substring(7);

            try {
                FirebaseAuth.getInstance().verifyIdToken(token);
            } catch (Exception e) {
```

```
13                response.setStatus(HttpServletResponse.SC_UNAUTHORIZED);
14                response.getWriter().write(ErrorMessages.INVALID_OR_EXPIRED_TOKEN);
15                return;
16            }
17        } else {
18            response.setStatus(HttpServletResponse.SC_UNAUTHORIZED);
19            response.getWriter().write(ErrorMessages.MISSING_OR_BAD_AUTH_HEADER);
20            return;
21        }
22        filterChain.doFilter(request, response);
23    }
```

6.6. kódrészlet. Token validálás

6.5. Backend szerkezete

A szerver szerkezetileg egy négy rétegű architektúrát valósít meg. A négy réteg: controller, service, repository, adatbázis. Ez az architektúra számos előnnyel jár, különösen összetettebb rendszerek esetén, mivel a különböző rétegek önállóan is kezelhetőek. A struktúra továbbá lehetővé teszi a rendszer bővíthetőségét is, mivel új funkciók hozzáadása nem igényel jelentős változtatásokat a teljes rendszeren. A tesztelhetőséget is megkönnyíti, mivel minden réteget külön-külön lehet tesztelni. A kód újrafelhasználhatósága szintén egy előny, mivel egyes rétegek módosítását megkönnyíthetik az alsó rétegek.



6.1. ábra. A backend rétegezett architektúrája: Controller, Service, Repository és Database.

6.5.1. Rétegek

Controller réteg

Ennek a rétegnek a feladata, hogy kezelje a beérkező kéréseket, az API végpontokért és a kliensalkalmazással való kommunikációért. Kommunikál a service réteggel, majd a kapott választ visszaküldi a kliensnek.

- *UserController* - Felhasználói műveleteket kezel.

  A user controllerben a következő végpontokat valósitja meg: register (6.7 kódrészlet), getUser, logout, login.

  A felhasználók azonosítóját a fejlécből, tokenből nyerjük ki, és a felhasználó azonosítójával hozunk létre új felhasználót, illetve a többi végpont felhasználó entitást adja át a service rétegnek, amit a tokenből határozunk meg. A login és logout műveletekre azért van szükség a Firebase autentikáció mellett, mert a bejelentkezett felhasználóknak a notification token-jét is eltároljuk. Ha egy felhasználó bejelentkezik és elfogadta, hogy értesítéseket is kaphat, akkor a token elmentésre kerül. Amikor a felhasználó kijelentkezik, akkor a token törlődik.

```
@PostMapping("/register")
    public ResponseEntity<User> registerUser(@RequestHeader("Authorization")
        String authorizationHeader, @RequestBody User user) {
        try{
            String userId =
                authFirebaseService.getUserIdFromToken(authorizationHeader);
            User savedUser = userService.saveUserWithId(user, userId);
            return ResponseEntity.status(HttpStatus.CREATED).body(savedUser);
        } catch (Exception e) {
            return ResponseEntity.status(HttpStatus.BAD_REQUEST).body(null);
        }
    }
```

6.7. kódrészlet. User controller, register endpoint

- *ConnectionController* - Felhasználók közötti kapcsolatokat kezel.

  Connection controllerben a kapcsolatok létrehozásával, lekérdezésével és torlésével kapcsolatos endpointok kerültek implementálásra (6.9 kódrészlet).

  Ebben a controller rétegben is a felhasználó azonosítóját a fejléc tokenéből nyerjük ki.

```
@PostMapping("/create")
    public ResponseEntity<Void> createConnection(@RequestHeader("Authorization")
        final String authorizationHeader, @RequestBody final Map<String, String>
        body) {
        try{
            final String requestingUserId =
                authFirebaseService.getUserIdFromToken(authorizationHeader);
            final User requestingUser =
                userService.getUserById(requestingUserId);
            final String targetUserEmail = body.get("email");
            final String targetUserId =
                userService.getUserIdFromEmail(targetUserEmail);
            final User targetUser = userService.getUserById(targetUserId);

            connectionService.createConnection(requestingUser, targetUser);

            return ResponseEntity.ok().build();
        } catch (Exception e) {
            return ResponseEntity.status(HttpStatus.BAD_REQUEST).body(null);
```

```
        }
    }
```

6.8. kódrészlet. Connection controller, create endpoint

- MeasurementController - Méréseket kezel.

  A measurement controllerben 4 darab endpoint van leimplementálva: specifikus mérés lekérdezése, felhasználóhoz tartozó mérések lekérdezése, mérés mentése és időintervallumbeli mérések lekérdezése.

  - *getMeasurement*: Egy adott mérés adatait adja vissza. Az URL-ből kinyeri a mérés azonosítóját, majd a service-nek átadja azt.
  - *getMeasurementsByUser*: Egy adott felhasználóhoz tartozó összes mérést kéri le. Ugyancsak URL változóból olvassa ki a felhasználó azonosítóját, majd meghívja a megfelelő service metódust az adatok lekérdezésére.
  - *save*: A felhasználók meghatározza, a Bearer token segítségével. A felhasználót és a mérést átadja a service-nek.
  - *getMeasurementsByUserAndInterval*: A getMeasurementsByUser metódushoz hasonlóan egy adott felhasználó méréseit kéri le, azonban egy megadott időintervallumon belül.

```
@GetMapping("/users/{id}/interval")
    public ResponseEntity<List<Measurement>> getMeasurementsByUserAndInterval(
            @PathVariable String id,
            @RequestParam("start") String start,
            @RequestParam("end") String end ) {
        List<Measurement> measurements =
            measurementService.getMeasurementsByUserAndInterval(id,
            Instant.parse(start), Instant.parse(end));
        if (measurements.isEmpty()) {
            return ResponseEntity.noContent().build();
        }
        return ResponseEntity.ok(measurements);
    }
```

6.9. kódrészlet. Connection controller, create endpoint

Service réteg

A service réteg az üzleti logikát valósítja meg. Feladata, hogy a controller-től érkező kéréseket feldolgozza, és szükség esetén a megfelelő adatokat továbbítsa az adattárolásért felelős réteghez (repository). Ezek az osztályok a @Service annotációval vannak ellátva.

Ebben a rétegben a következők vannak leimplementálva: felhasználók kezelése: *UserService*, felhasználók közötti kapcsolatok irányítása: *ConnectionService*, mérések kezelése: *MeasurementService*, AuthFirebaseService, FirebaseNotificationService, NotificationService

A *UserService*, *ConnectionService*, *MeasurementService* osztályok a megfelelő objektumokat kezelik és biztosítják az üzleti logikát, továbbítják az adatokat a repository és a controller rétegek között.

Az *AuthFirebaseService* osztály fő feladata a felhasználói azonosító (UID) kinyerése a tokenből.

A *FirebaseNotificationService* osztály felelős az értesítések küldéséért, amit a Firebase Cloud Messaging (FCM) használatával valósítottunk meg. Egyetlen és fő metódusa sendNotificationToUser (6.10 ábra), ami lekéri a felhasználóhoz tartozó értesítési tokent, ezután létrehoz egy értesítési üzenetet, amihez egy címet és törzs szöveget rendel, amit majd a kliens oldali alkalmazás megkap.

```
1  public void sendNotificationToUser(User user, String title, String body) {
2          String token = user.getNotificationToken();
3          if (token != null) {
4              Map<String, String> notificationData = new HashMap<>();
5              notificationData.put("title", title);
6              notificationData.put("body", body);
7
8              Message message = Message.builder()
9                      .setToken(token)
10                     .putAllData(notificationData)
11                     .build();
12
13             try {
14                 firebaseMessaging.send(message);
15             } catch (FirebaseMessagingException e) {
16                 throw new RuntimeException(ErrorMessages.FAILED_TO_SEND_NOTIFICATION,
                       e);
17             }
18         }
19     }
```

6.10. kódrészlet. sendNotificationToUser

A NotificationService a FirebaseNotificationService segítségével küld értesítést egy adott felhasználónak, vagy a kapcsolattartó személyeknek.

Repository réteg

A repository réteg a Spring Data JPA segítségével biztosítja az adatbázissal való interakciót. Biztosítja az adatok mentését, frissítését, törlését, lekérdezését az adatbázisból.

A *ConnectionReposity* a kapcsolatok kezeléséért és lekérdezéséért felelős két felhasználó között.

MeasurementRepository osztály a felhasználók mérési objektumait kezeli.

A *UserRepository* a felhasználók adatainak kezeléséért felelős.

Model réteg

A model réteg a backend architektúrájának egyik alapvető komponense. A modellek az adatbázis tábláit reprezentálják. A model osztályok JPA entitások. Ezek az osztá-

lyok felelnek az adatok tárolásáért és kezeléséért, a repository réteg segítségével kerülnek mentésre, törlésre, módosításra vagy lekérdezésre.

6.5.2. Backend kitelepítése

A szervert és az adatbázist a céges Kubernetes klaszterbe telepítettük ki. A Rancher platformon keresztül érhettük el a klasztert, ahol saját felhasználókat kaptunk. Dedikált névtér készült a mi projektünknek, amihez nekünk volt hozzáférésünk: `vitalink-ns`

Kitelepítés

A backend image-eket az `r.edu.codespring.ro` konténer image tárhelyen tároljuk. A Kubernetes innen fogja lehúzni a képet futtatáskor.

Elsősorban szükség volt GitLabon létrehozni egy tokent, aminek a felhasználásával a Kubernetes jogosultságot kap arra, hogy a container registry-ből sikeresen le tudja húzni a backend image-t. Azt a secretet, ami ezt a tokent tartalmazza, a következő paranccsal hoztam létre:

```
kubectl create secret docker-registry registry-credential \
  --docker-server=r.edu.codespring.ro \
  --docker-username=gitlab-ci-token \
  --docker-password=<token>
```

A backend alkalmazás Firebase integrációjához szükséges a `firebaseServiceKey.json` fájl, amely egy hitelesítési kulcsot tartalmaz. Mivel ez is érzékeny adat, nem tárolható a forráskódban. Ennek biztonságos kezelése Kubernetes Secret erőforrás segítségével történt. A következő parancs segítségével hoztuk létre a secretet, ami a kulcsot tartalmazza:

```
kubectl create secret generic firebase-secret \
  --from-file=firebaseServiceKey.json=../firebaseServiceKey.json
```

A *db-secret.yaml* (6.13 ábra) egy titkos adatokat tartalmazó fájl, amelyben érzékeny információk találhatóak, a PostgreSQL adatbázis felhasználóneve és jelszava van tárolva base64-ben kódolva. A típushoz (kind) adom meg, hogy egy secretet akarok létrehozni, a metadata résznél adom meg a secret tulajdonságait, a nevét és a névterét. A típusa Opaque, ami azt jelenti, hogy az adatokat titkosított formában tárolja. A data mezőben kulcs-érték párral tudom megadni a titkos adataimat, ami az adatbázis csatlakozásához a felhasználónév és jelszó.

A backend pod-ja (6.12 kódrészlet), ami tartalmazza az összes szükséges konfigurációt a szerver működéséhez. Az elején a Pod jelzi, hogy a Kubernetes objektum típusa Pod, amely a legkisebb egység a Kubernetes rendszerben. A metadata kulcsszó alatt vannak megadva a Pod általános információi, a neve, milyen névtérben van és az alkalmazás címkéje. Az „imagePullSecrets”-hez meg kell adni azt a secretet, ami tartalmazza a megfelelő tokent, amivel a pod jogosultságot kap az adott image letöltéséhez a Gitlab container registry-ből. A „containers” alatt definiáljuk a konténert, amelyet a pod fog futtatni. Az image-et a Gitlabnak a container registry-jéből fogja leszedni, az image így néz ki:

r.edu.codespring.ro/heart-rate-2024/hrt-backend:<VERSION>. A konténernek be kell állítani a környezeti változóit is, az adatbázishoz a felhasználónév és jelszót a secret-ből veszi ki, valamint a csatlakozási URL-t helyben adtuk meg. A firebase-secret tartalma fájlrendszerként csatolódik hozzá.

6.11. kódrészlet. `application.properties`

```
spring.application.name=Vitalink_backend
spring.datasource.username=${DB_USERNAME}
spring.datasource.password=${DB_PASSWORD}
spring.datasource.url=${DB_URL}
spring.datasource.driver-class-name=org.postgresql.Driver
spring.liquibase.change-log=/db/changelog/db.changelog-master.yaml
firebase.url=https://pulse-app.firebaseio.com
spring.jackson.serialization.fail-on-empty-beans=false
```

Az adatbázist az alábbi parancs segítségével hoztuk létre. A parancs telepíti a Bitnami PostgreSQL Helm Chart-ot a `vitalink-ns` Kubernetes névtérbe, aminek adtunk meg egyéni konfigurációt (6.14 kódrészlet).

6.12. kódrészlet. hrt-backdend-pod.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: hrt-backend
  labels:
    app: hrt-backend
  namespace: vitalink-ns
spec:
  imagePullSecrets:
    - name: registry-credential
  containers:
    - name: hrt-backend
      image: <IMAGE_URL>
      ports:
        - containerPort: 8080
      env:
        - name: DB_PASSWORD
          valueFrom:
            secretKeyRef:
              name: db-secret
              key: password
        - name: DB_USERNAME
          valueFrom:
            secretKeyRef:
              name: db-secret
              key: username
        - name: DB_URL
          value: "DB_URL"
        - name:
FIREBASE_SERVICE_ACCOUNT_KEY_PATH
          value:
/app/config/firebaseServiceKey.json
      volumeMounts:
        - name: firebase-key-volume
          mountPath: /app/config
          readOnly: true
  volumes:
    - name: firebase-key-volume
      secret:
        secretName: firebase-secret
```

6.13. kódrészlet. db-secret.yaml

```
apiVersion: v1
kind: Secret
metadata:
  name: db-secret
  namespace: vitalink-ns
type: Opaque
data:
  username: cG9zdGdyZXM=
  password: ZGV2ZWxvcGVy
```

6.14. kódrészlet. values.yam

```
architecture: "replication"

auth:
  user: "developer"
  postgresPassword: "developer"
  password: "developer"
  replicationPassword:
      "repDeveloper"

passwordUpdateJob:
  enabled: true
```

```
helm install my-postgres\
    bitnami/postgresql \
    -f values.yaml -n vitalink-ns
```

6.2. ábra. parancs és fájl az adatbázis létrehozásához

Az adatbázis külön pod-ba kerül. Pontosabban kettő pod hozódik létre az adatbázisnak, egy primary pod és egy read pod. Az elsődleges pod (primary) ez a fő adatbázis példány, írható és olvasható, itt történnek meg az adatbázis műveletek, például az adatok beírása és módosítása. A read pod példány csak olvasható, nem írható. Ez egy replika, ami folyamatosan szinkronizálódik a primary pod-dal, hogy lekérdezéseket végezhessen az adatbázison. Terheléselosztás céljából van használatban, hogy a rendszer képes legyen nagyobb olvasási terheléseket kezelni, eközben az írásokat csak a primary pod kezeli.

6.15. kódrészlet. hrt-backend-service.yaml

```yaml
apiVersion: v1
kind: Service
metadata:
  name: hrt-backend-service
  namespace: vitalink-ns
spec:
  type: ClusterIP
  selector:
    app: hrt-backend
  ports:
    - port: 8080
      targetPort: 8080
```

6.16. kódrészlet. hrt-backend-ingress.yaml

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: hrt-backend-ingress
  namespace: vitalink-ns
  annotations:
    cert-manager.io/cluster-issuer:
        letsencrypt-prod
    traefik.ingress.
    kubernetes.io/rewrite-target: /
spec:
  ingressClassName: traefik
  rules:
    - host:
        vitalink.k8s.edu.codespring.ro
      http:
        paths:
          - path: /
            pathType: Prefix
            backend:
              service:
                name: hrt-backend-service
                port:
                  number: 8080
  tls:
    - hosts:
        - "vitalink.k8s.edu.codespring.ro"
      secretName: letsencrypt-prod
```

A hrt-backend-service (6.15 kódrészlet) egy Kubernetes Service erőforrás, ami a `vitalink-ns` névtérbe került. A service típusa ClusterIP, ami biztosítja a hálózaton belüli kommunikációt. A selector mező határozza meg, hogy melyik podokra irányítsa át a kéréseket. Az összes olyan pod, amely rendelkezik a címkével, automatikusan továbbítódik oda a kérés. A port kommunikációban a `port` a szolgáltatáson belüli elérési port, amit más podok fognak használni a kommunikációhoz. A `targetPort` a célpodon belüli port, amiben az alkalmazás fut.

A hrt-backend-ingress (**??** ábra) Kubernetes Ingress erőforrás, ez is a `vitalink-ns` névtérbe kerül. Lehetővé teszi a HTTPS forgalom irányítását a hrt-backend-service szolgáltatás felé. Ez biztosítja, hogy a backend alkalmazás egy nyilvános URL-en keresztül elérhető legyen. A traefik ingress osztályt használja a forgalomirányításra. A traefik támogatja a dinamikus konfigurációt, HTTPS-t és beépített támogatással rendelkezik a Let's Encrypt tanúsítványkezeléshez. A backend alkalmazás ezen a domainen érhető el: `vitalink.k8s.edu.codespring.ro`. A tls szekció biztosítja, hogy a megadott domainhez HTTPS kapcsolaton keresztül történjen a hozzáférés. A tanúsítványt a cert-manager generálja és a `letsencrypt-prod` nevű secretben tárolja.

6.6. Frontend

A kliensoldali kódbázis fejlesztésére a React Native-et és ezen belül az Expo keretrendszert használtuk. A React Native egy JavaScript-alapú keretrendszer, amely lehetővé teszi a keresztplatformos mobilfejlesztést. Az elsődleges ok, amiért ezt a technológiát választottuk, az, hogy rendkívül nagy közösséggel rendelkezik. Ez számunkra egy nagyon fontos szempont volt, hiszen szükségünk volt arra, hogy minél több csomag álljon rendelkezésünkre a fejlesztés gördülékenysége miatt.

6.6.1. Authentikáció

6.6.2. Szerveroldallal való kapcsolat

Tekintve, hogy a szerveroldali kódbázis REST API formátumot követ, kifejezetten egyszerűen kapcsolódni tudunk hozzá. Ennek megvalósítására az *Axios* csomagot használjuk (6.17), mivel ez egy aszinkron HTTP-kérések kezelésére optimalizált, széles körben elterjedt megoldás.

6.17. kódrészlet. Endpoint meghívása Axios segítségével

```
 getMeasurementById = async (
   token: string,
   measurementId: number
): Promise<Measurement> => {
   const url = `${process.env.URL_BASE}/api/measurements/${measurementId}`;
   try {
      const response = await axios.get<Measurement>(url, {
         headers: {
            Authorization: `Bearer ${token}`,
            'Content-Type': 'application/json',
         },
      });
      return response.data;
   } catch (e) {
      throw new Error(`Failed to fetch measurement: ${e}`);
   }
};
```

A mi esetünkben kifejezetten hasznos az Axios használata, mivel csökkenti a kód méretét és átláthatóbbá válik. Ezen felül, automatikus JSON kezelést végez, nem kell manuálisan átalakítani a válaszul kapott adatokat. Végső soron pedig többféle hibakezelési lehetőség van beleépítve, mint amit a fetch API kínál.

Annak érdekében, hogy sikeresen tudjunk használni egy API-endpointot, szükséges a felhasználó autorizációs tokenjének átadása a fejlécben, hiszen ennek függvényében kaphat hozzáférést a szerveroldalhoz.

6.6.3. Pulzusmérés megvalósítása

Kameramodul beépítése

Az applikáció egyik legérdekesebb és egyben legkomplexebb része, a pulzusmérések megvalósítása. Ennek kivitelezéséhez a **React Native Vision Camera (RNVC)** csomagot használtuk fel, amely lehetőséget adott a kamera által rögzített képkockák individuális feldolgozására. Ezt a megoldást **Frame Processornak** nevezzük.

Első lépésként integrálni kellett az RNVC beépített kamera komponensét. Ehhez szükség volt a telefon kameramoduljának észlelésére és importálására, amelyhez a Vision Camera beépített 'useCameraDevice' függvényét használtuk fel. A felhasználáshoz azonban engedélykérésre van szükség a felhasználótól, ezért a useCameraPermission segítségével (6.18).

6.18. kódrészlet. Kameramodul engedélykérés

```
 const device = useCameraDevice('back');
const { hasPermission, requestPermission } = useCameraPermission();
const [permissionStatus, setPermissionStatus] = useState(
  INITIAL_PERMISSION_STATUS
);
```

6.19. kódrészlet. Android - natív kamera engedélykérés

```
<uses-permission android:name="android.permission.CAMERA"/>
```

Ezek után került sor a kamera komponens integrálására a pulzusmérést kezelő oldalra, amelynek átadódik az előzőekben lekért kameramodul referenciája. Ezenkívül a villanófény is ezen a komponensen belül kezelhető.

Fényintenzitás érzékelése és feldolgozása

A Vitalink alkalmazás szívritmus meghatározási módszere a fotopletizmográfián alapszik, amely az erekben levő vértérfogat változásának megfigyelését jelenti. Annak érdekében, hogy ezt megvalósítsuk, szükséges minden rögzített képkocka feldolgozása.

A feldolgozás menete:

- A frame processzor felhasználásával, minden képkockának elérjük a képpontonkénti adatait amelyek színcsatornákra bontva adódnak vissza.
- Minden képpont zöld csatornájának értékét összesítjük egy változóban.

- A kapott összeget elosztjuk a képpontok mennyiségével, amely eredményezni fogja az adott képkocka intenzitását.

Ez a folyamat matematikailag a következőképpen néz ki:

$$I = \frac{1}{N} \sum_{i=1}^{N} G_i$$

ahol I - intenzitás, N - képpontok mennyisége, G - a zöld csatorna értéke az adott képpontban.

A kiszámolt intenzitásadatokat időrendi sorrendbe helyezve, egy EKG-szerű diagrammot kapunk. Az eredményként kapott adathalmazból azonban elég nehéz lenne meghatározni a valódi szívverést képviselő csúcsokat, ezért itt jön képbe az adatsimítás folyamata.

A Savitzky-Golay algoritmust használtuk ennek a megvalósításához, amelynek használata a 6.20 kódrészletben látható. Az algoritmus működésének megértéséhez szükséges röviden tárgyalni a paramétereket és azok szerepét.

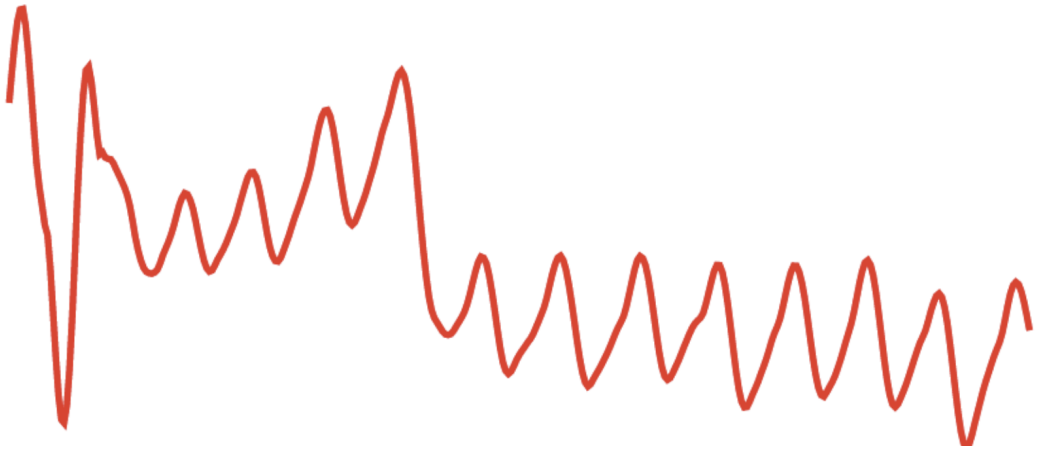
6.20. kódrészlet. Savitzky-Golay algoritmus használata

```
const savGolData = savitzkyGolay(data, 1, {
    windowSize: 21,
    polynomial: 5,
    derivative: 0,
    pad: 'pre',
    padValue: 'replicate',
  });
```

Alapvetően az algoritmus három paramétert vár: az adathalmazt, amelyen dolgoznia kell, az adatpontok közti távolságot és egy *options* objektumot, amelyben a működést szabályozhatjuk. Az *options* paraméter beállításai:

- **windowSize**: Ez a paraméter határozza meg, hogy hány adatponton dolgozik egyszerre az algoritmus.
- **polynomial**: Beállíthatjuk, hogy az algoritmus hányad rendű polinomot illesszen az adatpontokra. Minél magasabb számot adunk meg, annál flexibilisebb lesz a görbe és annál inkább tartja meg a részleteket.
- **derivative**: Ezzel a beállítással határozhatjuk meg, hogy hányad rendű deriváltat szeretnénk visszakapni. A mi esetünkben, az egyszerű simított adatra van szükségünk, ezért 0-át kell átadni az algoritmusnak.
- **pad**: Ez a paraméter az adathalmaz szélein levő adatpontok kezelését állítja. A mi esetünkben, a *pre* beállítás, az adatok meghosszabbítását eredményezi.
- **padValue**: Ez lehetőséget ad arra, hogy eldöntsük, milyen módon szeretnénl az adatok meghosszabbítását, kitöltését. A *replicate* mód azt eredményezi, hogy a legszélső értéket ismétli amíg kitöltődik az adott ablak.

A simítás utáni adatok az ábrán 6.3 láthatóakhoz hasonlóan fognak kinézni:



6.3. ábra. Savitzky-Golay eredménye

Szívritmus percenkénti kiszámolása

Az utolsó fázisban, a percenkénti szívritmust kellett meghatároznunk a rendelkezésre álló adathalmazból.

A valós szívritmust képviselő csúcsok meghatározásához egy egyszerű lokális minimumkeresést hajtunk végre (6.21). Felmerülhet a kérdés, hogy miért minimum és miért nem maximumkeresés. Tekintve, hogy mi a fényvisszaverődést mérjük, és a hemoglobinnak fényelnyelő képessége van, amikor a vérerek megtelnek vérrel, csökken a visszavert fény mennyisége.

A függvény működése a következő: Ha talál egy minimumot, akkor annak az indexét fogja eltárolni. Ez azért szükséges, mert mindegyik index egy képkockát képvisel. Ezáltal, ha ismerjük a felvétel időbeli hosszát, át tudjuk alakítani az értékeket másodpercre, majd percre.

6.21. kódrészlet. Lokális minimum kiszámolása

```
 export function findLocalMinima(n: number, arr: number[]) {
let localMinima = [];

for (let i = 2; i < n - 2; i += 2) {
   if (arr[i - 2] > arr[i] && arr[i] < arr[i + 2]) localMinima.push(i);
}

return localMinima;
 }
```

Utolsó lépésként, a percenkénti szívverést kellett meghatároznunk. Tekintve, hogy az adataink képkockánként voltak elmentve, át kell alakítanunk őket. Ehhez szükség volt megtudnunk, hogy a kameramodul milyen gyorsasággal képes rögzíteni (6.22).

6.22. kódrészlet. FPS és BPM kiszámolása

```
 const fps = data.length / MEASURE_TIME;
 const bpm = Math.floor(calculateBPM(peaks, fps));
```

Most, hogy ismerjük a képkockák számát, a csúcsokat tartalmazó képkockák sorszámát és a felvételi sebességet is, lehetőségünk van kiszámolni a percenkénti szívverést (6.23). A függvény, amit erre használunk, végigiterál a csúcsokon és összeadja az ezek közti távolságokat. A végső soron pedig kiszámolódik a megbecsült BPM, a következő módon:

- **totalDistance / (data.length - 1)**: Megnézi, hogy átlagosan mekkora az eltérés két egymás utáni érték között.
- **/ frameRate**: Átváltja ezt az eltérést időegységre (hány képkockánként történt a változás).
- **/ 60**: Mivel a BPM percenkénti érték, itt másodpercből percre váltunk.
- **1 / (...)**: A változások alapján kiszámolja, hogy milyen gyakori az ismétlődés (hányszor történik meg egy perc alatt).

6.23. kódrészlet. Percenkénti szívverés kiszámolása

```
const calculateBPM = (data: number[], frameRate: number) => {
  const totalDistance = data.reduce(
    (acc: number, value: number, index: number) =>
      acc + Math.abs(value - data[index - 1] || 0),
    0
  );
  return 1 / (totalDistance / (data.length - 1) / frameRate / 60);
};
```

7. fejezet

Továbbfejlesztési lehetőségek

A projekt továbbfejlesztési lehetőségei három fő területre összpontosítanak: AI-alapú mérésfeldolgozás, orvosi szerepkör integrálása és a mérések további pontosítása. Ezek a fejlesztések növelhetik a rendszer megbízhatóságát és felhasználóbarát működését.

7.1. Mesterséges intelligencia fejlesztése

Az AI az alkalmazásban nemcsak a mérésekből határozná meg a szívverést és javítaná a pontosságát, hanem lehetőséget kínál a felhasználók egészségi állapotának részletesebb elemzésére.

- Személyre szabott kiértékelések: Az AI hosszú távon képes lenne tanulni a felhasználó egyéni mintázataiból, így akár pontos egészségügyi előrejelzéseket is adhat.
- AI-alapú döntéstámogatás: Az AI által generált jelentések és javaslatok segíthetnek a felhasználónak az egészségügyi döntések meghozatalában.

7.2. Orvosi szerepkör integrálása

Az alkalmazás egyik fontos továbbfejlesztési lehetősége egy orvosi felület bevezetése.

- Orvosi szakértők bevonása: Az AI által detektált egészségügyi problémák esetén lehetőség nyílik arra, hogy egy szakorvos is megerősítse az esetleges rendellenességeket.
- Távoli konzultáció lehetősége: Az orvosi szerepkör bevonása lehetőséget ad a felhasználónak egészségügyi probléma esetén az online konzultációra, kapcsolattartásra.

7.3. Mérések további pontosítása

A pontos mérési eredmények kulcsfontosságúak az alkalmazás hatékonysága szempontjából.

- Fejlettebb jelfeldolgozás: A PPG jelből származó adatok zajszűrésének és feldolgozásának további fejlesztése, pontosítása és optimalizálása.

- Felhasználóra szabott mérések: Az alkalmazás figyelembe veheti a felhasználók által megadott egyéni egészségügyi adatait, amelyeket felhasznál a mérés pontosabb kalibrációja érdekében.

Összefoglaló

A Vitalink fejlesztésével egy olyan mobilalkalmazást hoztunk létre, amely egyszerűen használható, mégis hasznos eszköze lehet az egészségünk nyomon követésének. Az okostelefon kameráját használva bárki könnyedén megmérheti a szívritmusát, különösebb eszközök vagy orvosi háttér nélkül.

A mérések pontosságát összehasonlítottuk olyan, már jól bevált eszközökkel, mint az okosórák vagy a pulzoximéterek. Az eredmények alapján arra jutottunk, hogy bár a Vitalink jelenlegi formájában még nem alkalmas arra, hogy teljes mértékben kiváltsa ezeket az eszközöket, a megfelelő fejlesztésekkel nagyon közel kerülhet hozzájuk. Ez biztató kiindulópontot jelent a további munkához.

A fejlesztéshez a React Native és a Java Spring technológiákat választottuk, amelyek gyors, hatékony és platformfüggetlen fejlesztést tettek lehetővé. Ez lehetőséget ad arra, hogy az alkalmazásunk a jövőben szélesebb körben is elérhetővé váljon, akár Androidon, akár iOS-en.

Úgy gondoljuk, hogy a Vitalink jó példa arra, hogyan lehet a technológiát az egészség szolgálatába állítani. A jövőben szeretnénk még továbbfejleszteni az alkalmazást, például azzal, hogy képes legyen észrevenni hosszabb távú változásokat, vagy akár figyelmeztetni, ha valami szokatlant észlel. Célunk, hogy egy olyan eszközt adjunk az emberek kezébe, amellyel könnyebben odafigyelhetnek saját jólétükre.

Ábrák jegyzéke

Táblázatok jegyzéke

Kódrészletek jegyzéke

Irodalomjegyzék

[ANP23] Panayiotis Antoniou, Marios Nestoros, and Anastasis C. Polycarpou. Calculation of heartbeat rate and spo2 parameters using a smartphone camera: Analysis and testing. *Sensors*, 23(2), 2023.

[JZS21] B. Jalilian, M.S. Zakerhamidi, and M. Sahrai. Linear and nonlinear optical properties of human hemoglobin. *Spectrochimica Acta Part A: Molecular and Biomolecular Spectroscopy*, 244:118855, 2021.

[TMSY14] Toshiyo Tamura, Yuka Maeda, Masaki Sekine, and Masaki Yoshida. Wearable photoplethysmographic sensors—past and present. *Electronics*, 3(2):282–302, 2014.

[Wor23] World Health Organization. Cardiovascular diseases (cvds), 2023.