SAPIENTIA ERDÉLYI MAGYAR TUDOMÁNYEGYETEM
MAROSVÁSÁRHELYI KAR,
INFORMATIKA SZAK

Skinology: A Bőrápolás és Technológia Találkozása

TUDOMÁNYOS DIÁKKÖRI KONFERENCIA DOLGOZAT

Témavezető:
dr. Novák Pálma-Rozália,
Egyetemi adjunktus

Végzős hallgatók:
Dezső Szabolcs, Kádár Ákos-Gergő

2025

Kivonat

A bőrápolás egyre nagyobb jelentőségű terület a modern társadalomban, ahol a szubjektív, egyéni igények és a tudományos ismeret kulcsfontosságú szerepet játszanak az emberek életében. Az emberek egy része felismeri, hogy a jól ápolt és igényes bőr hozzájárul a magabiztos és pozitív megjelenéshez egyaránt a magán- és szakmai életükben is, ennek ellenére a társadalom nagy része nem fordít elég hangsúlyt a bőrük megfelelő ápoltságára.

Jelen dolgozat célja egy innovatív és kompakt mobilalkalmazás, a **Skinology** fejlesztése, amely mesterséges intelligencia (MI) segítségével személyre szabott és gyors bőrápolási megoldásokat kínál a felhasználóknak. Az applikáció három funkcióra épül, amelyekkel hozzájárul a felhasználók mindennapjainak átalakításához:

- Személyre szabott bőrápolási rutinok kialakítása a rendszeresség érdekében. A felhasználó emlékeztető értesítést kap a rutinja megkezdése előtt, így nem felejti el az elvégzését.
- Bőrápolási termékek összetevőinek elemzése, amely segítségével többletinformációt kaphat a felhasználó, arról, hogy pontosan mit is használ a bőrére. Az összetevők kategorizálása segítségével egyértelműen látszik az, hogy melyik termék mennyire egészséges a bőrfelületre nézve.
- Bőranalízis AI segítségével. Az alkalmazás egy saját, személyre szabott AI modell segítségével elemzi a felhasználók által feltöltött fotókat és azonosítja a potenciális bőrproblémákat, vagy pozitív visszajelzést ad, hogy nincs semmilyen betegség az adott felületen, ezzel időt spórolva a felhasználóknak.

A **Skinology** egy ígéretes eszköz a személyre szabott dermatológia területén, amely a mesterséges intelligencia és a bőrápolási tudomány ötvözésével hatékony megoldást kínál a felhasználóknak, mindezt emberközeli és intuitív módon. Az applikáció célja, hogy hozzájáruljon a tudatos bőrápoláshoz és növelje a bőrgyógyászati ellátás hatékonyságát. A Skinology segítségével az emberek bőre ápoltabb, megjelenésük igényesebb és magabiztosabb lesz.

Kulcsszavak: bőrápolás, dermatológia, mobil applikáció, mesterséges intelligencia

Tartalomjegyzék

1. fejezet

Bevezető

A modern bőrgyógyászati technológiák és a digitalizáció térnyerésével egyre inkább elérhetővé válnak azok az eszközök, amelyek megkönnyítik a bőrápolással kapcsolatos döntéseinket. Ezek közé tartozik a **Skinology** is, amely forradalmi módon segíti a felhasználókat a kozmetikumok összetevőinek megértésében és a bőrápolási rutinok személyre szabásában. De miért van szükségünk egy ilyen alkalmazásra és hogyan könnyítheti meg mindennapjainkat?

A bőrápolási termékek választéka ma már szinte végtelen, ugyanakkor az összetevők megértése és azok hatásának pontos ismerete gyakran komoly kihívást jelent. A Skinology ebben nyújt segítséget: az alkalmazás képfelismerő technológiát alkalmazva képes az összetevők gyors és pontos beazonosítására. Nincs szükség hosszas keresgélésre, elég lefotózni a termék összetevőlistáját és az applikáció azonnal részletes információval lát el bennünket. Ezáltal könnyen eldönthetjük, hogy egy adott termék valóban megfelel-e bőrtípusunk igényeinek, vagy érdemes inkább más alternatíva után nézni.

A Skinology nem csupán az összetevők elemzésében jeleskedik: további előnye, hogy támogatja a személyre szabott bőrápolási rutin kialakítását is. Az alkalmazás segítségével összeállíthatjuk azokat a lépéseket, amelyek a legfontosabbak számunkra, legyen szó a tisztításról, hidratálásról, vagy a fényvédelemről. Ez a funkcionalitás nemcsak rendszerezi a bőrápolási szokásainkat, hanem egy olyan fenntartható és hosszú távon hatékony folyamatot biztosít, amely figyelembe veszi az egyéni szükségleteinket.

Az applikáció harmadik, és egyben az egyik leghasznosabb funckionalitása a bőrdiagnoztizálás, amely segítségével a felhasználók analízist készíthetnek a sérült bőrfelületükről és további információkat szerezhetnek arról, hogy egészséges a bőrük vagy valamilyen betegség beazonosítására került sor.

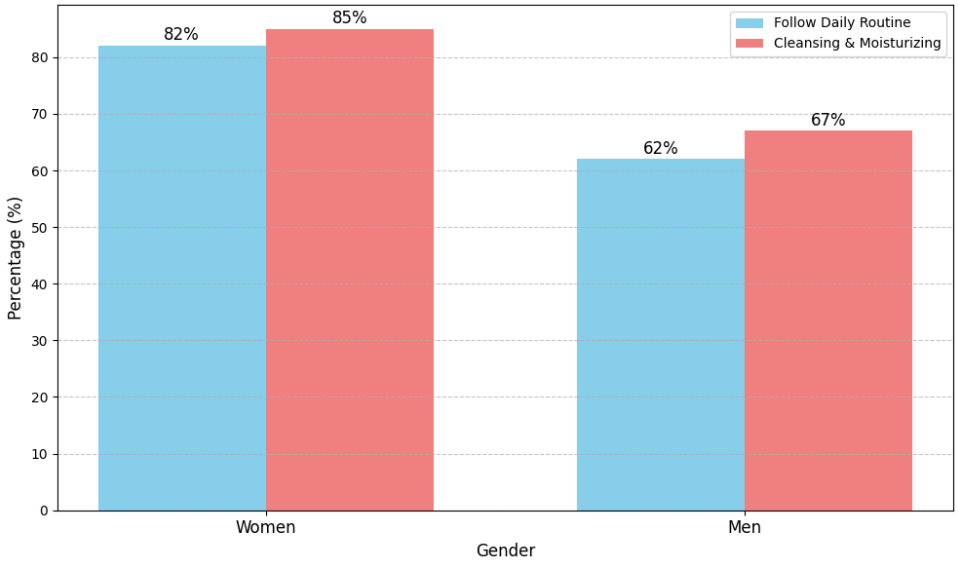
Az applikáció különösen hasznos azok számára, akik tudatosan szeretnék megválasztani bőrápolási termékeiket, miközben időt és energiát spórolnak. A Skinology egyszerűen kezelhető, felhasználóbarát megoldás, amely modern technológiát alkalmazva segít megalapozott döntéseket hozni, így biztosítva, hogy bőrünk a lehető legjobb ápolásban részesüljön. Ez az innovatív digitális eszköz egy új korszakot nyit a bőrápolás világában, amely sokkal kényelmesebb a páciensek számára.

2. fejezet

Szakirodalmi áttekintés

2.1. Bevezető

A bőrápolás területén az utóbbi években egyre nagyobb figyelem irányul a kozmetikai termékek összetevőinek tudatos kiválasztására, valamint a személyre szabott bőrápolási rutinok kialakítására, ahogy a 2.1 ábrán is megfigyelhető. A felhasználók gyakran szembesülnek olyan kihívásokkal, mint az összetevők értelmezése vagy a bőrtípusuknak megfelelő termékek megtalálása. A modern technológia, különösen a mesterséges intelligencia és a képfelismerési rendszerek fejlődése, új lehetőségeket kínál ezeknek az információknak a könnyű elérésére.



2.1. ábra. A napi bőrápolási rutin követése nőknél és férfiaknál (Forrás: Statista)

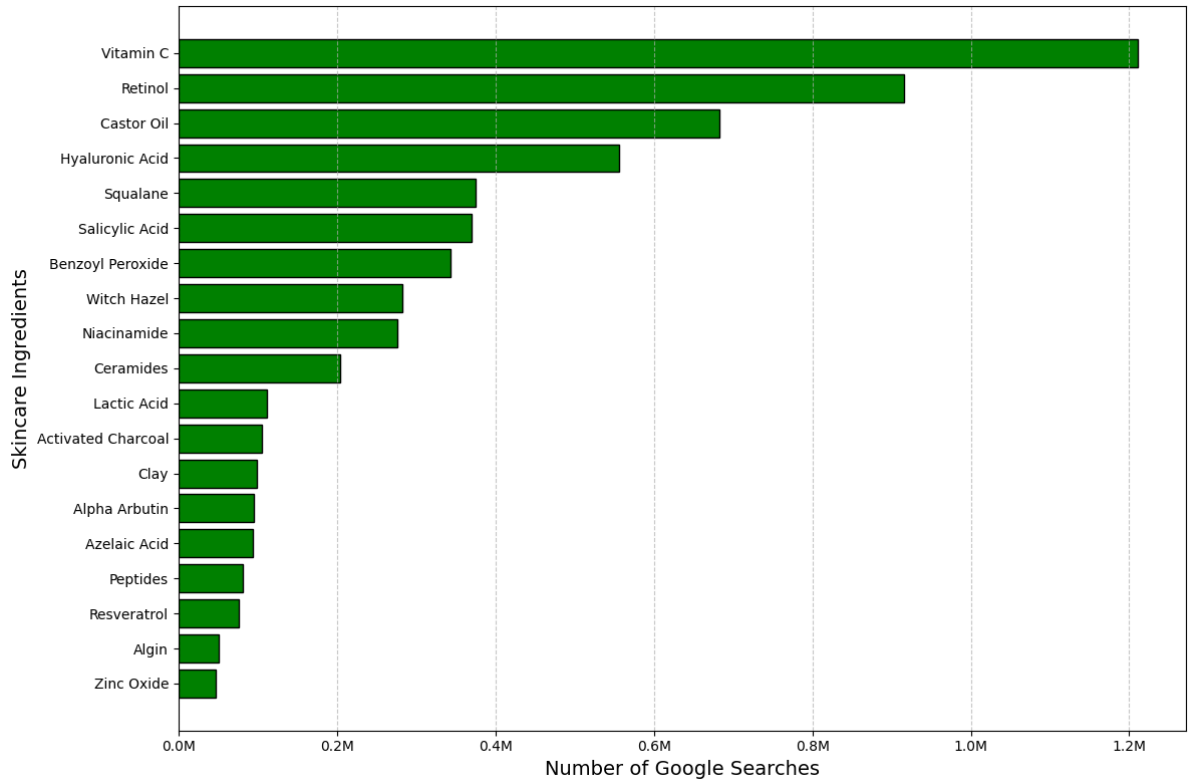
2.1.1. Bevezetés a bőrápolás és kozmetikai összetevők világába

Az elmúlt évtizedekben a bőrápolás és kozmetikai termékek piaca jelentős változásokon ment keresztül, amelyeket a fogyasztói tudatosság növekedése és a bőr egészségének

kiemelt szerepe inspirált. A fogyasztók ma már egyre inkább tudatosan választanak és keresnek olyan termékeket, amelyek nemcsak a szépséget szolgálják, hanem a bőr egészségére is jótékony hatással vannak.

Az iparági innovációknak köszönhetően a szépségápolás egyre inkább személyre szabottá válik, figyelembe véve a különböző bőrtípusok és bőrproblémák egyedi igényeit. A bőrgyógyászati kutatások és a modern technológia lehetővé tették, hogy a bőrápolás hatékonyabbá váljon, és az összetevők pontosan a kívánt hatásra koncentráljanak. Az aktív hatóanyagok, mint a hialuronsav, a C-vitamin vagy a retinol, kulcsfontosságúak a bőrápolásban, mivel jelentősen befolyásolják a bőr állapotát és megjelenését.

A bőrápolás hatékonysága szempontjából az összetevők pontos ismerete alapvető. Az egyes összetevők, mint például a növényi olajok, antioxidánsok vagy hidratáló anyagok, különböző bőrproblémákra adnak megoldást, így segíthetnek a bőr fiatalon tartásában, az irritációk csökkentésében, vagy a hidratáltság megőrzésében. Mivel a fogyasztók egyre inkább keresik a minőségi és biztonságos termékeket, az iparági trendek is azt mutatják, hogy a személyre szabott, hatékony és tudományos alapú bőrápolás felé mozdul el a piac. Emellett a fenntarthatóság és az etikus előállítás is egyre fontosabb szerepet kap, mivel a fogyasztók nemcsak a bőrük, hanem a környezetük védelme iránt is érzékenyebbek. A természetes, organikus összetevők használata és a környezetbarát csomagolás iránti igények tovább erősítik a piac változását.



2.2. ábra. Bőrápolási összetevők népszerűsége Google-keresések alapján

2.2. A kozmetikai összetevők szerepe és kiválasztása

A bőrápolás alapját azok az aktív hatóanyagok képezik, amelyek célzottan javítják a bőr állapotát és megjelenését, például a hidratáció fokozásával, a textúra finomításával vagy a ráncok csökkentésével. Olyan ismert összetevők, mint a hialuronsav, a retinoidok, a niacinamid és a C-vitamin, mind speciális célokat szolgálnak a bőrápolásban. A hialuronsav például kiválóan hidratál, míg a retinoidok és a C-vitamin elősegítik a bőr megújulását és védelmet nyújtanak az öregedés jeleivel szemben. Az Európai Unió által fenntartott COSING[1] adatbázis – amely bárki számára elérhető – átfogó információforrásként szolgál a kozmetikai összetevőkről. Az adatbázis részletes tájékoztatást nyújt az engedélyezett és korlátozott összetevőkről, segítve mind a szakértőket, mind a fogyasztókat az összetevők tulajdonságainak és jogszabályi hátterének megértésében. A COSING adatbázis lehetőséget biztosít arra, hogy bárki ellenőrizze a kozmetikai összetevők biztonságosságát és hatékonyságát, valamint tájékozódjon az európai szabályozásokról.

2.3. Tudományos alapok és klinikai kutatások

Számos tudományos tanulmány vizsgálja a kozmetikai összetevők hatékonyságát és biztonságosságát, különösen azokat, amelyek széles körben használtak a bőrápolásban. A bőr hidratációját, védőrétegének erősítését és a textúra javítását célzó összetevőket rendszeresen klinikai kísérleteknek vetik alá, hogy megértsék azok rövid- és hosszú távú hatásait. A British Journal of Dermatology[2] [FHR+10] egyik tanulmánya például részletezi a niacinamid, a retinoidok és a peptidek kombinációját alkalmazó készítmények hatásait. Az ilyen kutatások alátámasztják, hogy ezek az összetevők nemcsak a bőr hidratáltságát növelik, hanem antioxidáns és gyulladáscsökkentő hatással is bírnak, ami egészségesebb és egységesebb bőrt eredményez. A tudományos eredmények hozzáférhetőségének és a szigorú szabályozásoknak köszönhetően a fogyasztók egyre inkább megalapozott döntéseket hozhatnak a bőrápolási termékek kiválasztásában. A klinikailag is igazolt hatóanyagok alkalmazása révén a felhasználók személyre szabottan választhatják ki azokat az összetevőket, amelyek leginkább megfelelnek saját bőrtípusuk igényeinek és egyedi bőrproblémáiknak.

2.4. A mesterséges intelligencia szerepe a bőrápolásban

Az utóbbi években a mesterséges intelligencia (AI) alkalmazása jelentős fejlődésen ment keresztül a bőrápolás területén, különösen a bőrdiagnosztikai eszközök és a személyre szabott bőrápolási ajánlások terén. Az AI képes felismerni a bőrproblémákat, mint például a pattanások, pigmentációs zavarok, valamint különböző bőrbetegségeket. Az AI rendszerek elemzik a bőr állapotát, valamint gyorsan és pontosan képesek azonosítani a gyanús elváltozásokat, így segítve a korai diagnózist és a megfelelő kezelési lehetőségek kiválasztását.

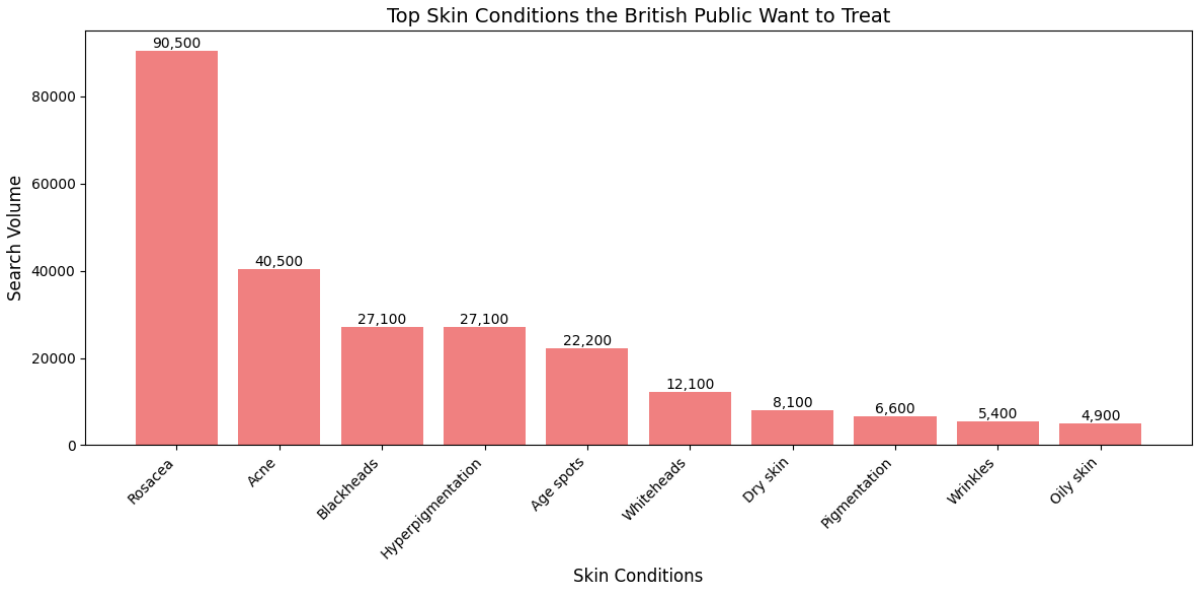
A mesterséges intelligencia szerepe nemcsak a diagnosztikára korlátozódik. Az AI hasznos a bőrápolási termékek fejlesztésében is, mivel képes előre jelezni az összetevők

[1] Cosmetic Ingredients Notification

[2] A British Journal of Dermatology egy elismert szakfolyóirat, amely a bőrgyógyászat klinikai, kísérleti és molekuláris kutatásaival foglalkozik

hatékonyságát és segíteni a bőrápoló készítmények formulájának optimalizálásában, alkalmazkodva a különböző bőrtípusokhoz. Különösen a kozmetikai ipar számára fontos az AI, mivel képes figyelembe venni a felhasználók egyedi igényeit, és személyre szabott bőrápolási megoldásokat kínálni. Ezen kívül segít a bőrproblémák kezelésére alkalmazott termékek összetevőinek leghatékonyabb kombinálásában is.

A mesterséges intelligencia alkalmazásának egyik legnagyobb előnye, hogy gyorsabbá és pontosabbá teszi a bőrápolás folyamatát, miközben lehetővé teszi a kezelések egyéni igényekhez való igazítását. Egy 2020-as kutatás [GKYH20] hangsúlyozza, hogy az AI különösen hasznos a bőrbetegségek, például a melanóma[3] korai felismerésében, mivel gyorsabb és pontosabb orvosi diagnózisokat tesz lehetővé. Ez végső soron javítja a kezelések hatékonyságát és csökkenti a bőrproblémák súlyosbodásának esélyét.



2.3. ábra. A legkeresettebb bőrápolási problémák a brit közönség körében, keresési volumenek alapján[Coo23]

[3]A melanóma a bőr egyik legveszélyesebb rákos megbetegedése, amely a bőr pigmentsejtjeiben (melanocitákban) keletkezik. Gyorsan terjedhet a szervezet más részeire, ezért fontos a korai felismerés és kezelés

3. fejezet

Hasonló alkalmazások bemutatása

3.1. Hasonló alkalmazások keresése

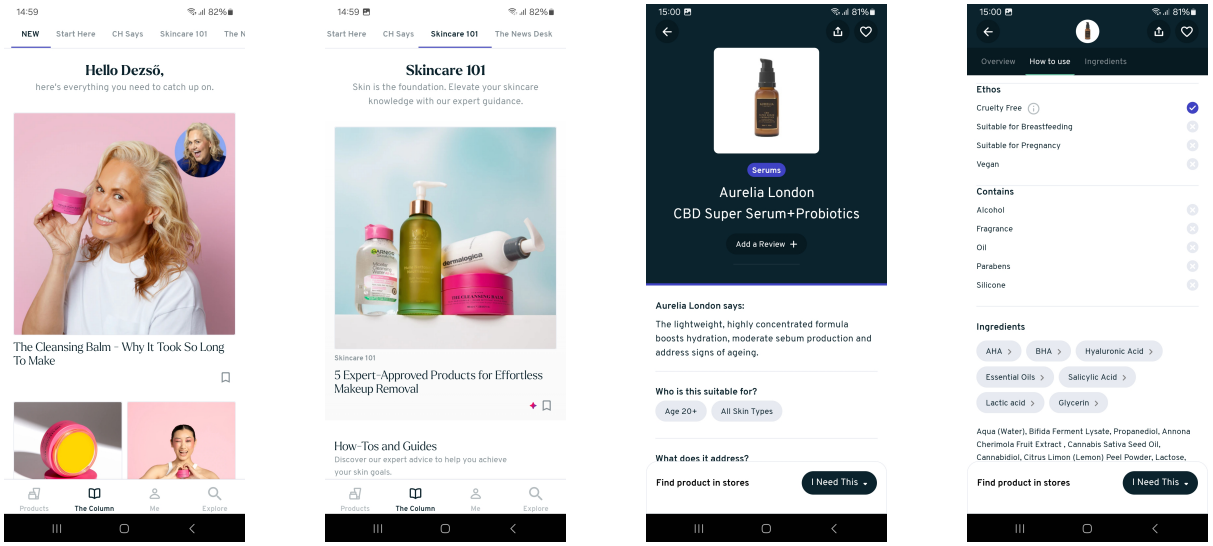
A technológia gyors fejlődése napjainkban számos digitális szolgáltatást integrált az emberek mindennapjaiba, és ez alól az orvostudomány sem kivétel. Egyre több mobilalkalmazás segíti a pácienseket abban, hogy akár otthonról is felismerjék tüneteiket és követhessék egészségügyi állapotukat. Az egyik első ilyen kísérlet a bőrgyógyászat területén 2008-ban látott napvilágot DermLink néven. Ezt az alkalmazást kifejezetten bőrgyógyászok számára fejlesztették ki, hogy támogassa őket a bőrbetegségek diagnosztizálásában és kezelésében. A DermLink lehetővé tette a szakemberek számára, hogy képeket és adatokat osszanak meg pácienseikről más orvosokkal konzultáció céljából. Az alkalmazás úttörő szerepet töltött be, hiszen a telemedicina és a mobiltechnológia ötvözésével új utakat nyitott az orvosi együttműködés és a betegellátás terén. Napjainkban több mint 229 bőrgyógyászati vonatkozású alkalmazást azonosítottak a következő kategóriákban: általános bőrgyógyászati referencia (61 [26,6%]), önmegfigyelés/diagnózis (41 [17,9%]), betegségkalauz (39 [17,0%]), oktatási segédlet (20 [8. 7%]), napvédő/UV ajánlás (19 [8,3%]), kalkulátor (12 [5,2%]) stb [BEH+13].

Az alkalmazás fejlesztése előtt alapos kutatást végeztem a hasonló alkalmazások körében, amelyeket a mindennapjainkban használnak az emberek bőrápolás céljából. A Google Play Áruházban több olyan alkalmazást is találtam, amelyek különféle bőrápolási funkciókat kínálnak. Sajnos, ezek közül sokban a valóban hasznos funkciók – például a bőrbetegségek felismerése – fizetős szolgáltatásokhoz kötődnek, ami jelentősen korlátozza a felhasználók lehetőségeit. Ennek ellenére a keresés során azt tapasztaltam, hogy még mindig nagy az érdeklődés ezek iránt az alkalmazások iránt, hiszen több tízezer letöltéssel rendelkeznek. Az alábbiakban részletesen bemutatom bőrápolási funckionalitásokra csoportosítva a talált alkalmazásokat.

3.1.1. Bőrápolási rutin menedzselő alkalmazások

Számos olyan alkalmazással találkoztam, amelyek inkább bőrápolási rutinok nyomon követésére, illetve bőrápolási termékekre és kozmetikumokra fókuszálnak, mintsem a bőrbetegségek diagnosztizálására. A 3.1 képsorozaton látható, Skin Rocks alkalmazás inkább egy blogszerű felület, ahol a felhasználók szabadon böngészhetnek különböző bőrápolási cikkek, rutinok és beszámolók között. A Skinology-val ellentétben azonban nem

nyújt lehetőséget arra, hogy személyre szabott rutinokat hozzanak létre, így a felhasználók nem tudják saját igényeikhez igazítani a tartalmat – mintha csak egy digitális magazint olvasnának, interakció nélkül. Bár a Skin Rocks részletes információkkal szolgál számos előre meghatározott bőrápolási termékről, a felhasználók nem kereshetik meg saját termékeiket, ha azok épp nincsenek az adatbázisban. A Skinology egyik célja az, hogy lehetőséget adjon arra, hogy a felhasználók bármilyen termék összetevőlistáját beolvassák és személyre szabott információkat kapjanak.

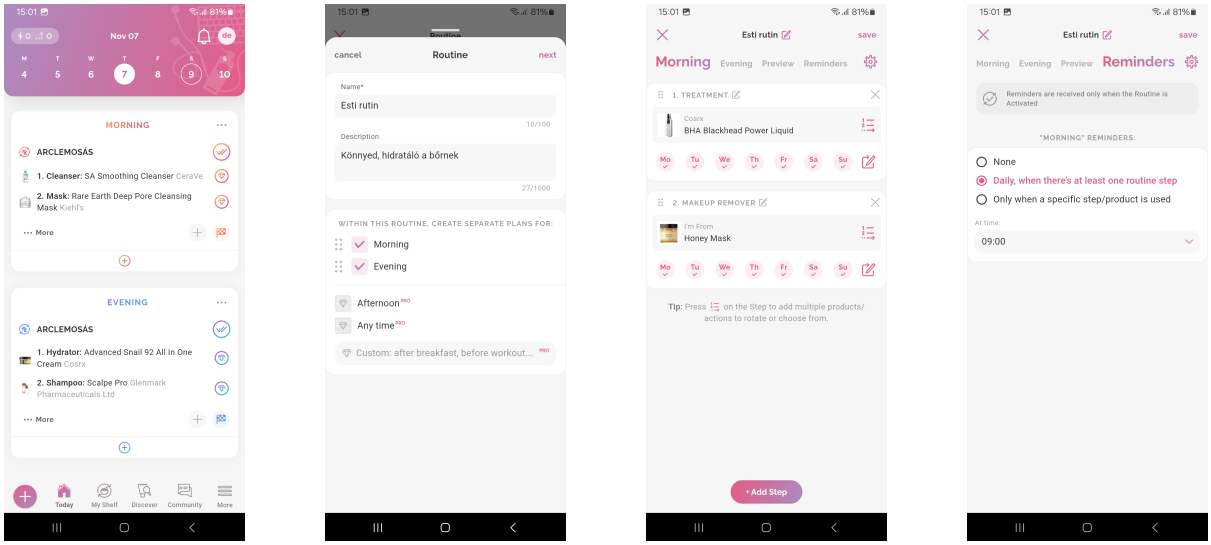


3.1. ábra. Skin Rocks alkalmazás fotók

A FeelinMySkin (lásd 3.2 képsorozat) egy másik alkalmazás, amely bőrápolási rutin menedzselést kínál, és lehetővé teszi, hogy a felhasználók nevet és leírást adjanak a rutinjaiknak, illetve meghatározzák a rutin lépéseit. A Skinologyval ellentétben a FeelinMySkin előre beépített termékeket tart számon, és a felhasználók csak ezek közül választhatnak, saját termékeket nem adhatnak hozzá. Emellett a FeelinMySkinben a bőrápolási rutinok csak reggeli és esti bontásban érhetők el, mivel egyéb időszakok – például a délutáni rutin – hozzáadása fizetős funkció. A Skinology küldetése ezzel szemben az, hogy ingyenesen tegye lehetővé a rutinok rugalmas időzítését, akár óra és perc pontossággal is, és automatikusan döntse el, hogy az adott rutin reggeli, déli vagy esti.

Továbbá a FeelinMySkin alkalmazásban az emlékeztető értesítéseket manuálisan kell aktiválni minden egyes rutinra, míg a Skinology ezt automatikusan megteszi a felhasználók számára. A FeelinMySkin egyik legnagyobb hátránya véleményem szerint az, hogy egyszerre csak egyetlen rutinról érkezik emlékeztető ingyenesen, míg a Skinologyban minden létrehozott rutin automatikusan aktiválva és időzítve van.

A funkcionalitások mellett a FeelinMySkin felhasználói felülete is meglehetősen összetett és kevésbé felhasználóbarát. Míg egy rutin létrehozása a FeelinMySkinben minimum 20 kattintást igényel, a Skinologyban ez mindössze minimum 14 kattintással megvalósítható. Ez is bizonyítja azt, hogy az alkalmazás fejlesztése közben fokozott figyelmet kapott a fehasználóbarát felhasználó felület megtervezése és az, hogy minden funkcionalitás a lehető legkönyebben elérhető legyen a felhasználók számára.



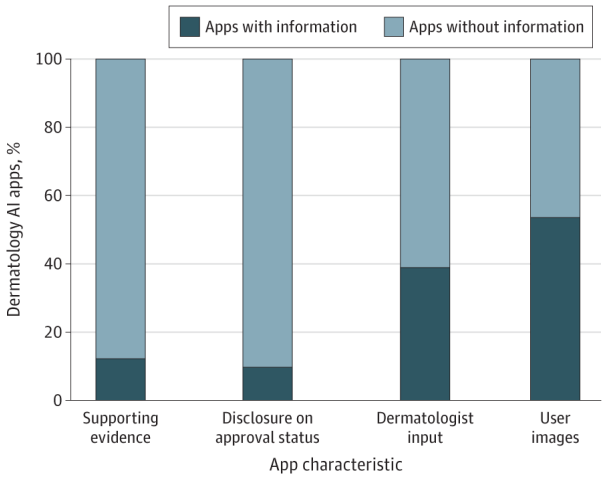
3.2. ábra. FeelinMySkin alkalmazás fotók

3.1.2. Bőrbetegség azonosító alkalmazások

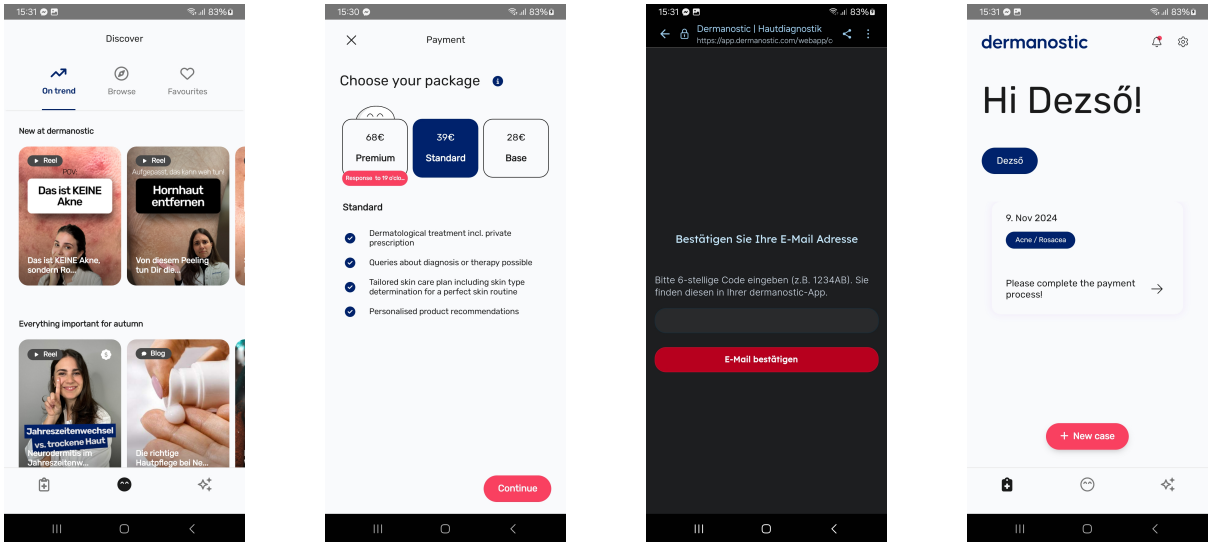
Az előzetes kutatás során olyan alkalmazásokat is igyekeztem keresni, amelyek dermatológiai funkciókkal rendelkeznek, de nem kifejezetten bőrápolásra, hanem bőrbetegségek felismerésére és beazonosítására szolgálnak. Találtam néhány ilyen alkalmazást, de sajnos szinte mindegyik esetében a bőrbetegségek tényleges beazonosítása fizetős szolgáltatás volt. Amint az a 3.3 ábrán látható, számos olyan alkalmazás található az áruházakban, amelyek hitelessége megkérdőjelezhető, mivel nem állnak mögöttük megbízható bizonyítékok, és nem rendelkeznek orvosi vagy bőrgyógyászati szakértők általi hitelesítéssel. Nagyon kevés applikáció szolgál tényleges és használható információval, így az Egyesült Államok Élelmiszer- és Gyógyszerügyi Hatósága még egyiket sem hagyta jóvá. Bár a mesterséges intelligencia megjelenése segítette a dermatológiai alkalmazások térhódítását, még így is jelentős részük nem ad érdemi információt a felhasználóknak [WYP+24].

Az egyik ilyen alkalmazás a 3.4 fotókon látható Dermanostic, amelyet kifejezetten bőrbetegségek felismerésére és kezelésére fejlesztettek. Azonban itt is elengedhetetlen a fizetés, így ingyenes diagnózisra nincs lehetőség, míg a Skinology célja az, hogy ingyenes kezelést ajánljon. További hátrányként említhető, hogy a Dermanostic alkalmazást főleg a német felhasználók számára fejlesztették: sok szöveges tartalom kizárólag német nyelven érhető el - sőt, még a regisztráció utáni e-mail megerősítés is német nyelven van - fordítási lehetőség nélkül, és a diagnosztikai szolgáltatások is kizárólag németországi magánklinikákra korlátozódnak. Az alkalmazás egy blogszerű szekciót is tartalmaz, ahol különböző oktató anyagok, videók és cikkek találhatók, de ezek szintén német nyelvűek, és hivatalos fordítás nincs hozzájuk. Ráadásul ahhoz, hogy a felhasználó tényleges diagnózist kapjon, a Dermanostic alkalmazásban 19 különböző lépésen kell végighaladnia, miközben számos személyes adatot kell megadnia. A Skinology kapcsán az a cél, hogy mindez egyszerűbb lépéseken keresztül és gyorsan történjen meg.

Az Aysa alkalmazás az egyik legnépszerűbb és orvosok által hitelesített bőrgyógyászati tünetazonosító eszköz. Segítségével egyszerűen lefotózhatjuk a problémás bőrfelüle-



3.3. ábra. Használható információval rendelkező bőrgyógyászati alkalmazások



3.4. ábra. Dermanostic alkalmazás fotók

tet, és díjmentesen tájékozódhatunk az esetleges betegségekről. Az app információt nyújt a tünetek lehetséges okairól, megmutatja, mikor érdemes orvoshoz fordulni, és milyen kezeléseket javasolhat egy szakember. Hasonló cél vezérli a Skinology fejlesztését is: a páciensek támogatása teljesen ingyenesen, anyagi elköteleződés nélkül. Az Aysa egyetlen hátránya, hogy a diagnosztikai folyamat során különböző bőrbetegségek listáját és leírását adja meg, de nem ad egyértelmű választ arra, hogy a lefotózott bőrfelület egészséges-e vagy sem.

4. fejezet

A rendszer specifikációja

4.1. Felhasználói követelmények

4.1.1. Bevezető

A felhasználói követelmények meghatározása kulcsfontosságú annak érdekében, hogy az applikáció megtervezése és kialakítása a célcsoport igényeire szabott legyen. Az alábbi követelmények biztosítják, hogy az alkalmazás hatékonyan támogassa a bőrápolás folyamatát és könnyen hozzáférhető információt nyújtson a kozmetikumok összetevőiről. Ezek az elvárások szolgálnak alapul a fejlesztési és tesztelési folyamatok során, biztosítva, hogy az alkalmazás megfeleljen a felhasználók elvárásainak és elősegítse a tudatos bőrápolási szokások kialakítását.

4.1.2. Felhasználói szerepek

Az alkalmazás két alapvető felhasználói szerepkört különít el, amelyek különböző funkciókhoz és jogosultságokhoz biztosítanak hozzáférést, ahogy a use-case diagramon (lásd 4.1) is látható. A bejelentkezés folyamata a 4.2 szekvencia diagramon található, amely részletesen leírja a bejelentkezés menetét.

1. Bejelentkezett felhasználó

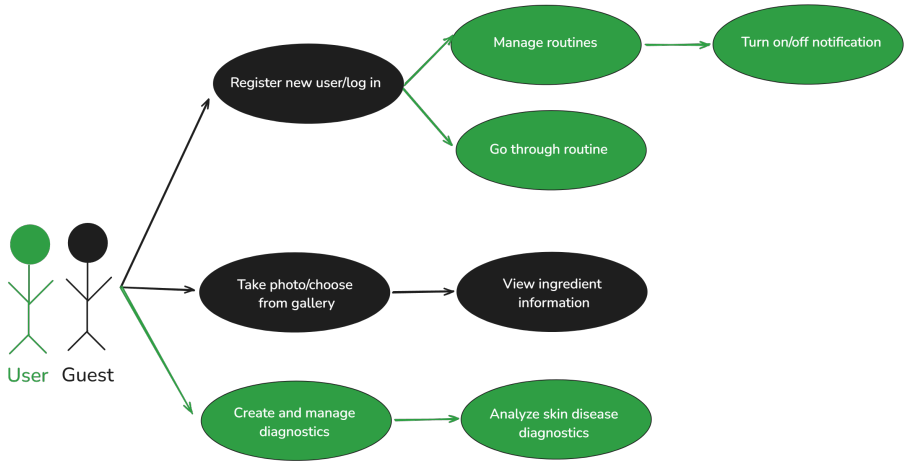
A bejelentkezett felhasználók számára a következő funkciók érhetők el:

- **Saját rutinok létrehozása:** A felhasználók személyre szabott bőrápolási rutinokat hozhatnak létre és menthetnek.
- **Értesítések küldése:** Az alkalmazás értesítéseket küld a felhasználóknak a bőrápolási rutinjuk előre beállított kezdési időpontjában.
- **Kép alapú információszerzés:** Lehetőség van a kozmetikumok összetevőiről többletinformáció szerzésére képfelismerés alapján.

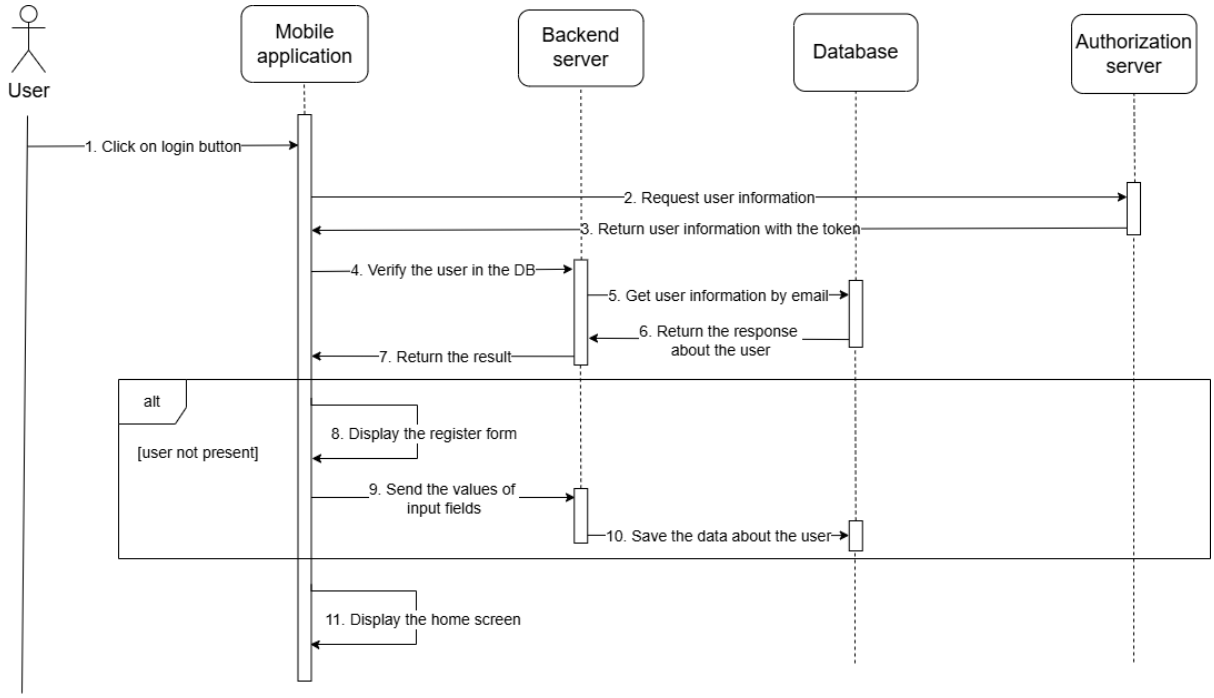
2. Vendég felhasználó

A vendég felhasználók számára a következő funkció érhető el:

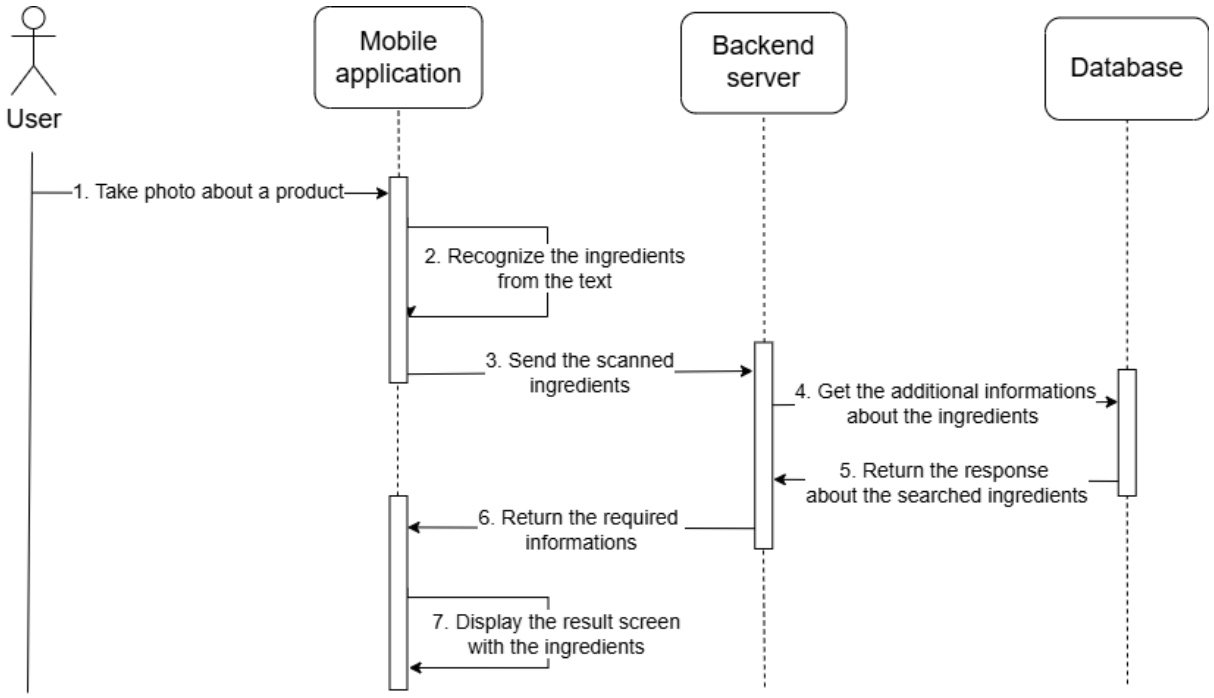
- **Kép alapú információszerzés:** A kozmetikumok összetevőiről képfelismerés alapján többletinformációt szerezhetnek, regisztráció vagy bejelentkezés nélkül, amint az a 4.3 diagramon részletesen is látható.



4.1. ábra. Az applikáció use-case diagramja



4.2. ábra. A bejelentkezés szekvencia diagramja



4.3. ábra. A szkennelés szekvencia diagramja

4.1.3. Felhasználói elvárások

A felhasználói elvárások meghatározzák azokat az alapvető igényeket, amelyeket az alkalmazásnak teljesítenie kell annak érdekében, hogy a felhasználók számára kényelmes, hatékony és pozitív élményt biztosítson. Az alábbiakban felsoroljuk azokat a főbb elvárásokat, amelyek figyelembevételével az alkalmazás fejlesztésére és felhasználói felület tervezésére sor kerülhet:

- **Könnyen használható felület:** Az alkalmazás felületének felhasználóbarátnak kell lennie, hogy a felhasználók könnyedén navigálhassanak és gyorsan megtalálják a keresett funkciókat. A Skinology ennek értelmében egy minimalista, letisztult dizájnt kínál.
- **Személyre szabhatóság:** A bejelentkezett felhasználók számára lehetőséget kell biztosítani a bőrápolási rutinok saját igényeik szerint történő létrehozására és testreszabására.
- **Adatbiztonság és védelem:** Az alkalmazásnak kiemelt figyelmet kell fordítania a felhasználói adatok védelmére, beleértve a személyes információkat és a bőrápolási preferenciákat.
- **Szinkronizálás több eszköz között:** A felhasználói adatokat minden eszközön szinkronizálni kell, hogy a felhasználók bárhonnan és bármikor elérhessék a személyre szabott információkat, ezért a rutinok lokális adatbázis helyett egy központi adatbázisban vannak eltárolva

- **Pontosság és megbízhatóság:** Az alkalmazásnak biztosítania kell, hogy a kozmetikumokkal és azok összetevőivel kapcsolatos információk pontosak és naprakészek legyenek, segítve a felhasználókat a megalapozott döntéshozatalban.

4.2. Rendszerkövetelmények

4.2.1. Funkcionális követelmények

Felhasználó kezelése

- A rendszer lehetővé teszi a felhasználók regisztrációját, ahol a felhasználó megadja nevét, email címét, születési dátumát és egyéb alapvető adatokat.
- A rendszer biztosítja a felhasználók biztonságos bejelentkezését Google segítségével, amelyhez a felhasználó email címét és jelszavát kell megadni.
- A bejelentkezést követően a rendszer hozzáférést biztosít a felhasználó profiljához, ahol lehetőség van a személyes adatok megtekintésére.
- A rendszer biztosítja a kijelentkezést, amely lehetővé teszi, hogy a felhasználó véglegesen kijelentkezzen a rendszerből, így ha újonnan szeretné elérni rutinjait, újra be kell jelentkeznie.
- A felhasználók automatikusan kijelentkezésre kerülnek egy megadott idő után, amelyet előzetesen kell specifikálni, ezzel is elősegítve a biztonságot.

Bőrápolási termékek összetevőinek szkennelése

- A rendszer a felhasználó számára egy kamerát kellene biztosítson, amellyel lehetősége van lefényképezni a bőrápolási termékek összetevőinek listáját.
- Ha a felhasználó egy régebbi, már lefényképezett termék összetevő listáját szeretné újra beszkennelni, a rendszer lehetőséget ad arra, hogy a galériából válasszon régebbi fényképeket.
- A fényképezést vagy a fénykép kiválasztását követően a rendszer kiegészítő információkat nyújt a felhasználónak az összetevőkről, amelyek a termék címkén szerepelnek, mint például az adott összetevő leírása vagy hatásai. Továbbá az alkalmazás kilistázza a felhasználónak azt, hogy a megtalált összetevők közül melyek hasznosak, károsak vagy esetleg semlegesek a felhasználó bőrére.

Bőrápolási rutinok menedzselése

- Az alkalmazás lehetőséget biztosít a felhasználónak egyedi, személyre szabott bőrápolási rutinok lérehozására.
- Amennyiben a felhasználónak több lépésből állnak a rutinjai, a rendszer biztosítja, hogy a rutin lépéseket is specifikálja a felhasználó például a lépés nevének, leírásának, időtartamának meghatározásával. E mellett a rendszer lehetőséget kínál a felhasználónak, hogy a terméket is specifikálja, amivel a rutint szeretné végrehajtani.

- A rutin létrehozása után az applikáció lehetővé teszi, hogy a felhasználó végre tudja hajtani az előzetesen létrehozott rutinjait és lépésről-lépésre minden rutinlépésen végigiteráljon.
- Az applikáció a rutinok kategorizálást is el kell végezze három különböző kategória szerint: reggeli, déli és esti.
- Abban az esetben ha a felhasználó már haszontalannak gondolja a létrehozott rutinjait a rendszer lehetőséget biztosít a már nem használt rutinok törlésére.

Értesítések küldése

- A rendszer képes emlékeztető értesítéseket küldeni a felhasználó készülékére a rutin kezdése előtt 5 perccel, ezzel tájékoztatva a felhasználót, hogy nemsokára kezdődik a rutinja.
- Abban az esetben ha a felhasználó több eszközzel van bejelentkezve az alkalmazásba a rendszer biztosítja, hogy ugyanabban az időpillanatban megkapja a felhasználó az összes bejelentkezett eszközön az értesítést.

Bőrbetegségek beazonosítása

- Az applikáció képes kell legyen különböző bemenő képek alapján eldönteni, hogy a fotókon levő bőrfelületen betegség látható-e.
- Az alkalmazás eredményként meg kell jelenítse a felhasználó számára, hogy mekkora esély van egy adott bőrbetegség kialakulására és további információkat kell tudjon szolgáltatni a diagnosztizált betegségről.

4.2.2. Nem funkcionális követelmények

Termék követelmények

Biztonság

- Az alkalmazásnak biztosítania kell a felhasználói adatok titkosságát, és védenie kell a személyes adatokat a jogosulatlan hozzáféréstől.
- A rendszernek képes kell lennie autentifikálni a felhasználót és számon tartania azt, hogy milyen jogosultágokkal rendelkezik.
- Az alkalmazásnak csak a szükséges jogosultságokat kell kérnie, és azokat megfelelő módon kell kezelnie. Ide tartoznak például a kamera, a galéria és az értesítések használatára vonatkozó engedélyek.

Teljesítmény

- Az alkalmazásnak gyors válaszidővel kell működnie, a felhasználói interakciókra (pl. gombnyomás, navigálás) 1-2 másodpercen belül reagálnia kell.

- Az alkalmazásnak képesnek kell lennie nagy adatállományok gyors betöltésére anélkül, hogy a felhasználói élményt rontaná. Az adatok begyűjtése nem vehet igénybe maximum 2 másodpercnél több időt.

Hardver és hálozati követelmények

- A felhasználónak okostelefonnal kell rendelkeznie, amelyen Android vagy IOS operációs rendszer fut és stabil internetkapcsolattal kell rendelkeznie az alkalmazás gördülékeny működése érdekében.

Elérhetőség és hibatűrés

- A rendszer 99,9%-os éves rendelkezésre állással kell működjön. Amint a 4.4 táblázatban látható, a hónapos leállás nem haladhatja meg a 43.83 percet.
- Abban az esetben ha leállás történik, a rendszernek képesnek kell lennie helyreállítani saját magát és tovább szolgálni a bejövő kéréseket.

Availability %	Downtime per year	Downtime per month	Downtime per week	Downtime per day
99% ("two nines")	3.65 days	7.31 hours	1.68 hours	14.40 minutes
99.5% ("two nines five")	1.83 days	3.65 hours	50.40 minutes	7.20 minutes
99.9% ("three nines")	8.77 hours	43.83 minutes	10.08 minutes	1.44 minutes
99.95% ("three nines five")	4.38 hours	21.92 minutes	5.04 minutes	43.20 seconds
99.99% ("four nines")	52.60 minutes	4.38 minutes	1.01 minutes	8.64 seconds
99.995% ("four nines five")	26.30 minutes	2.19 minutes	30.24 seconds	4.32 seconds
99.999% ("five nines")	5.26 minutes	26.30 seconds	6.05 seconds	864.00 milliseconds

4.4. ábra. Magas elérhetőség táblázat [1]

Szervezeti követelmények

Skálázhatóság és terheléselosztás

- Az alkalmazás a folyamatos skálázódásra nyitott kell legyen a növekedő terhelés esetén, annak érdekében, hogy minden kérést ki tudjon szolgálni. A minumum memória igény 512Mi, CPU 500m kell legyen.
- A rendszer képes kell legyen a bejövő kéréseket megfelelően elosztani és a terhelést korlátozni.

Integrációs képességek

- A rendszer kompatibilis kell legyen más alkalmazásokkal is kommunikálni a REST API segítségével, így biztosítva a teljes mértékű integritását a rendszernek.

[1] Forrás: Wikipédia, "High availability", elérve: 2024. december 12.

- A rendszer integrációja során képes kell legyen szabványos hitelesítési és engedélyezési protokollokkal való kommunikációra, mint például OAuth2 vagy OpenID Connect.

Üzemeltetési monitoring

- A rendszerhez folyamatos monitorozást kell biztosítani, beleértve a CPU és RAM teljesítményt illetve fogyasztást.
- A rendszer hibajelentést kell megjelnítsen abban az esetben ha leállás vagy hiba történik a rendszerben.

Külső követelmények

GDPR és adatvédelem

- A rendszernek be kell tartania az Európai Unión belüli GDPR szabályozást, a felhasználók adatait a megfelelőképpen kell kezelje és tárolja.
- A rendszer a bőrbetegségek fotóit képes kell legyen biztonságosan kezelni és kiszivárogtatás nélkül feldolgozni. Kell biztosítsa azt, hogy a bőrbetegések fotóit csak az azt felvezető felhasználó láthatja.

Third-party Integration

- A rendszer kompatibilis kell legyen harmadik fél által biztosított API-okkal való kommunikációra, különböző harmadik féltől származó funckionalitások megvalósításához.

5. fejezet

Felhasznált technológiák

5.1. Backend fejlesztésben használt technológiák

Az applikáció backend részének fejlesztése során a következő technológiákat használtam:

5.1.1. Java Spring

A Spring Framework egy átfogó és sokoldalú platform a vállalati Java-fejlesztéshez. Ismert az Inversion of Control (IoC)[1] és Dependency Injection (DI)[2] képességeiről, amelyek megkönnyítik a moduláris és tesztelhető alkalmazások létrehozását. A legfontosabb funkciók közé tartozik a Spring MVC a webfejlesztéshez, a Spring Boot a gyors alkalmazásbeállításhoz, valamint a Spring Security a megbízható hitelesítéshez és engedélyezéshez.

A fentebb felsorolt funckionalitások közül szinte mindegyiket használtam a REST API lérehozása során, ezért is volt fontos, hogy egy ismert és széleskörben használt keretrendszert válasszak és használjak a fejlesztés során.

5.1.2. Spring Data JPA

A Spring Data JPA, amely a nagyobb Spring Data család része, megkönnyíti a JPA-alapú (Java Persistence API) tárolók egyszerű megvalósítását. Megkönnyíti az adatelérési technológiákat használó Spring-alapú alkalmazások készítését.

Egy alkalmazás adatelérési rétegének megvalósítása meglehetősen nehézkes lehet. Túl sok boilerplate kódot kell írni a legegyszerűbb lekérdezések végrehajtásához. Ha hozzáadunk olyan dolgokat, mint a lapozás, az auditálás és más, gyakran szükséges opciók, akkor a végén elveszünk.

Számomra ezért is volt kézenfekvő a Spring Data JPA használata, mivel megkönnyítette a modellezés és a lekérdezések megírásának folyamatát. Az alapértelmezetten ismert,

[1] Az Inversion of Control (IoC) egy olyan programozási elv, amelyben a vezérlési folyamatot az alkalmazás fő logikájából a keretrendszer veszi át. Ez segít a laza csatoltság elérésében és a kód újrahasznosíthatóságában.

[2] A Dependency Injection (DI) egy olyan tervezési minta, amely lehetővé teszi, hogy egy osztály függőségeit külső forrásból (például egy keretrendszer által) injektálják az osztályba. Ez csökkenti a kód csatoltságát és megkönnyíti a tesztelést.

gyakran használt és univerzális lekérdezések egytől-egyig a rendelkezésemre álltak és így hatékonyan tudtam kommunikálni az adatbázissal az adatelérési rétegben. Az osztályaimat könnyedén táblákká tudtam alakítani és nem kellett manuálisan az adatbázisban ezt megtennem.

5.1.3. Mockito

A Mockito egy népszerű Java-alapú tesztelési keretrendszer, amely lehetővé teszi a mock (helyettesítő) objektumok egyszerű létrehozását és kezelését. Kifejezetten hasznos egységtesztelés[3] (unit testing) során, ahol a tesztelendő osztályt izolálni kell a külső függőségeitől.

A keretrendszer egyik legfontosabb előnye az egyszerű használat és az intuitív API, amely jelentősen csökkenti a tesztírás bonyolultságát. A Mockito lehetővé teszi a metódushívások szimulációját és viselkedésük konfigurálását, illetve biztosítja a metódushívások ellenőrzését, hogy egyértelművé váljon, milyen interakciók történtek a teszt során. Ezen felül támogatja a különböző Java-szintaxisokat és integrálható más tesztelési keretrendszerekkel, például a JUnit vagy a TestNG rendszerekkel, amelyek tovább növelik a hatékonyságot.

5.1.4. PostgreSQL

A PostgreSQL egy nagy teljesítményű, nyílt forráskódú objektum-relációs adatbázis-rendszer, amelynek több mint 35 éves aktív fejlesztése során a megbízhatóság, a funkciók robusztussága és a teljesítménye szerzett jó hírnevet.

Az alkalmazás tervezésénél azért esett a választás a PostgreSQL-re, mivel nagyon népszerű és számos segédanyag illetve videó található róla az interneten, valamint előzetes tapasztalataim is voltak az adatbázis-rendszerrel. A Spring Data JPA segítségével könnyedén hozzákapcsolható bármely REST API-hoz. Üzemeltetés és skálázhatóság tekintetéből is előnyös volt a PostgreSQL használata, mivel hatékonyan kezeli a párhuzamos, nagy volumenű tranzakciókat és lekérdezéseket illetve vertikális és horizontális skálázásra is lehetőség van.

5.1.5. Docker

A Docker egy nyílt platform alkalmazások fejlesztésére, szállítására és futtatására. A Docker lehetővé teszi, hogy elválasszuk az alkalmazásokat az infrastruktúrától, így gyorsan szállíthatóvá válik a szoftver. A Docker kódszállításra, tesztelésre és telepítésre szolgáló módszertanainak kihasználásával jelentősen csökkenthetjük a kód megírása és a kód termelésben való futtatása közötti késedelmet.

A Docker használata azért volt kulcsfontosságú a projekt megvalósításában, mert nélküle nem tudunk modern üzemeltetésről beszélni. A konténerizáció[4] a modern szoft-

[3] Az egységteszt a szoftvertesztelés egy olyan típusa, amelynek célja egy adott kódegység – például egy metódus, osztály vagy modul – működésének tesztelése izolált környezetben.

[4] A konténerizáció egy olyan szoftvertelepítési folyamat, amely egy alkalmazás kódját az összes olyan fájllal és könyvtárral együtt csomagolja, amelyre szüksége van ahhoz, hogy bármilyen infrastruktúrán fusson. Hagyományosan bármely alkalmazás futtatásához a számítógépen a gép operációs rendszerének megfelelő verziót kellett telepíteni.

verfejlesztés egyik megkerülhetetlen technológiájává vált és segítségével könnyedén platformfüggetlenné tudtam tenni az alkalmazásomat. A Docker Compose segítségével egy parancs beírásával egyszerűen el tudom indítani a REST API-t az adatbázissal együtt, amelyek egymástól elszigetelve, külön konténerekbe futnak.

5.1.6. Kubernetes

A Kubernetes, más néven K8s, egy nyílt forráskódú rendszer a konténeres alkalmazások telepítésének, skálázásának és kezelésének automatizálására. Az alkalmazást alkotó konténereket logikai egységekbe csoportosítja az egyszerű kezelés és felfedezés érdekében. A Kubernetes a Google-nél a termelési munkaterhelések futtatásában szerzett 15 éves tapasztalatra épül, a közösség legjobb ötleteivel és gyakorlataival kombinálva.

Az alkalmazás üzemeltetése során megkerülhetetlen volt a Kubernetes használata is. A Kubernetesre legfőképp az autoskálázás miatt esett a választás, mivel segítségével a folyamatosan növekvő terhelés alatt az alkalmazás képes több példánnyá skálázódni, ezzel is elkerülve az elhalást. Ha valami hiba lép fel a rendszerben és leáll valamelyik konténer a Kubernetes automatikusan újra tudja indítani a példányt, így minimalizálva a kiesett időt.

5.1.7. Git/GitLab

A jelenlegi felgyorsult szoftverfejlesztésben a hatékony, gördülékeny munka és az egyszerűsített munkafolyamatok alapvető fontosságúak a csapatok számára a kiváló minőségű szoftvertermék előállításához. A GitLab olyan teljes körű elrendezésként jelenik meg, amely egyetlen platformon koordinálja a verziókezelést, a problémakövetést, a folyamatos integrációt/folyamatos telepítést (CI/CD) és az együttműködési eszközöket, lehetővé téve a csapatok számára, hogy zökkenőmentesen kezeljék a teljes DevOps-életciklusukat[5]. A GitLab a legátfogóbb AI-alapú DevSecOps[6] platform.

Számomunkra a GitLab volt a legjobb és legegyszerűbb választás a DevOps metodológia megvalósításához. A platformon a csapatban könnyen tudtuk kezelni a sprinteket és a célkitűzéseket is. A Continuous Integration[7] és Continuous Delivery[8] elveket is könnyedén meg tudtuk oldani a GitLab által szolgáltatott pipeline szolgáltatás segítségével, így nem volt szükség harmadik féltől származó automatizációs szoftverre. Ezek mellett a kód verziókezelése és a csapatmunka támogatása is a GitLab előnyeként említendő.

[5]A DevOps egy olyan átalakító kultúra és gyakorlat, amely egyesíti a szoftverfejlesztő (Dev) és az IT-üzemeltetési (Ops) csapatokat. Az együttműködés elősegítésével és az automatizálási technológiák kihasználásával a DevOps gyorsabb, megbízhatóbb kódtelepítést tesz lehetővé a termelésbe, hatékonyan és megismételhető módon.

[6]A DevSecOps a fejlesztés, a biztonság és az üzemeltetés rövidítése. Ez egy olyan keretrendszer, amely a biztonsági gyakorlatokat a szoftverfejlesztési életciklus minden fázisába integrálja

[7]A folyamatos integráció (CI) egy olyan szoftverfejlesztési gyakorlat, amelyben a fejlesztők gyakran egyesítik kódváltozásaikat egy központi tárolóba. Ez a gyakorlat a DevOps kulcsfontosságú eleme, és célja a szoftverminőség javítása, a kiadási folyamat felgyorsítása és a kódváltozások integrálásának megkönnyítése.

[8]A folyamatos szállítás egy olyan szoftverfejlesztési gyakorlat, amely a kódváltozások automatikus előkészítését jelenti a termelési környezetbe történő kiadáshoz.

5.2. Frontend fejlesztésben használt technológiák

Az applikáció frontend részének fejlesztése során a következő technológiákat használtam:

5.2.1. React Native

A React Native egy nyílt forráskódú keretrendszer, amelyet a Facebook fejlesztett ki, és amely lehetővé teszi mobilalkalmazások fejlesztését JavaScript használatával. A keretrendszer legfőbb előnye, hogy egyetlen kódbázisból képes natív alkalmazásokat létrehozni mind Android, mind iOS platformokra, ezáltal jelentős idő- és költségmegtakarítást biztosít a fejlesztés során.

A React Native alapját a React keretrendszer adja, amely a komponensalapú fejlesztési megközelítést alkalmazza. A fejlesztők a Reactben megszokott JSX[9] szintaxist használhatják, miközben a felhasználói felület natív komponensekké alakul át a platform specifikus megjelenés és működés biztosítása érdekében. Ez azt jelenti, hogy a felhasználók ugyanazt a natív élményt kapják, mint egy hagyományos, platformspecifikusan fejlesztett alkalmazás esetében.

5.2.2. Expo

Az Expo egy nyílt forráskódú eszközkészlet és platform, amely kifejezetten a React Native alapú mobilalkalmazások fejlesztésére készült. Az Expo célja, hogy egyszerűbbé és gyorsabbá tegye a cross-platform[10] alkalmazások fejlesztését, különösen a React Native kezdő és haladó felhasználói számára egyaránt. Az Expo biztosít egy teljes eszközkészletet, amely magában foglal fejlesztési, buildelési, tesztelési és publikálási folyamatokat.

Az Expo ideális olyan projektekhez, ahol gyors prototípusokat vagy egyszerűbb alkalmazásokat kell létrehozni. Gyakran használják startupok és kisebb fejlesztőcsapatok, akiknek fontos a gyors iteráció és a könnyű karbantarthatóság. Az Expo ugyanakkor nagyobb projektekben is megállja a helyét, különösen akkor, ha a fejlesztési folyamat egyszerűsítésére és gyorsítására van szükség.

5.2.3. Typescript

A TypeScript egy nyílt forráskódú programozási nyelv, amelyet a Microsoft fejlesztett, és a JavaScript típusbiztos kiterjesztéseként működik. A TypeScript hozzáad egy típusos rendszert a JavaScripthez, amely erőteljesebb és robusztusabb fejlesztést tesz lehetővé. A TypeScript teljes mértékben kompatibilis a JavaScript-tel, ami azt jelenti, hogy bármely meglévő JavaScript kód TypeScript környezetben is futtatható.

[9] A JSX egy kiterjesztett szintaxis a JavaScriptben, amelyet elsősorban a React keretrendszer használ. Lehetővé teszi, hogy a JavaScript kódban HTML-szerű struktúrákat írjunk, amelyek könnyen olvashatók és karbantarthatók. Bár a JSX hasonlít a HTML-re, valójában JavaScript kifejezésekre fordul le a háttérben.

[10] A cross-platform (többplatformos) fejlesztés azt jelenti, hogy egy alkalmazás egyetlen kódbázisból több különböző platformon (pl. Android, iOS, web) is futtatható. Ez lehetővé teszi, hogy a fejlesztők egyszer írjanak kódot, és azt különböző eszközökön, operációs rendszereken használják, így jelentősen csökkentve a fejlesztési időt és költségeket.

A TypeScript mára a modern frontend fejlesztés egyik meghatározó eszközévé vált. Statikus típusossága[11], fejlesztői élményt javító funkciói és a React keretrendszerrel való szoros integrációja révén a TypeScript nemcsak a kód minőségét növeli, hanem a fejlesztési folyamat hatékonyságát is. A React Native projektekben különösen nagy előny, hogy a TypeScript használatával megbízhatóbb, karbantarthatóbb és biztonságosabb alkalmazások készíthetők.

5.2.4. Firebase

A Firebase egy Google által kifejlesztett mobil- és webalkalmazás-fejlesztési platform, amely számos szolgáltatást kínál a fejlesztők számára, például adatbázisokat, autentikációt, felhőalapú tárolást és értesítési rendszereket. A Firebase célja, hogy gyorsabbá és egyszerűbbé tegye az alkalmazások fejlesztését és üzemeltetését, miközben magas szintű funkcionalitást biztosít.

A Firebase egyes szolgáltatásai közül kiemelkedő szerepet kap a Firebase Cloud Messaging (FCM)[12], amely egy teljes körű, cross-platform értesítési rendszer. Az FCM lehetővé teszi, hogy a fejlesztők könnyen küldjenek push-értesítéseket Android, iOS és webalkalmazásokra. Az FCM segít abban, hogy a felhasználók mindig naprakészen értesüljenek az új információkról, miközben minimalizálja a háttérfeladatok és az alkalmazások közvetlen interakcióinak szükségességét.

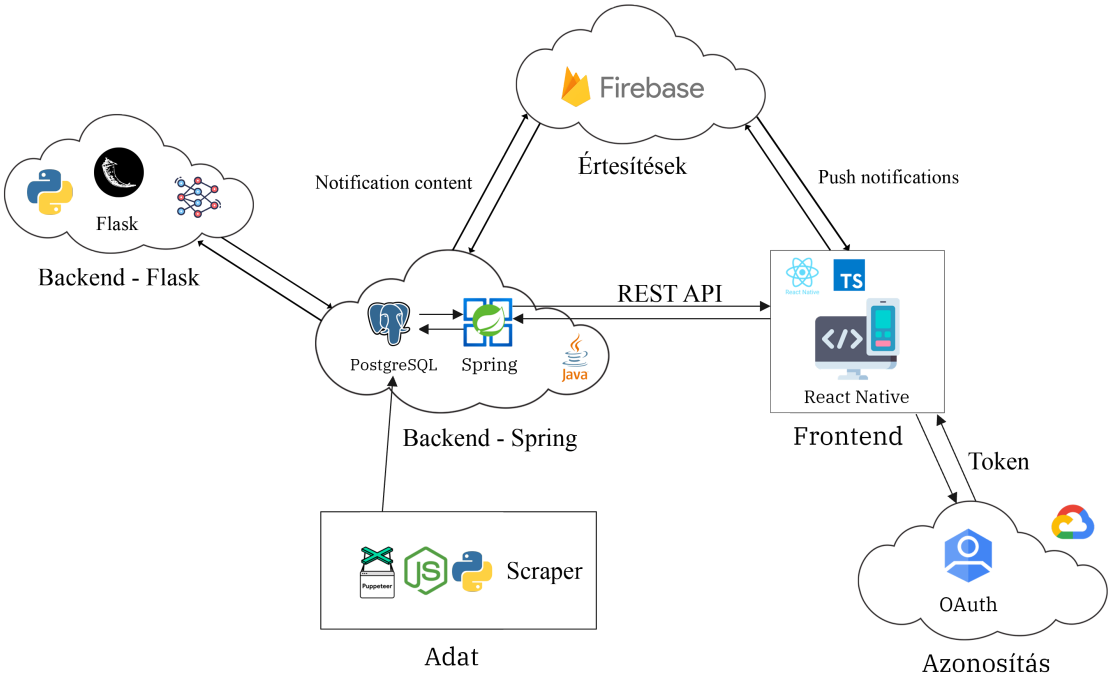
[11] A statikus típusosság egy olyan tulajdonság, amelynél a változók és kifejezések típusát már a fordítási időben meghatározzák, nem pedig futási időben. Ez azt jelenti, hogy a fejlesztőnek előre meg kell adnia, milyen típusú értékekkel dolgozik egy adott változó (például szám, szöveg, boolean stb.).

[12] A Firebase Cloud Messaging (FCM) egy ingyenes, megbízható és skálázható üzenetküldési szolgáltatás, amely lehetővé teszi push-értesítések és adatüzenetek küldését a felhasználóknak

6. fejezet

Megvalósítás

6.1. A rendszer architektúrája



6.1. ábra. Architektúra diagram

Amint az a 6.1-es diagramon látható, a rendszer több, jól elkülönült részből tevődik össze:

1. Java Spring Backend: A fő biznisz logikáért felelős komponens. Ez a webszerver kommunikál a PostgreSQL adatbázissal és a frontend is innen kéri le a megjelenítendő adatokat.

2. React Native Frontend: Az alkalmazás kliens oldala, a felhasználó ezzel a komponenssel interaktál. A mobilaklamazás az egyetlen kliens felület, amely kérések segítségével jeleníti meg a backendtől kapott adatokat.

3. Python Flask Backend: Ez a service felelős az AI model működtetéséért és a predikciók elvégzéséért. Közvetlenül a Flask API mellett található az AI modellünk is, amellyel kommunikál.

4. Adat scraperek: Két scrapert használtam a munkám során. Az első Javascriptben, a Puppeteer könyvtárcsomag segítségével írodott, a másik Pythonban, a Beautiful Soup használatával. Szükséges volt két nyelv használatára, mivel amíg a Beautiful Soup használata közben nem lehet interaktálni a weboldallal adatletöltés során, a Puppeteer csomag ezt lehetővé teszi.

5. Azonosítás: A felhasználók azonosítása a Google segítségével történik a Google Cloud Platformon. Ez a third-party azonosítás megfelelően biztonságos és egyszerű az alkalmazás számára.

6. Értesítések: Az értesítéseket a Firebase platform ütemezi és küldi ki a felhasználók készülékeire, így ez is egy különálló, a felhő alapú komponens.

6.2. Frontend megvalósítása

6.2.1. Design

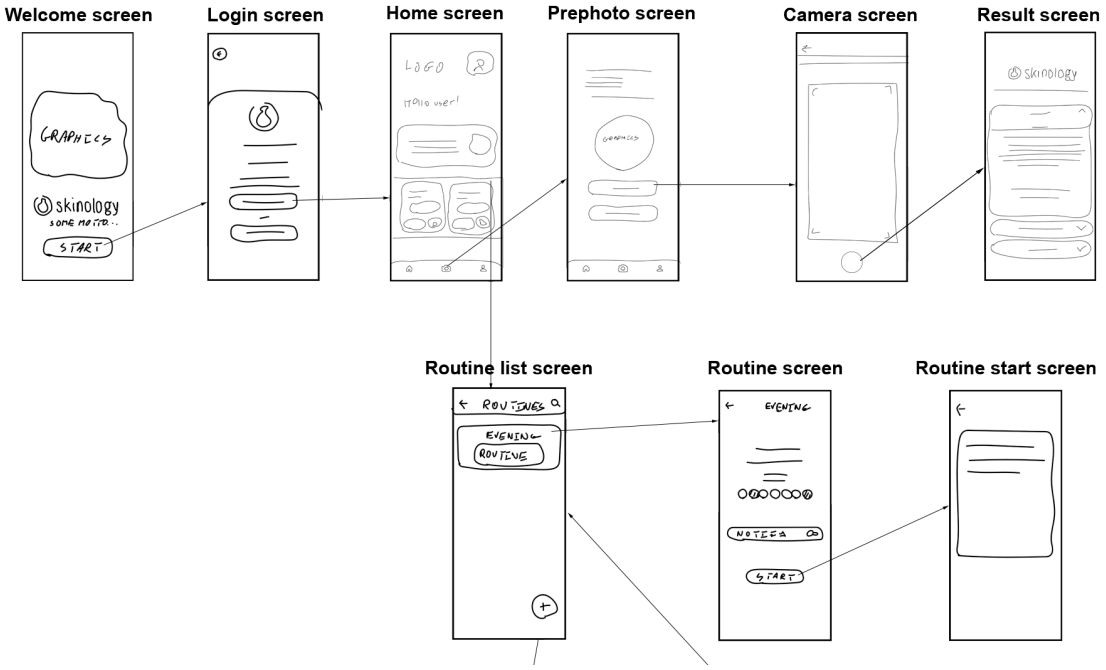
Napjainkban nem lehet úgy mobilapplikációt tervezni és eladni, ha annak nincs egy letisztult, intuitív és magával ragadó designja. Fontos volt számomra, hogy az applikáció szépen mutasson, jól nézzen ki és a fejlesztés folyamán igyekeztem egy egyedi és egységes brandet kialakítani az alkalmazás köré. Ennek eredményeként, nem csak egy alkalmazást csináltam, hanem egy terméket, amellyel felhasználóként jól lehet azonosulni. Az applikáció felületét Figmában terveztem meg és inspirációként a Dribbble oldalt használtam. A fejlesztés során még egy designerrel is egyeztettem, kikértem a tanácsát, hogy mit lehetne a felhasználói felületen módosítani, mivel még vonzobbá szerettem volna tenni az alkalmazást. Számos hasznos véleményt fogalmazott meg, amelyekből én is sokat tanulhattam. Az appliációnak wireframe[1] sémát is készítettem, így designolás során sokkal könnyebb volt az elrendezés és a flow megtervezése (lásd 6.2 ábra).

Lentebb látható a Figma (lásd 6.3) design egy kis része, amellyel szeretném illusztrálni azt, hogy nem csak impulzus designolás folyt az implementáció közben, hanem előre megtervezett és megfontolt design döntések alapján implementáltam az alkalmazás frontend részét.

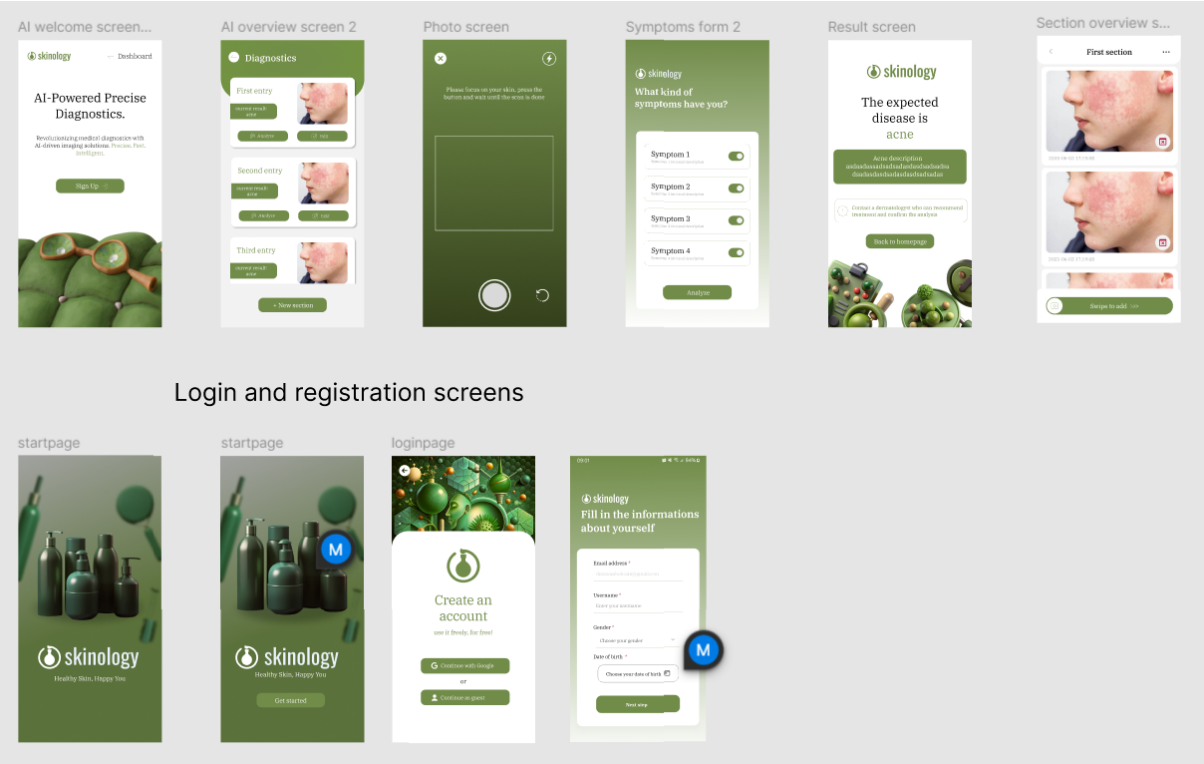
6.2.2. Implementáció

Az alkalmazás fejlesztéséhez olyan technológiára volt szükség, amely lehetővé teszi a cross-platform mobil alkalmazás fejlesztését, így megkönnyítve a kódbázis duplikálását platform specifikus kódokra. Ezen elvárásnak a React Native tökéletesen megfelelt, mivel a segítségével könnyedén tudtam buildelni Android illetve iOS készülékekre is. E mellett egy nagyon felkapott és keresett technológiáról van szó, így nagyon sok forrásból tudtam információt is szerezni róla, ha esetleg valahol elakadtam vagy kérdésem volt. Főbb előnyei közé tartozik a személyre szabott, újrafelhasználható komponensek definiálása, illetve a

[1] Alacsony részletességű vázlat, amely az alkalmazás elrendezését illetve szerkezetét mutatja be.



6.2. ábra. Wireframe



6.3. ábra. Figma design

számos beépített könyvtár támogatása is, amelyek gyorsították a fejlesztési folyamatot, mivel nem kellett mindent nulláról megírnom. Programozási nyelvként a TypeScryptet választottam, mivel a típusellenőrzés kompilálási időben segítette a munkámat, ha esetleg valahol valamilyen hiba volt könnyen lehetett debuggolni a típusdefiniálások miatt. A projekt struktúráját tekintve az Atomic Design Patternt követtem. Mivel a React úgy van megalkotva, hogy kis és könnyen újrahasznosítható komponensekből tevődik össze a felhasználói felület, az Atomic Design Pattern egy tökéletes választás a kis alkotóelemek strktúrálására. Így a komponenseimet három kategóriára osztottam fel, amelyeket a későbbiekben részletezni fogok:

1. Atomok
2. Molekulák
3. Organizmusok

A mobilalkalmazás forráskódját a következő részekre bontottam le:

- *api*: Itt találhatóak az Axios[2] HTTP kérések, amelyek segítségével kommunikálni tud a mobilapplikáció a backenddel, illetve a Google Cloud Platoformmal az azonosítás végrehajtásáért. Ezekben a függvényekben van meghatározva az, hogy az adott kérés **GET, POST, PUT** vagy esetleg **DELETE** típusú, itt kerül átadásra az azonosításért felelős Bearer token is, illetve e mellett meghatározásra kerül az az endpoint, amivel kommunilálnia kell a frontendnek. A 6.1 kódrészlet egy példát mutat be egy GET Axios használatra.

```
async getRoutinesByUserId(
userId: number | null,
token: string | null,
): Promise<Routine[] | null> {
    return axios({
      method: 'get',
      url: `${BASE_URL}${ROUTINE_BY_USERID(userId)}`,
      headers: {
        Authorization: `Bearer ${token}`,},})
      .then(response => response.data)
      .catch(error => {
        console.log(error); });}
```

6.1. kódrészlet. Példa Axios fetch hívásra

- *assets*: Az eszközök mappában a használt betűtípusok, képek, ikonok illetve a töltőképernyőhöz használt animáció fájlja található. Az alkalmazásban egységesen az IBMPlexSerif betűtípust használtam. Animációk létrehozásáért a LottieFiles oldalt használtam, amelyen könnyedén, semmilyen grafikus ismeret nélkül tudtam összeraknie egy mutatós, saját animációt.

[2]Az Axios egy Javascript HTTP-kliens, amely főként Promise-alapú aszinkron kérések küldésére alkalmas kliens oldalon.

- *components*: A fentebb említett Atomic Design Pattern fájlstruktúrája tartozik ide. Három kategóriába vannak besorolva a komponenseim: atoms, molecules illetve organisms. Az atomok között egyszerű, kis, személyre szabott komponensek vannak, mint például a *BackButton* vagy a *HorizontalLine*. A molekulák között már egy fokkal összetettebb és funckionalitásokkal rendelkező komponensek vannak, mint például a *MultiProgressBar*, amely egy diagramot rajzol ki a felhasználói felületre. Az organizmusok a legkomplexebb, legösszetettebb komponensek, amelyek több kisebb molekulából vagy atomból épülnek fel. Ide tartozik például a *Header* vagy a *ScanResultModal*. A design pattern betartásának segítségével jól struktúrált a komponenseim elrendezése és ha navigálni kell, könnyen tudom azt, hogy melyik mappába kell keresgélnem. A 6.2 kódrészletben egy személyre szabott vissza gomb látható, amely több képernyőn is meghívásra kerül, így nem kell minden alkalommal összerakjam a gombot. Próbáltam minél jobban testreszabhatóvá tervezni a személyre szabott komponenseimet, hogy minden use-caseben felhasználható legyen. Ez esetben a gomb méreteit, stílusát, illetve színét is lehet specifikálni, mivel volt olyan alkalom, ahol fehér kellett legyen a gomb és volt ahol fekete.

```
const BackButton: React.FC<BackButtonProps> = ({dark, height = 30,
width = 30, style,}) => {
  const navigation = useNavigation<NavigationProp>();
  return (
    <TouchableOpacity onPress={() => navigation.goBack()}
        style={style}>
      {dark ? (
        <Image source={BACK_DARK} style={{ width, height }} />
      ) : (
        <Image source={BACK} style={{ width, height }} />
      )}
    </TouchableOpacity>);};
```

6.2. kódrészlet. Példa egy atomra

- *constants*: Itt tárolom azokat az adatokat, amelyek az alkalmazás működése során nem változnak, mint például a statikus szövegek, számértékek - amelyeket a dizájnban használtam, színeket és képfájlokat. Előre terveztem, amikor elkezdtem az alkalmazás fejlesztését és azért vannak a szövegek konstans változókban helyezve, hogyha a jövőben majd többnyelvűsíteni szeretném az alkalmazást, akkor könnyedén meg tudjam valósítani és ne kelljen a kódban írni át a hardcodeolt szövegeket. A 6.3 kódrészlet a képek importálását szemlélteti, ami segítségével nem kell a képernyő komponenseben hosszú útvonalakat deklaráljak, hanem egyszerűen csak a **ROUTINE BACKGROUND** konstans értékre tudok hivatkozni.

```
export const ROUTINE_BACKGROUND =
    require('../assets/images/routine-background.png');
```

6.3. kódrészlet. Példa egy kép importálására

- *models*: Ebben a mappában vannak deklarálva a frontenden használt model osztályok. Amikor a backendtől adatot kérek le minden alkalommal átalakítom az adott objektum típusára és utána kezdek dolgozni vele. A Typescript kompilálási időben végzett típusellenőrzése számos alkalomkor megkönnyítette a dolgomat, amikor fetchelni kellett majd megjeleníteni, mert automatikusan észleli azt, hogyha az objektum olyan mezőjére szeretnék hivatkozni megjelenítéskor, ami a model osztályban nem létezik. A 6.4 kódrészletben az alkalmazásban használt összetevők model osztálya látható, amelynek több tulajdonsága is van. A típusellenőrzés segítségével hiába próbálnék hivatkozni például egy összetevő kémiai vegyjelére - ha az nincs deklarálva az osztályban, kompilálási hibát kapnék.

```
export type Ingredient = { name: string; description: string;
  color: string; functionalities: string[]; };
```

6.4. kódrészlet. Példa egy model osztályra

- *repository*: A repositotyban található interfacekben vannak definiálva azok a metódusok fejlécei, amelyeket az api hívásoknak kell implementálniuk. Ez azért előnyös, mert elválasztja az üzleti logika részét az adateléréstől, így nem fogg fügni a logika közvetlenül az adatbázistól.

- *screens*: Azok a képernyő komponensek vannak ebben a mappában, amelyekkel a felhasználó közvetlen interaktál. Ezekre a komponensekre kerülnek fel a személyre szabott, kisebb komponensek, amelyek a components mappában vannak tárolva és itt hívódnak meg az api mappában deklarált Axios HTTP kérések is, ha a felhasználó rámegy egy gombra vagy tovább navigál egyik képernyőről a másikra. Ezek a komponensek jelentik a frontend törzsét, ezek a legngyobbak és legkomplexebbek. Minden egyes képernyőnek saját mappája van, amin belül található maga a képernyő komponens, illetve a hozzá tartozó stílus fájl. Előnyösnek találtam a stílus és a komponens szétválasztását, mert egyes képernyők esetében nagyon a stílusfájl, ezért nehéz lenne a komponensen belüli navigálás, ha még a stílus is ott lenne meghatározva. Az alábbiakban részletesen ismertetem a **HomeScreen** komponens felépítését:

  A 6.5 kódrészlet a HomeScreen komponens első része. Itt vannak deklarálva a szükséges változók a komponens helyes működése érdekében. A useNavigation() hook[3] az alkalmazáson belüli navigációért felelős. A useUser() egy általam definiált React hook, amely segítségével az AsyncStorageból elérem a Google által visszaadott felhasználói adatokat és dolgozni tudok velük. A userData változóban a saját adatbázisomban tárolt felhasználói adatok vannak. A loading változó a komponens betöltését és az adatok lekérését figyeli, míg az error az esetleges hibáknál kap értéket. Az allRoutines stateben az adott felhasználó rutinjai fognak tárolódni, amelyeket majd a képernyőn fogok megjelníteni. Az egyszerű state management érdekében a React beépített useState hookját használtam, amely segítségével könnyedén tudom az állapotokat tárolni.

[3]A React Hook egy speciális JavaScript függvény, amely lehetővé teszi a React funkcionális komponensek számára, hogy állapotot (state) és más React-funkciókat használjanak osztálykomponensek nélkül.

```
export const HomeScreen = (): React.ReactElement => {
  const navigation = useNavigation();

  // user = Google user, setUser = set Google user, isLocalBuild
     = boolean
  const { user, setUser, isLocalBuild } = useUser();
  // userData = user data from database
  const [userData, setUserData] = useState<User | null>(null);

  // variables for fetching the routines
  const [allRoutines, setAllRoutines] = useState<Routine[]>([]);
  const [loading, setLoading] = useState(true);
  const [error, setError] = useState<string | null>(null);
```

6.5. kódrészlet. A HomeScreen komponens első része

A 6.6 kódrészletben a useEffect beépített hook segítségével, miután betöltődik a képernyő a React automatikusan lefuttatja a getUserData függvényt. Az isLocalBuild globális változóban **development** vagy **production** értékek lehetnek, amelyeket környezeti változóból állítok be az alkalmazás indulásánál. Ez azért kedvező számomra, mivel ha development környezetben vagyok, akkor ha valami nem működik Snackbar[4] formájában értesítést kapok a kijelzőn és könyebben megy a debuggolás. A getUserData függvényben a Google által visszaadott felhasználó segítségével lekérem a saját adatbázisomban tárolt felhasználó adatait és beállítom a megfelelő statet, így minden információt megkapok a felhasználóról. Ezután beállítom a felhasználó eszközét is, ha korábban még nem használta az alkalmazást az adott telefonon, hogy a későbbiekben tudjam, hogy melyik eszközökre kell pontosan értesítést küldeni a bőrápolási rutin kezdése előtt. A kommunikálás magas hibatűréssel rendelkezik, mivel minden HTTP kérés hibakezeléssel van körülvéve.

```
useEffect(() => {
  if (isLocalBuild) {
    try { getUserData(); }
  } else { getUserData(); } }, []);

const getUserData = async (): Promise<void> => {
  const registerController = new UserApiCalls();
  const token = await getToken();
  let deviceToken;
  try { deviceToken = await getDeviceToken(); }
  registerController.getUserByEmail(user?.email,
      token).then(data => {
    setUserData(data); });
  if (deviceToken && !userData?.devices.includes(deviceToken)) {
    const devices = userData?.devices;
    devices?.push(deviceToken);
```

[4] A Snackbar egy rövid, ideiglenes értesítés, amely a képernyő alsó részén jelenik meg, és információkat közöl a felhasználóval.

```
        try {
          await registerController.updateDeviceToken(
            userData?.id,
            userData?.email,
            userData?.username,
            userData?.dateOfBirth,
            userData?.gender,
            devices,
            token,
          ); } } };
```

6.6. kódrészlet. A HomeScreen komponens második része

A 6.7 kódrészletben a felhasználó bőrápolási rutinjait kérem le a useFocusEffect hook és a fetchData függvény segítségével. Azért volt előnyösebb itt useFocusEffect hookot használni a useEffect helyett, mivel a useEffect csak a renderelését észlelte a kijelzőnek, míg a useFocusEffect minden navigálást, így mikor létrehoztam egy rutint és visszanavigáltam a főképernyőre, akkor nem láttam az új rutint megjelenítéskor, mivel a useEffect nem kérte le újra a felhasználó rutinjait. A useFocusEffect megoldotta ezt a problémát.

```
      useFocusEffect(
        useCallback(() => {
          fetchData(); }, [fetchData]), );
```

6.7. kódrészlet. A HomeScreen komponens harmadik része

A 6.8 kódrészletben a töltés és hibafigyelés látható. Amíg adatot kérek le, beállítom a loading változót, így az animáció fog megjelenni a képernyőn, ezzel is biztosítva egy jobb felhasználói élményt. Ha valami hiba kerül a rendszerbe, akkor egy error képernyő jelenik meg a felhasználónak, ami részletezi a hiba okát.

```
      if (loading) {
        return (
          <View style={styles.loadingOverlay}>
            <LottieView
              source={LOADING_ANIMATION}
              autoPlay loop
              style={{ width: 200, height: 200, }} />
        </View> ); }
```

6.8. kódrészlet. A HomeScreen komponens negyedik része

A következő, 6.9 részben a tényleges renderelés látható, amelyet a felhasználó is lát. Itt kerül beállításra a képernyő header része, a profilkép megjelenítése, egy köszöntő a felhasználó számára, illetve egy shortcut gomb, amely az alkalmazás egyik fő funkcionalitásához vezet. Próbáltam odafigyelni arra is, hogy minél könyebben kezelhető és intuitívabb legyen a felület, ezért használtam shortcut gombokat is, hogy a felhasználó egyből és a lehető legegyszerűbben tudja elérni azt a funkcionalitását az applikációnak amit éppen szeretne. Abban az esetben, amikor nem

annyira komplex stílust kell alkalmazni egy komponensre, akkor inline deklarálom a stílusát, ha komplexebb és több sort igénylő stílusdefiniálás szükséges akkor azt a képernyő komponenshez tartozó stílusban definiálom, ahogy az a 6.10 részletben látható. Az alkalmazás layoutjának definiálásakor az adott készülék méreteit használtam a **Dimensions.get('window')** segítségével, így bármelyik készüléken pontos és rendezett lesz a layout, nem fog elcsúszni.

```
      return (
          <ScrollView contentContainerStyle={styles.container}>
            <View style={styles.headerView}>
              <Image source={HOME_SCREEN_LOGO} style={styles.logo} />
              <TouchableOpacity
                onPress={() => {
                  navigation.navigate('Profile' as never); }}>
                  <Image
                    source={{ uri: user.picture }}
                    style={styles.profilePicture} />
              </TouchableOpacity>
            </View>
            <View style={styles.welcomeHeaderView}>
              <View>
                <Text style={styles.welcomeText}>Hello </Text>
                <Text
                  style={[styles.welcomeText, { fontFamily:
                      'IBMPlexSerifSemiBold' }]}>
                  {userData === null ? 'Guest!' : userData?.username}
                </Text>
              </View>
            </View>

            <BigCardWithImage
              navigateTo={'PrePhoto' as never}/>
```

6.9. kódrészlet. A HomeScreen komponens ötödik része

```
      const height = Dimensions.get('window').height;
      const width = Dimensions.get('window').width;

      export const styles = StyleSheet.create({
        headerView: {
          flex: 1,
          flexDirection: 'row',
          alignItems: 'flex-start',
          paddingHorizontal: 48,
          paddingTop: 48,
          marginTop: 32,
          justifyContent: 'space-between',
          width: '100%'},
```

6.10. kódrészlet. A HomeScreen komponens stílusának egy része

A 6.11 kódrészletben a felhasználó rutinjainak megjelenítése látható. A rutinok egy **HorizontalScrollView** custom komponensben vannak megjelenítve, amely lehetővé teszi a minél jobb személyreszabhatóságot. Abban az esetben ha a felhasználónak még semmilyen rutinja sincs létrehozva egy shortcut gomb segítségével átnavigálhat direkt a rutin létrehozás funckionalitáshoz. Ezzel a placeholderrel nemcsak a design lesz szebb, de a felhasználói élmény is sokkal gördülékenyebb lesz. Amennyiben a felhasználó csak vendégként lép be az alkalmazásba a 6.12 kódrészletben látható, hogy arra próbálom sarkallni, hogy jelentkezzen be az alkalmazásba és használja a rutin menedzsment funckionalitást.

```
{userData !== null ? (
    <View style={styles.routineView}>
      {allRoutines.length > 0 ? ( <>
          <View style={styles.routineTextView}>
            <Text style={styles.routineForYou}>
                {HOME_SCREEN_ROUTINE_FOR_YOU_TEXT}</Text>
            <TouchableOpacity
              onPress={() => { navigation.navigate('Routines' as
                  never);}}>
              <Text style={styles.routineViewMore}>
                {HOME_SCREEN_ROUTINE_VIEW_MORE_TEXT}{' '}
              </Text>
            </TouchableOpacity>
          </View>
          <View style={styles.scrollView}>
            <HorizontalRoutineScrollView routines={allRoutines} />
          </View> </>) : <BigCardWithImage
         navigateTo={"RoutineCreate" as never}/>}
    </View>)
```

6.11. kódrészlet. A HomeScreen komponens hatodik része

```
<View>
  <HorizontalLine color={Colors.MOSS_GREEN} width={250}
      height={0.5} />
  <View style={styles.photoShortcutView}>
    <TouchableOpacity
      onPress={() => { navigation.navigate('Login' as never); }}>
      <View style={styles.photoButtonView}>
        <View style={styles.photoButtonTextView}>
          <Text style={styles.photoButtonText}>
            {HOME_SCREEN_LOGIN_BUTTON_BIG_TEXT} </Text>
          <Text> {HOME_SCREEN_LOGIN_BUTTON_SMALL_TEXT} </Text>
          <HorizontalLine color={Colors.MOSS_GREEN} width="100%"
              height={0.5}/>
```

```
            </View>
            <Image source={HOME_SCREEN_LOGIN_BUTTON_GRAPHICS_IMAGE}/>
          </View>
        </TouchableOpacity>
      </View>
    </View> )}
```

6.12. kódrészlet. A HomeScreen komponens hetedik része

- *service*: Abban az esetben ha komplexebb üzleti logika végrehajtása volt szükséges frontend oldalon, akkor azt a service mappában deklaráltam. Itt található például az összetevő szkennelés funckionalitás logika része, amely felismeri a képen lévő szöveget és regexek segítségével kiveszi a felismert szövegből a számomra releváns részt.

- *storage*: A felhasználó adatait az AsyncStorage könyvtár segítéségvel tárolom frontend oldalon, hogy bárhol és bármikor hozzá tudjak férni akármelyik tulajdonságához. E mellett a Google által visszatérített token is itt tárolódik bejelentkezés után, hogy bármelyik fetch kérésben könnyedén a tokenhez tudjak jutni és le tudjam kérni (lásd 6.13 kód).

```
export const storeToken = async (token: string): Promise<void> =>
    {
  try {
    await AsyncStorage.setItem(TOKEN_KEY, token);
  } catch (error) { console.error('Failed to store token',
      error);} };

export const getToken = async (): Promise<string | null> => {
  try { return await AsyncStorage.getItem(TOKEN_KEY);
  } catch (error) {
    console.error('Failed to get token', error);
    return null; } };
```

6.13. kódrészlet. Példa a token tárolására

6.2.3. Autentikáció

A felhasználók nyilvántartása érdekében egy bejelentkezési rendszert kellett kialakítani az alkalmazásban. Mivel nem szerettem volna saját implementációval ellátott regisztrációs illetve bejelentkezési rendszert, így a Google által ajánlott OAuth 2.0[5] megoldást választottam. A Google Cloud Platformon létrehozott **client id**-t regisztráltam a kliens oldali kódbázisban, így meg volt az összeköttetés a két fél között. Az én esetemben nem volt fontos az, hogy jogosultságokat is tudjon kezelni a bejelentkezési rendszer, mivel csak vendég és bejelentkezett felhasználó szerepkörök vannak az applikációban, így nem

[5]Az OAuth 2.0 egy nyílt hitelesítési (auth) és engedélyezési (authorization) protokoll, amely lehetővé teszi, hogy egy alkalmazás hozzáférjen egy másik szolgáltatás erőforrásaihoz anélkül, hogy a felhasználó jelszavát közvetlenül kezelné.

kell admin vagy esetleg superadmin jogosultságokat kezelni. A bejelentkezési képernyőn a *Continue with Google* gombra rányomva és a megfelelő e-mail címet kiválasztva az Expo Auth Session csomag segítségével az alkalmazás lekéri a Google által tárolt felhasználó adatait illetve megkapjuk az **accessToken**-t is. A token kulcsfontosságú az alkalmazás további működésében, mivel nélküle nem lehet egy HTTP kérést sem küldeni a szerveroldalnak. A 6.14 kódrészlet szemlélteti a Google és az alkalmazás közötti kommunikációt.

```
const [_, response, promptAsync] = Google.useAuthRequest(config);
```

6.14. kódrészlet. Googleval való kommunikáció

A lenti 6.15 kódrészletben látható az a folyamat, amikor a Google által visszatérített access tokent tárolom el a felhasználó készülékének lokális tárhelyében, az Async-Storage könyvtárcsomagot használva. Miután a metódus meghívódik, a bejelentkezés befejeződik és bejelentkezett felhasználó lesz a jogkör. Amennyiben kéréseket szeretnénk küldeni a szerveroldal felé a 6.16 kódrészletben látható módon, az Axios kérés header részébe definiálni kell az előzetesen eltárolt access tokent a **Bearer** kulcsszóval, mert hanem a szerveroldal **401 UNAUTHORIZED** hibát fog visszaadni. Ha a token szerepel a kérés fejlécében, akkor szerveroldalon elsődlegesen egy ellenőrzés történik, hogy a token valid-e? A Google által szolgáltatott **https://www.googleapis.com/oauth2/v1/tokeninfo?access_token=<YOUR-TOKEN>** URI-t meghívva és átadva a kliens oldalról kapott tokent validálni tudjuk annak helyességét. Ha a válasz **200**-as kóddal jön vissza a backendre, akkor elkezdjük a business logikát.

```
const handleToken = useCallback(async (): Promise<void> => {
const storedToken = await getToken();
if (!storedToken) {
  if (response?.type === 'success') {
    const { authentication } = response;
    const token = authentication?.accessToken;
    if (token) {
      await storeToken(token);
      await getUserInfo(token);
    } } } else {
  await getUserInfo(storedToken); }
}, [response, navigation]);
```

6.15. kódrészlet. A Google által visszatérített access token tárolása

```
return axios({
  method: 'get',
  url: `${BASE_URL}${GET_DISEASE_BY_ID(id)}`,
  headers: {
    Authorization: `Bearer ${token}`, }, }).then(response => response.data)
```

6.16. kódrészlet. Az access token átadása HTTP kérés esetén

Kijelentkezéskor a 6.17 kódrészletben látható függvény hívódik meg kliens oldalon, amely a **https://oauth2.googleapis.com/revoke** végpontot meghívva és átadva annak a még érvényben lévő access tokent, visszavonja annak érvényességét.

```
export const revokeGoogleToken = async (token: string): Promise<void> => {
    await revokeAsync({ token }, { revocationEndpoint:
        REVOKE_GOOGLE_TOKEN_URL });};
```

6.17. kódrészlet. Token visszavonása kijelentkezés esetén

A Google egy biztonságos és gyors autentifikációs rendszert kínál, amely segítségével könnyedén és egyszerűen tudom számontartani a bejelentkezett felhasználók hitelességét és validálni tudom őket. E mellett az első alkalommal, amikor a felhasználó be szeretne lépni az applikációba egy regisztrációs űrlapon további, többletinformációkat kér tőle az alkalmazás, mint például a neme, és a születési dátuma. Ezek az információk a saját adatbázisomban vannak tárolva. A további fejlesztési lehetőségek érdekében hasznosnak találtam egy személyre szabott űrlap bevezetését az első bejelentkezésnél, amit majd bővíteni tudok olyan információk bekérésével, amelyek segítenek a jövőben új funkcionalitások implementálásában (például a felhasználó bőréről szerzett információk, illetve az étrendje, amely nagy szerepet játszik a bőrápolási szokások kialakításában).

6.3. Üzemeltetés

6.3.1. Konténerizáció

Nem beszélhetünk üzemeltetésről konténerizáció nélkül, így az első lépés a szerveroldal konténerizációja volt a deployment munkafolyamatban. Eszközként a Dockert használtam, mert széles körben ismert és megbízható technológiáról van szó. A Java Spring REST API szerveroldali kódomat egy Dockerfile segítségével gyorsan és könnyedén tudtam konténerizálni. Amint az a 6.18 kódrészletben is látható multi-stage[6] Dockerfilet használtam, hogy minél jobban tudjam leredukálni a végső image méretét. Egy egyszerű gradle base imaget használva buildeltem le az első szakaszban az alkalmazást, majd másoltam át a végterméket a második szintre. A kötelező frissítések és healtcheck után, a 8080-as portot kiterjesztve futtatni is lehet a REST API-t. Mivel lokálisan egy PostgreSQL adatbázissal együtt futtatom a szerveroldali alkalmazást, így egy docker-compose fájlt is deklarálni kellett, amelyben az adatbázis is szerepel. Éppen ezért kellett a Spring Dockerfileban a **waitforpostgres.sh**, általam létrehozott bash scriptet is átmásolni a konténerbe, mivel ennek segítségével tudja megvárni a Spring, míg elindul előtte az adatbázis, enélkül hibát adna a Hibernate a csatlakozáskor.

```
FROM gradle:8.10.2-jdk21 AS builder
WORKDIR /app
COPY build.gradle .
COPY settings.gradle .
COPY src/ src/
RUN gradle build
```

[6]A multi-stage Dockerfile egy olyan Dockerfile, amely több építési lépést (stage-et) használ a végső konténer méretének csökkentésére és hatékonyabbá tételére.

```
FROM gradle:8.10.2-jdk21
WORKDIR /app
COPY --from=builder /app/build/libs/*.jar app.jar
EXPOSE 8080
RUN apt-get update && apt-get install -y curl bash netcat && apt-get clean
HEALTHCHECK CMD curl -f http://localhost:8080/health || exit 1
COPY wait_for_postgres.sh .
RUN chmod +x wait_for_postgres.sh
RUN apt-get update && apt-get install -y dos2unix && apt-get clean
RUN dos2unix wait_for_postgres.sh
ENTRYPOINT ["java", "-jar", "app.jar"]
```

6.18. kódrészlet. Java Spring Dockerfile

Mindezek után a kész Dockerfilet a 6.19 parancsokkal lehet buildelni és futtatni. Ha az adatbázisra is szükségem van, akkor a docker-compose fájlt a 6.20 paranccsal tudom egyszerre buildelni és futtani is.

```
docker build -t r.edu.codespring.ro/borgyogyasz-app/borgyogyasz-backend .
docker run -p 8080:8080
   r.edu.codespring.ro/borgyogyasz-app/borgyogyasz-backend
```

6.19. kódrészlet. Docker build és futtatás

```
docker-compose up --build
```

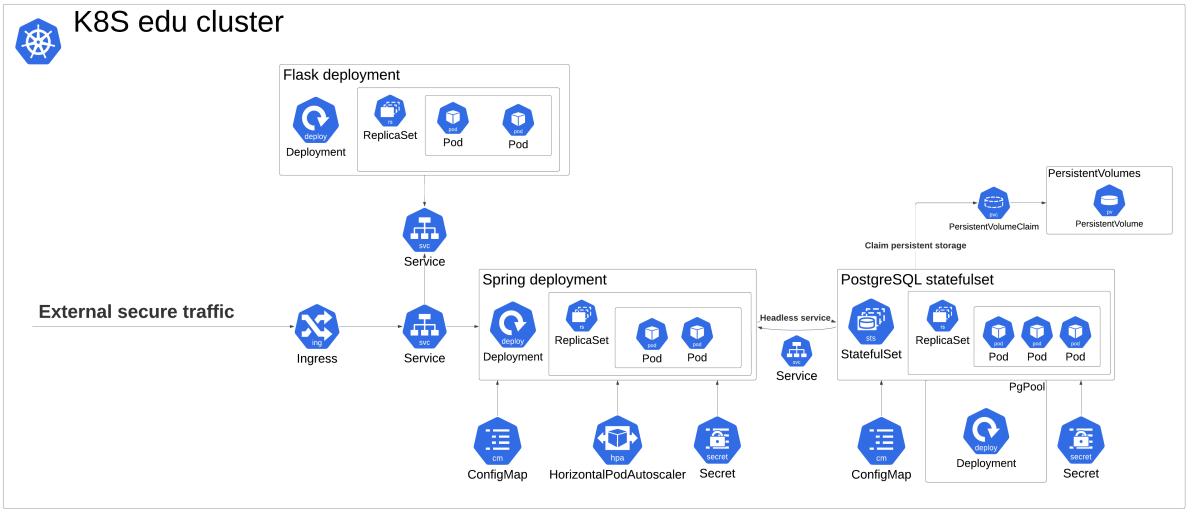
6.20. kódrészlet. Docker compose futtatás

Hasonlóképpen van megoldva a Python Flask API-nál is a konténerizáció, egy Python base image segítségével építem fel a Dockerfilet, amibe átmásolom a forráskódot. A Flask API-t a Gunicorn segítségével tudom futtani, amire azért van szükség mert a Flask beépített fejlesztői szervere nem alkalmas nagy terhelés alatti, productionben való futtatásra. A Gunicorn lehetővé teszi a többszálúsítását is a Flask API-nak. A forráskód mellett itt szükséges az AI model átmásolása is a konténerbe. Az AI backend konténerizációjánál fontos szempont volt a végső image méretének a leredukálása a lehető legjobban, mivel például a pythonban használt tensorflow könyvtár illetve az AI model nagyon sok helyet folglal. A méret mellett az erőforrásigénye is nagyon nagy volt eleinte a futó konténernek, ezért ezt is elővigyázatosan és optimalizálva kellett kezelni.

6.3.2. Production environment

Napjainkban a stabilitás, hibatűrés illetve skálázhatóság kulcsfontosságú szerepet játszik az alkalmazások fejlesztésében, ezért is kapott kiemelt fontosságú szerepet az üzemeltetés megvalósítása a fejlesztés során. A mai szoftveriparban nem csak a fejlesztői tapasztalatok és tudás számít sokat, hanem az üzemeltetéssel járó problémamegoldás is.

A kliensoldali APK egyszerűen hozzáférhető azon felhasználók számára, akiknek továbbítom vagy elküldöm az APK fájlt, így könnyedén tudják telepíteni a saját készülékeikre.



6.4. ábra. Kubernetes architektúra

A szerveroldali üzemeltetés - ahogy az a 6.4 diagramon is látható egy fizikai Kubernetes cluster[7] segítségével valósult meg. A Kubernetes egy megkerülhetetlen technológiává vált az évek során, ha üzemeltetésről vagy DevOps-ról van szó, ezért is választottam azt, hogy egy ilyen clusterre szeretném kitelepíteni az alkalmazást. Ez a technológia lehetővé teszi a szerveroldal rugalmas skálázását a terhelés változásának függvényében és automatikus hibakezelést is végez. Az alkalmazás kitelepítéséhez Helm chartokat[8] hoztam létre, amelyek segítségével egyszerűen, csupán egy parancsot használva tudtam az egyes komponenseket kitelepíteni a clusterre.

Amint az a 6.21 kódrészletben is látható, a Java Spring REST API egy Kubernetes **Deployment** keretében fut. Elsődlegesen két futó **Pod**-ot indít, amelyek ugyanazt a Docker imaget használják. A deployment fájlban szükséges beállítani a konténerek számára a környezeti változókat is, mert ezek nélkül nem tudnák elérni az adatbázist a clusterben. Ezek mellett különböző livenessProbe és readinessProbe utasításokat is tartalmaz a yaml fájl, amelyek az egészséges futtatásért vannak definiálva. A podok erőforrásmeghatározása is ebben a fájlban van, ahogy az a 6.22 kódrészletben látható, a szükséges és maximális CPU illetve memória deklarálásával szabályozni tudom az erőforrásigényüket a konténereknek. Egy pod elindításához legalább **512 MB RAM** és **0.5 CPU mag** szükséges. E mellett **HorizontalPodAutoscaler**-t is használok, amelynek az a feladata, hogy észlelje a növekvő terhelést és abban az esetben ha már nem elég a kezdetben elindított két pod a kérések kiszolgálására automatikusan, illetve horizontálisan skálázza a szerveroldalt. Az autoscaler akár tíz pod példányt is el tud indítani párhuzamosan, amelyek egyidejűleg tudják kiszolgálni a bejövő kéréseket. Akkor indít a scaler új podot, ha a CPU használat a meglévő podok összesített átlagában 40%-ra emelkedik. Ha a CPU használat alacso-

[7] A K8s cluster (Kubernetes cluster) egy konténerorchesztrációs rendszer, amely segít kezelni és méretezni a konténerizált alkalmazásokat. Egy Kubernetes klaszter több csomópontból (node) áll, és biztosítja az alkalmazások automatikus telepítését, skálázását és kezelését.

[8] A Helm Chart egy csomag, amely tartalmazza egy Kubernetes-alkalmazás összes erőforrásának (pl. Deployment, Service, ConfigMap) sablonját és konfigurációs fájljait.

nyabb, mint a kívánt **40%**, az autoscaler csökkentheti a podok számát a minReplicas: 1 értékhez.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: spring-backend-deployment
  namespace: skinology-ns
  labels:
    app: spring-backend
spec:
   replicas: 2
```

6.21. kódrészlet. Kubernetes Spring Deployment

```
resources:
    requests:
      memory: "512Mi"
      cpu: "500m"
    limits:
      memory: "1Gi"
      cpu: "1000m"
```

6.22. kódrészlet. Kubernetes Spring Deployment resource specification

Mivel közvetlenül a deployment által létrehozott podokkal nem tudunk kommunikálni a clusterben, szükséges egy **Service** létrehozása is, amelyen keresztül kéréseket tudunk irányítani a podok felé. A 6.23 kódrészletben látható egy service deklarálása, amely egy ClusterIP típusú service. Ezáltal bármely, a cluster belsejében lévő alkalmazás el tudja érni a Java Spring REST API endpointjait. A service egy TCP típusú kommunikációt biztosít, viszont azt, hogy a Java Spring deployment melyik podjához továbbítja a kérést, azt nem tudjuk, ezt a Kubernetes dönti el futásidőben.

```
apiVersion: v1
kind: Service
metadata:
  name: spring-backend-service
  namespace: skinology-ns
spec:
  selector:
    app: spring-backend
  ports:
    - protocol: TCP
      port: 8021
      targetPort: 8080
  type: ClusterIP
```

6.23. kódrészlet. Példa egy Kubernetes Service-re

Ezek után felvetődhet az a kérdés, hogy ha a ClusterIP típusú service segítségével csak a clusteren belüli forgalom tudja elérni a REST API-t, akkor külső környezetből, a kliensoldal, hogyan fogja tudni meghívni a számára szükséges endpointokat? E probléma megoldására használtam **Ingress** Kubernetes erőforrást, ami által a kliensoldal **HTTPS**, biztonságos kommunikációval tudja elérni a szerveroldali alkalmazásomat. Az ingress erőforrás yaml fájlában deklarált TLS szabályozásnak köszönhetően és a lets encrypt által kiállított illetve hitelesített certificate segítségével a külső környezettel a kommunikáció titkosított lesz. A 6.24 kódrészletben látható - a tls key alatt, annak a host címnek a definiálása, amelyre azt szeretném hogy a kommunikáció titkosított legyen. Fontos az, hogy ez az IP cím publikusan, bárki számára elérhető legyen és valid domain név legyen rá foglalva. Továbbá az ingress yaml fájlban egy útvelválasztó van deklarálva, amellyel szabályozni tudom azt, hogy a hoszt (esetemben a **skinology.k8s.edu.codespring.ro**) végpontjaira beérkező kéréseket pontosan melyik servicere irányítom át a clusteren belül. Mivel az én esetemben minden lehetséges végpont a Java Spring API-hoz kell menjen, én a path mezőnél csak a gyökér útvonalat kellett megadjam.

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: spring-backend-ingress
  namespace: skinology-ns
  annotations:
      cert-manager.io/cluster-issuer: letsencrypt-prod
      traefik.ingress.kubernetes.io/rewrite-target: /
spec:
  ingressClassName: traefik
  rules:
    - host: skinology.k8s.edu.codespring.ro
      http:
        paths:
          - path: /
            pathType: Prefix
            backend:
              service:
                name: spring-backend-service
                port:
                  number: 8021
  tls:
    - hosts:
        - "skinology.k8s.edu.codespring.ro"
      secretName: letsencrypt-prod
```

6.24. kódrészlet. Titkosított kommunikáció az Ingress segítségével

A PostgreSQL adatbázis kitelepítéséhez **StatefulSet** Kubernetes erőforrást használtam, amely lehetővé tette az adatbázis több példányban való futását. A Kubernetes által szolgáltatott **PersistentVolumeClaim** segítségével perzisztens marad az adat és ha véletlen az adatbázis egy instanciája elhal, a tárolóból könnyedén újratöltheti az elvesztett adatokat. Amint az a 6.25 kódrészletben is látható **1GB**-os tárhelyet definiáltam az

adatbázis adatainak tárolására. Az adatbázis inicializálásához és működéséhez szükséges adatokat **Secretben** és **ConfigMapben** definiáltam. A secretben az adatbázis jelszója található base64-es kódolással, a configmapben pedig az olyan információk találhatóak meg, amelyek nem muszáj titkosak maradjanak, mint például az adatbázis neve.

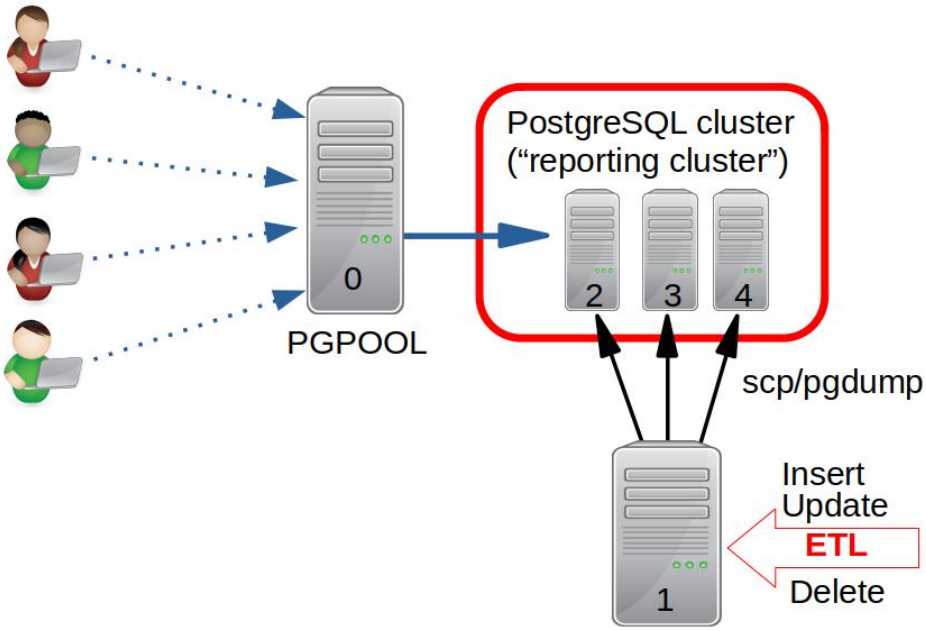
```
volumeClaimTemplates:
    - metadata:
        name: data
      spec:
        accessModes:
          - "ReadWriteOnce"
        resources:
          requests:
            storage: "1Gi"
        storageClassName: "nfs-client"
```

6.25. kódrészlet. Persistent Volume Claim Kubernetesben

Az adatbázis replikálásánál egy nehézségbe ütköztem. Az volt a probléma, hogy bármelyik adatbázisból lehetett olvasni, mert szinkronizálva voltak az adatok az instanciák között, viszont az írásnál konkurencia probléma volt. Ezt a problémát, közismertebb nevén Write-Read Skew problémának is nevezik. Ezt PostgreSQL környezetben egy PgPool-II nevű middleware réteggel lehet megoldani, amely terheléselosztást, replikációt, kapcsolat-kezelést és egyéb hasznos funkciókat biztosít. A 6.5 ábrán látható, hogy a PgPool ún. belépési kapuként szolgál az adatbázis fele, ő menedzseli a bejövő kéréseket és kimenő adatokat. A PgPool-t egy külön deploymentként kell telepíteni a clusterre, majd egy külön, a PgPool számára létrehozott servicen keresztül lehet vele kommunikálni. A harmadik komponense egy ConfigMap, amelyben többek között a PostgreSQL instanciák közötti terheléseloszlást lehet specifikálni (lásd 6.26). Írásra jogosult adatbázis instancia csak egyetlen egy van, amelyet a PgPool menedzsel, ezt nevezik **master servernek**. A többi replika csak **slave server**, amelyek csak olvasásra szolgálnak. E mellett fontos azt is megemlíteni, hogy az adatbázis esetén muszáj volt **headless servicet** használjak, ami lehetővé tette azt, hogy direkt tudjak az adott adatbázis podra hivatkozni. Míg például a Java Spring deployment esetén mindegy volt az, hogy a kérés melyik podra irányul, itt fontos az, hogy a PgPool el tudja dönteni, hogy melyik PostgreSQL podra irányítja az írás és melyikre az olvasás kéréseket. Amint az a 6.5 ábrán is látható, a Java Spring REST API elsődlegesen a PgPool II servicet hívja meg, az továbbítja a kérést a megfelelő adatbázis podra, így megoldva a terheléseloszlást és a Write-Read Skew problémát is.

```
backend_hostname1 = 'postgres-sts-replication-1.postgres-headless-svc-
replication.skinology-ns.svc.cluster.local'
backend_port1 = 5432
backend_weight1 = 1
backend_hostname2 = 'postgres-sts-replication-2.postgres-headless-svc-
replication.skinology-ns.svc.cluster.local'
backend_port2 = 5432
backend_weight2 = 2
```

[9]Forrás: Disy-Tech Blog, "PgPool-II", elérve: 2025. március 10.



6.5. ábra. PgPool II [9]

```
backend_hostname3 = 'postgres-sts-replication-3.postgres-headless-svc-
replication.skinology-ns.svc.cluster.local'
backend_port3 = 5432
backend_weight3 = 2
```

6.26. kódrészlet. Terheléselosztás PgPool segítségével

A fentebb részletezett Kubernetes yaml fájlokat Helm chartokba rendezve telepítem a clusterre (lásd 6.27) A helm használata azért előnyös számomra, mivel nem kell egyenként az erőforrásokat telepítenem a **kubectl apply -f** paranccsal, helyette elég az egész helm chartot egyszer telepítenem csak. Ez megkönnyíti a munkafolyamatot és ez mellett egy **template engine**-t is szolgáltat, amely segítségével a yaml fájlokban lévő hardcodeolt értékeket dinamikusan tudom módosítani a későbbiekben, ha szükség lesz rá.

```
helm create java-spring-chart
helm install skinology-backend ./java-spring-chart
```

6.27. kódrészlet. Helm chart létrehozása és telepítése

6.3.3. CI/CD

A fejlesztési és üzemeltetési folyamatok megkönnyítése érdekében fontosnak találtam egy **CI/CD** pipeline[10] bevezetését is a fejlesztési folyamatba. A GitLab CI/CD funkciója biztosította a pipeline megírására és futtatására szolgáló felületet. Mivel céges GitLab-

[10] A CI/CD pipeline egy automatizált folyamat, amely segít a szoftverfejlesztésben a kód integrálásában, tesztelésében és telepítésében.

ot használók, így a már előre definiált GitLab Runnerek[11] közül tudtam válogatni, hogy melyikkel szeretném végrehajtani a megírt pipelinet, ezért nem volt szükséges manuálisan csináljak egy runnert. A lentebb részletezett kódot egy **.gitlab-ci.yaml** fáljba kellett definiáljam, amely a GitLab CI/CD rendszerének egy specifikus fájlja. A Java Spring szerveroldali kódnak egy pipelinet írtam, amit két nagyobb részre osztottam fel:

- **Continuous integration**: A pipeline első része, amely a lokális kóddal kapcsolatos teendőket végzi el. Három staget tartalmaz: build, test, build-image. Az első stagen a forráskód buildelése történik (lásd 6.28), így ha valami probléma lenne buildeléskor hamar kiderül és javítani tudom. A buildelést a gradle segítségével végzem el. Mivel a kód nagy százaléka unit és integration tesztekkel van ellátva, így a pipeline második stagen, lefuttatva az előzőleg megírt teszteket hamar kiderül az, ha esetleg valami hiba lenne a biznisz logikában. Az utolsó része az integration résznek a docker image buildelése és feltöltése a privát konténer tárolóba. A docker imagek tárolására a GitLab által szolgáltatott Private Image Registryt használom, amely lehetővé teszi az ingyenes, privát tárolást ellenben a Dockerhubbal.

```
build:
    stage: build
    image: gradle:8.10.2-jdk-21-and-22
      script:
        - gradle build -x test
      artifacts:
        paths:
          - build/libs/
        expire_in: 1 week
      only:
        - merge_requests
```

6.28. kódrészlet. CI build stage

- **Continuous deployment**: Az előző részben buildelt és pusholt docker imaget a pipeline deployment részében alkalmazom a Kubernetes clusterben lévő deploymentre. Látható a 6.29 kódrészletben, hogy a mergelés után nem csinálok egy teljesen új deployment erőforrást a clusteren belül, hanem a már meglévő alól cserélem ki a docker imaget. A Kubernetes ezt a rollout folyamatot nagyon szépen és gördülékenyen tudja kezelni, így új funckionalitás hozzáadásakor semmilyen downtime nem észlelhető az alkalmazásban, hisz amíg egy pod alatt cserélődik ki az image, addig a másik pod változatlanul tudja kezelni a kéréseket. A lényeg az, hogy felváltva cserélődjenek a podokban használt imagek.

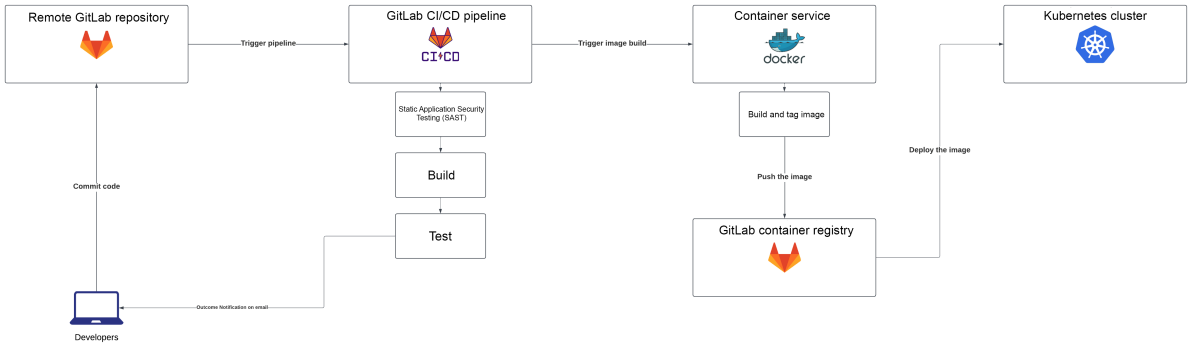
```
deploy_to_k8s:
  image:
    name: google/cloud-sdk:latest
  stage: deploy
  script:
```

[11] A GitLab Runner egy háttérben futó program, amely végrehajtja a CI/CD pipeline lépéseit a GitLab CI/CD rendszerben.

```
    - echo "$KUBECONFIG_DATA" > /tmp/kubeconfig
    - export KUBECONFIG=/tmp/kubeconfig
    - kubectl config set-context --current
        --namespace=skinology-ns
    - kubectl set image deployment/spring-backend-deployment
        spring-backend=$DOCKER_IMAGE:$DOCKER_IMAGE_TAG
```

6.29. kódrészlet. Continuous deployment

A CI/CD technológiának köszönhetően könnyedén tudok új funkcionalitásokat bevezetni a szerveroldalra, mivel gyakorlatilag downtime nélkül, automatikusan alkalmazom az új funckionalitásokat. A fentebb részletezett és bemutatott részek közül a Continuous integrationben található stagek merge request előtt futnak le (mivel addig nem enged mergeleni a GitLab, amíg minden kóddal kapcsolatos stage nem fut le helyesen), illetve a deployment stage a merge request után kerül végrehajtásra a main branchen. A 6.6 diagramon látható a teljes folyamat menete, amelyet fent részleteztem.



6.6. ábra. CI/CD flow

6.4. Backend megvalósítása

A backend rendszer Java-ban írodott Spring Boot keretrendszer segítségével, amely egy hatékony és jól skálázható megoldás a szerveroldali fejlesztésekhez. A cél egy biztonságos és megbízható RESTful API biztosítása, amely lehetővé teszi a kliensalkalmazás számára az adatok elérését és kezelését.

A rendszer főbb funkciói:

- Adatok tárolása és kezelése PostgreSQL adatbázisban.
- Object-Relational Mapping (ORM)[12] használata Hibernate-tel, amely leegyszerűsíti az adatbázis-műveleteket.

[12]Az Object-Relational Mapping (ORM) egy olyan technológia, amely lehetővé teszi az objektumorientált programozás és a relációs adatbázisok közötti automatikus átalakítást. Az ORM célja, hogy a fejlesztők ne közvetlenül SQL lekérdezésekkel manipulálják az adatokat, hanem azokat objektumokként kezeljék a programozási nyelvükben.

- Spring Data JPA és Java Database Connectivity (JDBC)[13] alkalmazása az adatelérés hatékonyságának növelésére.
- Biztonságos REST API végpontok, amelyek megfelelő jogosultságkezelést alkalmaznak.

6.4.1. Adatbázis-kezelés és ORM

A rendszer PostgreSQL adatbázist használ az adatok tárolására, amely egy megbízható és nagy teljesítményű relációs adatbázis-kezelő rendszer. Az adatbázis eléréséhez és kezeléséhez három fő technológiát alkalmaztam:

- Hibernate – Az egyik legnépszerűbb ORM keretrendszer, amely lehetővé teszi az adatok objektumorientált kezelését, miközben automatikusan kezeli az SQL lekérdezéseket és az adatbázis tranzakciókat.
- Spring Data JPA – Egy magasabb szintű absztrakció a Hibernate fölött, amely egyszerűsíti az adatbázisműveletek megvalósítását és a repository interfészek segítségével minimalizálja a boilerplate kód mennyiségét.
- JDBC – Az alacsonyabb szintű adatbáziskapcsolatok kezelésére, különösen akkor, ha nyers SQL lekérdezésekre vagy speciális teljesítmény-optimalizációkra van szükség.

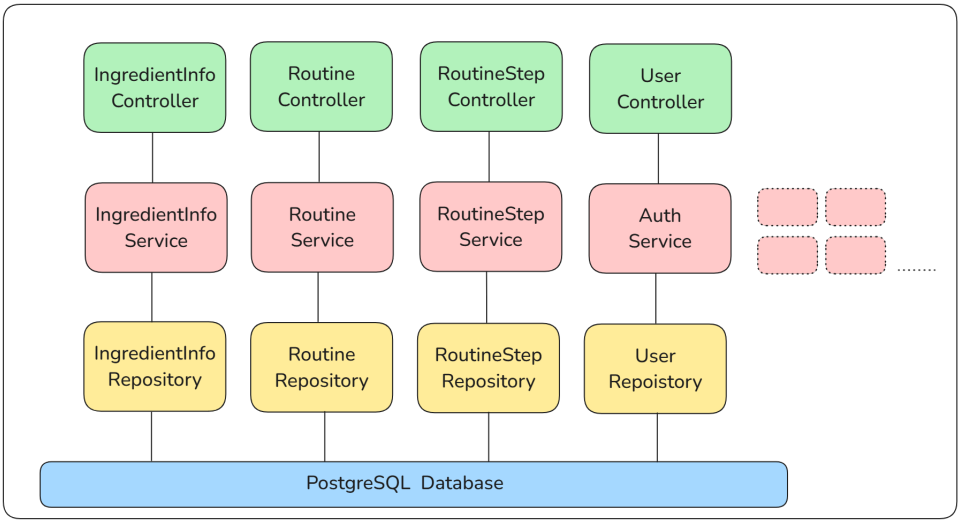
A Hibernate és a JPA kombinációja lehetővé teszi az adatbázis objektumok automatikus kezelését és az adatok gyors elérését, míg a JDBC szükség esetén alacsony szintű kontrollt biztosít az SQL műveletek felett.

6.4.2. A rendszer architektúrája

A backend alkalmazás rétegezett (Layered) architektúrát követ (6.7), amely egy jól strukturált, könnyen karbantartható és bővíthető szoftvertervezési minta. Ez a megközelítés lehetővé teszi az alkalmazás különböző felelősségi körökre való felosztását, csökkentve ezzel a kódfüggőségeket és növelve az olvashatóságot.

A rétegezett architektúra lényege, hogy az egyes komponensek egymásra épülnek, és az alkalmazás működése egyértelműen szétválasztható különböző logikai szintekre. Ez segíti a modularitást, a karbantarthatóságot és a skálázhatóságot.

[13] A Java Database Connectivity (JDBC) egy Java API, amely lehetővé teszi az alkalmazások számára, hogy kapcsolatot létesítsenek relációs adatbázisokkal. A JDBC biztosítja az adatbázisokkal való kommunikációhoz szükséges alacsony szintű interfészeket, például SQL lekérdezések futtatását, adatok beillesztését, módosítását és törlését.



6.7. ábra. Részletes backend architektúra

Controller réteg (Presentation Layer)

A Controller réteg az alkalmazás bejárati pontja, amely felelős a klienssel való kommunikációért. Ez a réteg kezeli a HTTP[14] kéréseket és válaszokat, valamint biztosítja a végpontokat, amelyeken keresztül az alkalmazás elérhető.

Feladatai:

- A kliens által küldött HTTP kérések fogadása
- A bejövő adatok validálása (pl. formátumellenőrzés, kötelező mezők megléte)
- Az üzleti logika meghívása a Service rétegen keresztül
- A válaszok megfelelő formátumban történő visszaküldése

Egy példa a Routine Controller megvalósítására 6.30 kódrészletben:

1. **Controller létrehozása**

   A RoutineController osztály kezeli a HTTP kéréseket, és különböző logikai rétegek közötti kommunikációt biztosít. A Spring **@RestController** annotációval van ellátva, így automatikusan JavaScript Object Notation (JSON)[15] formátumban válaszol a kérdésekre.

   - **@RestController**: Ez az annotáció egyesíti az @Controller és @ResponseBody annotációkat, így automatikusan JSON vagy XML formátumban válaszolunk, nem kell manuálisan megadni, hogy mi legyen a válasz formátuma.

[14] A HTTP (HyperText Transfer Protocol) egy olyan protokoll, amely lehetővé teszi a kommunikációt egy kliens (például egy webböngésző) és egy szerver között a World Wide Web-en (WWW). Az HTTP a legszélesebb körben használt protokoll a webes adatátvitelre, és alapvetően szövegalapú, ami azt jelenti, hogy a kérések és válaszok szöveges formában történnek.

[15] A JavaScript Object Notation (JSON) egy könnyen olvasható és írható adatcsere-formátum, amely szöveges formában tárolja az adatokat. Bár a JSON eredetileg a JavaScript nyelv részét képezte, mára széles körben elterjedt és számos programozási nyelv támogatja.

- **@RequestMapping**(path = "api/v1/users/userId/routines"): Ez az URL mintát jelöli ki, amelyhez a Controller hozzá van rendelve. Az userId dinamikus paraméter, amelyet a kérés során kapunk meg.
- **@Validated**: Ez biztosítja, hogy a bemeneti paraméterek validálása megtörténjen.

2. **Szolgáltatás rétegek beillesztése (Dependency Injection)**

   A Spring Framework az Inversion of Control (IoC)[16] elvet alkalmazza, amely lehetővé teszi a függőségek automatikus kezelését és injektálását. A rendszer a szolgáltatásokat a konstruktorokon vagy mezőkön keresztül biztosítja, ezzel minimalizálva a kód közvetlen példányosítását és kezelhetőségét.

   - **@Autowired**: Az annotáció jelzi a Spring konténernek, hogy automatikusan be kell injektálni a szükséges komponenseket a RoutineController osztályba. Az @Autowired lehetővé teszi, hogy a Spring kezelje a komponensek életciklusát, és megfelelő példányokat biztosítson számukra.
   - **RoutineService, UserService, AuthService, és RoutineMapper**: Ezek a szolgáltatások különböző üzleti logikai feladatokat látnak el az alkalmazásban. Míg a RoutineService és UserService a felhasználói adatokat és rutinokat kezelik, az AuthService a hitelesítési folyamatokat, míg a RoutineMapper a rutinok Data Transfer Object-be (DTO)[17] való átalakítását végzi.

3. **A GET kérés kezelése**

   - A **@GetMapping** annotáció azt jelzi, hogy egy GET kérést vár az adott utvonalon.
   - **@PathVariable** annotáció arra utal, hogy a routineId és userId paraméterek az URL-ből érkeznek.
   - **@RequestHeader(HttpHeaders.AUTHORIZATION)**: annotáció az Authorization fejlécből kivonja a token értéket, amely a felhasználó autentikációjához szükséges.

4. **Token érvényesítés**

   A token érvényesítés kritikus lépés az alkalmazás biztonságában, mivel ez biztosítja, hogy csak jogosult felhasználók férjenek hozzá az erőforrásokhoz. A kódrészletben a token ellenőrzése két lépésben történik:

[16] Az Inversion of Control (IoC) egy olyan tervezési elv, amely a program vezérlését megfordítja. Hagyományos programozásban a kód irányítja a függőségek létrehozását és kezelését, míg az IoC esetében egy külső keretrendszer (jelen esetben a Spring) veszi át ezt a felelősséget. A Spring konténer kezeli az objektumok életciklusát és függőségeit, így a fejlesztőnek nem kell közvetlenül implementálnia ezt a logikát. Ez a megközelítés növeli a kód modularitását, tesztelhetőségét és karbantarthatóságát.

[17] A Data Transfer Object (DTO) egy olyan objektum, amelyet adatátvitelre használnak különböző rétegek vagy rendszerek között. A DTO-k célja, hogy a komplex adatszerkezetek átvitelét egyszerűsítse, minimalizálja a hálózati kommunikációs költségeket, és a domain logikától elkülönítse az adatokat, amelyek szükségesek más rendszerekhez vagy rétegekhez.

- **authService.extractToken(token)**: Az AuthService osztály extractToken metódusa kivonja a JSON Web Token (JWT)[18] értékét a kérés Authorization fejlécéből.
- **authService.isTokenValid(stringToken)**: a token érvényességének ellenőrzése

Ha a token érvénytelen, a szerver **401 Unauthorized** státuszkóddal válaszol, ami azt jelenti, hogy a kérés hitelesítése nem sikerült, és a kliensnek új tokenre van szüksége. Ez a mechanizmus segít megelőzni az illetéktelen hozzáféréseket, és biztosítja, hogy az API végpontokat csak az érvényes felhasználók érhessék el.

5. **Felhasználó validálása**

   Ellenőrsére kerül, hogy a megadott felhasználói azonosító szerepel-e a nyilvántartásban. Amennyiben az azonosító érvénytelen (null értékű vagy nem megfelelő formátumú), a szerver **400 Bad Request** státuszkódú választ küld vissza. Ha az azonosító formailag helyes, de nem található meg az adatbázisban, vagyis a felhasználó nem létezik, a rendszer **404 Not Found** válasszal jelez vissza.

6. **Rutinok lekérése és keresés azonosító által**

   - **Rutinok lekérése**: Miután a felhasználó validálásra került, lekérjük az összes rutint, amely ehhez a felhasználóhoz tartozik, ha a rutinok nem találhatók, akkor **404 Not Found** választ adunk
   - Rutin keresése **routineId** alapján: A felhasználó rutinjainak lekérése után a listából az a rutin térítődik vissza amelyiknek megegyezik azonosítója a megadott azonosítóval, amennyben nincs ilyen rutin, akkor **404 Not Found** válasszal jelzi a program
   - Válasz visszaadása DTO-ként: Ha a rutin megtalálható, akkor azt egy DTO formájában adja vissza a metódus

```
@RestController
@RequestMapping(path="api/v1/users/{userId}/routines")
@Validated
public class RoutineController {
    ...
    @GetMapping("/{routineId}")
    public ResponseEntity<RoutineDTO>
        getRoutineById(@RequestHeader(HttpHeaders.AUTHORIZATION) String token,
        @PathVariable("routineId") Long routineId, @PathVariable("userId")
        Long userId) throws BadRequestException {
        String stringToken = authService.extractToken(token);
        if (!authService.isTokenValid(stringToken))
            return ResponseEntity.status(HttpStatus.UNAUTHORIZED).build();
```

[18] A JSON Web Token (JWT) egy biztonságos, tömör és önállóan hordozható formátumú token, amelyet az adatok hitelesítésére és információcserére használnak. A JWT-t leggyakrabban felhasználói azonosításra és engedélyezésre használják webes alkalmazásokban.

```
    if(userId == null || userId <= 0)
        return ResponseEntity.status(HttpStatus.BAD_REQUEST).build();

    User user = userService.getById(userId);
    if(user == null)
        return ResponseEntity.status(HttpStatus.NOT_FOUND).build();

    List<Routine> routines = routineService.getByUserId(userId);
    if(routines == null || routines.isEmpty())
        return ResponseEntity.status(HttpStatus.NOT_FOUND).build();

    Routine searchedRoutine = routines.stream().filter(routine ->
        routine.getId().equals(routineId)).findFirst().orElse(null);

    if(searchedRoutine == null)
        return ResponseEntity.status(HttpStatus.NOT_FOUND).build();

    return ResponseEntity.ok(routineMapper.toDto(searchedRoutine));
}
```

6.30. kódrészlet. Példa REST API végpont megvalósítására

Service réteg (Business Logic Layer)

A Service réteg az alkalmazás üzleti logikájának központi eleme. Ez a réteg felelős az adatok feldolgozásáért, a tranzakciók kezeléséért és az üzleti szabályok betartásáért. A **Controller** rétegtől kapott kéréseket dolgozza fel, és szükség esetén a **Repository** rétegen keresztül kommunikál az adatbázissal.

A **RoutineService** egy **@Service** annotációval ellátott osztály, ahogy a 6.31 kódrészletben is látható, amely az üzleti logika végrehajtásáért felelős. Az osztály konstruktoron keresztül kapja meg a szükséges függőségeket, amelyeket a Spring automatikusan injektál.

1. **Annotációk és függőséginjektálás**

   A Spring automatikusan kezeli az osztály példányosítását és elérhetővé teszi más komponensek számára (lásd 6.31):

   - A **routineRepository**, **quartzSchedulerService**, és **userService** privát mezők, amelyek külső szolgáltatásokra vagy adatelérési rétegre hivatkoznak.
   - A **routineRepository** az adatbázis műveletekért felelős.
   - A **quartzSchedulerService** egy értesítési ütemező szolgáltatás, amely a rutinokhoz tartozó értesítések kezeléséért felelős.
   - A **userService** a felhasználói adatok kezelésére szolgál.

```
@Service
public class RoutineService {
```

```
    private final RoutineRepository routineRepository;

    @Autowired
    private final QuartzSchedulerService quartzSchedulerService;

    @Autowired
    private final UserService userService;

    @Autowired
    public RoutineService(RoutineRepository routineRepository,
        QuartzSchedulerService quartzSchedulerService, UserService
        userService) {
        this.routineRepository = routineRepository;
        this.quartzSchedulerService = quartzSchedulerService;
        this.userService = userService;
    }
```

6.31. kódrészlet. Példa a Service réteg megvalósítására

2. **Függvény megvalósítás**

   A 6.32 kódrészlet célja, hogy egy adott felhasználóhoz tartozó rutinokat lekérje az adatbázisból, ami a következő képpen valósul meg:

   - Felhasználói azonosító ellenőrzése
   - Rutinok lekérdezése az adatbázisból a **Routine Repository** által

```
public List<Routine> getByUserId(Long userId) throws BadRequestException
    {
        if (userId <= 0) {
            throw new BadRequestException("Invalid user id");
        }

        return routineRepository.findByUserId(userId);
    }
```

6.32. kódrészlet. Példa egy függvényre a Service rétegben

Repository réteg (Data Access Layer)

A Repository réteg az alkalmazás és az adatbázis közötti absztrakciós rétegként szolgál, amely elkülöníti az üzleti logikát (Service réteg) az adattárolás részleteitől. Fő feladata az adatbázis-műveletek optimalizált kezelése ORM (pl. Hibernate, JPA) eszközökkel, miközben biztosítja az adatintegritást és teljesítményt. Feladatkörei, valamint jellemzői a következő képpen néznek ki:

1. **Adatbázis-műveletek végrehajtása (CRUD)**:

- **Create (Létrehozás)**: Entitások perzisztálása **JpaRepository**[19] segítségével
- **Read (Lekérdezés)**: Adatok lekérése egyedi azonosítóval (*findById*) vagy feltételes lekérdezésekkel (*WHERE*)
- **Update (Frissítés)**: Meglévő adatok módósítása, felülírása
- **Delete (Törlés)**: Entitások eltávolítása fizikai vagy logikai törléssel

2. **ORM-alapú lekérdezések**:
   - Natív **SQL** helyett **HQL** (*Hibernate Query Language*) vagy **JPQL** (*Java Persistence Query Language*) használata (lásd 6.33).
   - Többtáblás műveletek kezelése (**@OneToMany**, **@ManyToOne** kapcsolatok)
   - A **similarity** függvény szerepe ORM-alapú lekérdezésekben:
     A ***similarity*** függvény a PostgreSQL **pg_trgm** moduljának egy speciális kiterjesztése, amely két szöveges adat hasonlóságát méri trigramok (három egymást követő karakterből álló részletek) alapján. Az eredmény egy 0 és 1 közötti érték, ahol 1 a teljes egyezést, 0 pedig a teljes eltérést jelenti.

3. **Integráció a Service réteggel**:
   - Függőséginjektálás (**@Autowired**, **@Inject**) a Repository osztályokba
   - Üzleti logika és adatelérés szigorú elkülönítése (**SoC – Separation of Concerns**)

```
@Repository
public interface IngredientInfoRepository extends
    JpaRepository<IngredientInfo, Long> {


    @Query(
            "SELECT i FROM IngredientInfo i " +
                    "WHERE :name LIKE CONCAT('%', i.name, '%')" +
                    "ORDER BY similarity(i.name, :name) DESC " +
                    "LIMIT 1"
    )
    IngredientInfo findIngredientInfosByNameIn(@Param("name") String name);

    @Query("SELECT i FROM IngredientInfo i ORDER BY i.id ASC")
    List<IngredientInfo> findFirst5ByOrderByIdAsc(Pageable pageable);
}
```

6.33. kódrészlet. Példa Repository interfészre

[19] A JpaRepository a Spring Data JPA egyik interfésze, amely az adatbázis-kezelést egyszerűsíti. Örökli a CrudRepository és a PagingAndSortingRepository metódusait, így automatikusan biztosítja az alábbi

Adatbázis réteg (Database Layer)

Az adatbázis réteg az alkalmazás egyik legfontosabb komponense, amely biztosítja az adatok tárolását, módosítását és lekérdezését. Egy jól megtervezett adatbázis-réteg elengedhetetlen az alkalmazás megbízható működéséhez, hiszen ez felel az adatok konzisztens kezeléséért, gyors elérhetőségéért és hatékony feldolgozásáért. Az alkalmazás a PostgreSQL adatbázis-kezelő rendszert használja, amely egy nyílt forráskódú, nagy teljesítményű és skálázható relációs adatbázis. Az adatokkal való műveletek végrehajtásához a Hibernate ORM (Object-Relational Mapping) és a Java Persistence API (JPA) biztosítja az objektumorientált adatelérést, minimalizálva az SQL parancsok manuális kezelésének szükségességét.

```
@Entity
@Table
public class IngredientInfo {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    @Column(length = 4000)
    private String name;
    @Column(length = 10000)
    private String description;
    @ElementCollection
    private List<String> functionalities = new ArrayList<>();
    ...
```

6.34. kódrészlet. Példa egy adatbázis táblára

6.4.3. Értesítések

A modern mobilalkalmazások egyik alapvető funkciója az értesítések küldése, amelyek segítségével a felhasználók folyamatosan naprakészen tarthatók az alkalmazásban zajló fontos eseményekről. Az általam fejlesztett rendszerben az értesítések központi szerepet játszanak a bőrápolási rutin fenntartásában, mivel emlékeztetik a felhasználókat a napi teendőik elvégzésére. Az értesítések automatizálása segít abban, hogy a felhasználók következetesen betartsák a személyre szabott bőrápolási tervüket, és ezáltal hosszú távon javíthassák bőrük egészségét.

Az értesítési rendszer megtervezésekor kiemelt szempont volt a megbízhatóság, hiszen a késedelmes vagy elmaradó értesítések csökkenthetik az alkalmazás hatékonyságát és a felhasználói elégedettséget. Ezen túlmenően az értesítéseknek pontos időzítésűnek kell lenniük, hiszen a bőrápolási rutinok gyakran szigorúan meghatározott időpontokhoz kötöttek. Az is fontos szempont volt, hogy a rendszer skálázható legyen, így a növekvő felhasználószám mellett is megfelelően működjön.

Quartz Scheduler – Értesítések időzítése és ütemezése

A Quartz Scheduler egy megbízható és rugalmas időzítő könyvtár, amely lehetővé teszi az ütemezett feladatok végrehajtását egy Java alapú alkalmazásban. Az értesítések

esetében ezt a megoldást használtam az egyes felhasználók által beállított időpontokhoz kapcsolódó üzenetek generálására.

A rendszerben minden felhasználó beállíthatja, hogy mely időpontokban szeretne értesítéseket kapni. Az értesítési időpontokat a Quartz Scheduler egyedi ütemezési szabályok alapján tárolja és kezeli (6.35). A rendszer biztosítja, hogy minden egyes értesítés pontosan akkor kerüljön elküldésre, amikor arra szükség van.

```
// creating trigger time to send notification 5 minutes before the routine
   starts
LocalTime triggerTime = time.minusMinutes(5);
JobDetail jobDetail = JobBuilder.newJob(RoutineNotificationJob.class)
       .withIdentity(routineId + "-" + userId + "-" + token)
       .usingJobData("routineId", routineId)
       .usingJobData("routineName", routine.getName())
       .usingJobData("token", token).build();
```

6.35. kódrészlet. Példa Quartz Scheduler Job felépítése

Firebase Cloud Messaging (FCM) – Értesítések kézbesítése

Az értesítések tényleges eljuttatását a Firebase Cloud Messaging (FCM) végzi (lásd 6.36). Az FCM egy Google által biztosított szolgáltatás, amely lehetővé teszi az értesítések megbízható és hatékony kézbesítését Android és iOS eszközökre.

Az FCM főbb funkciói az értesítési rendszerben:

- Valós idejű értesítések küldése, így a felhasználók azonnal értesülhetnek a számukra fontos információkról
- Eszközspecifikus üzenetküldés – minden felhasználóhoz egyedi Device Token tartozik, így az üzenetek célzottan küldhetők ki
- Offline támogatás, vagyis az értesítések akkor is megérkeznek az eszközre, ha a felhasználó éppen nincs internetkapcsolatban – ebben az esetben az FCM automatikusan kézbesíti őket, amint az eszköz újra elérhető lesz

```
notificationService.sendNotification(
                new NotificationMessage(
                       token,
                       "\uD83C\uDF3F It's skincare time!",
                       "Hurry up! Your routine \"" + routineName + "\"
                          starts soon!",
                       "",
                       Map.of("routineName", routineName)));
```

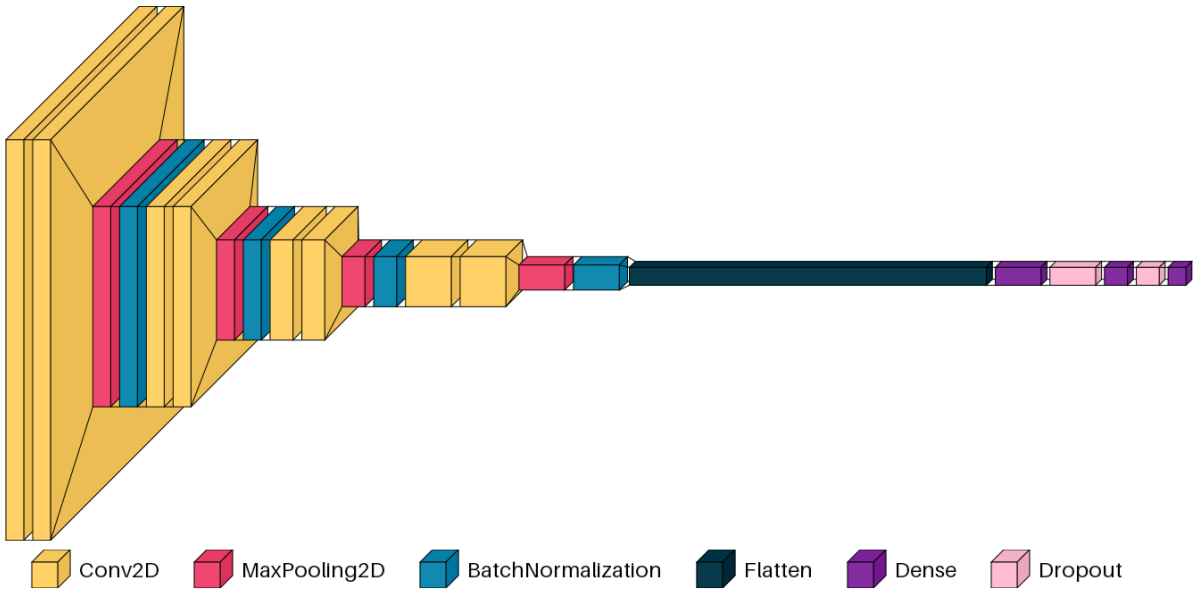
6.36. kódrészlet. Példa egy értesítés felépítésére

6.5. Mesterséges intelligencia

A bőrbetegségek automatikus osztályozására kifejlesztett neurális hálózat egy mély konvolúciós neurális hálózat (CNN), amelyet 24 különböző kategória felismerésére terveztem. A modell architektúrája többrétegű, és a modern képfeldolgozási módszerek legjobb gyakorlatait követi. A hálózatot TensorFlow keretrendszerben fejlesztettem, és körülbelül 40 000 képpel tanítottam. Az eredményeket kb. 10 000 validációs és kb. 10 000 tesztadaton értékeltem ki, hogy biztosítsam a modell általánosíthatóságát. Az alábbiakban részletesen bemutatom a modell szerkezetét, a rétegek funkcióit és a tervezési döntések mögötti indoklásokat..

6.5.1. Modellszerkezet és rétegek

Ahogy a 6.8 ábrán is látható, a modell négy, egymás után következő **konvolúciós blokkból** áll, amelyeket a klasszifikációt végző teljesen összekapcsolt rétegek követnek. Minden konvolúciós blokk két **Conv2D** réteget, egy **MaxPooling2D** réteget és egy **BatchNormalization** réteget tartalmaz. A konvolúciós fázis után a térbeli reprezentációt egy **Flatten** réteggel lineárissá alakítja, majd három **Dense** réteg segítségével végzi el a végső osztályozást, amelyek között **Dropout** rétegek helyezkednek el a túlilleszkedés csökkentése érdekében.



6.8. ábra. A modell rétegeinek vizualizációja

Konvolúciós Blokkok

1. **Conv2D rétegek**: Minden blokkban két egymás utáni konvolúciós réteg található. Ezek a rétegek 3x3-as szűrőméretet használnak, amelyek a bemeneti képek térbeli mintázatait (pl. szöveti struktúrák, színváltozások) érzékelik. A ReLU (Rectified Linear Unit) aktivációs függvényt alkalmaztam, hogy vezessek be nemlinearitást a

hálózatba, ami lehetővé teszi összetett mintázatok tanulását. A konvolúciós rétegek kimeneti csatornáinak száma fokozatosan növekszik az architektúra mélyülése során, hogy a hálózat egyre absztraktabb jellemzőket tudjon kinyerni.

2. **MaxPooling2D rétegek**: A konvolúciós rétegeket követő max pooling rétegek 2x2-es ablakmérettel csökkentik a térbeli felbontást. Ez a lépés csökkenti a számítási igényt és elősegíti a hálózat általánosítási képességét azáltal, hogy kiemeli a domináns jellemzőket és csökkenti a zajérzékenységet.

3. **BatchNormalization**: A batch normalizációs rétegek a tanítási folyamat stabilizálására szolgálnak. A rétegek normalizálják a bemeneti adatokat minden batch esetén, ami gyorsabb konvergenciát és jobb stabilitást eredményez. Emellett implicit regularizációs hatással is rendelkeznek, ami csökkenti a túlilleszkedés kockázatát.

Osztályozó Rétegek A konvolúciós blokkok után a modell egy **Flatten réteggel** átalakítja a 3D-s kimenetet (pl. [batch, magasság, szélesség, csatornák]) 1D-s vektorrá. Ez a vektor ezután egy sor **teljesen összekapcsolt** rétegbe (**Dense**) kerül, amelyek a magas szintű jellemzők kombinálásával végzik el a klasszifikációt.

- **Első Dense réteg**: Magas neuron-számú réteg (pl. 512 vagy 1024 neuron), amely a komplex jellemzőkapcsolatokat modellezi.

- **Dropout**: A Dense rétegek között elhelyezett Dropout rétegek véletlenszerűen kikapcsolnak neuronokat a tanítás során (50% eséllyel), ami radikálisan csökkenti a túlilleszkedés kockázatát.

- **Kimeneti Dense réteg**: 24 neuronnal rendelkezik, amelyek mindegyike egy-egy osztályhoz tartozó valószínűségi értéket reprezentál. A **Softmax aktivációs függvény** biztosítja, hogy ezek az értékek 0 és 1 közötti valószínűségekké alakuljanak, és összegük 1 legyen.

6.5.2. Tanítási beállítások és hiperparaméterek

A modell tanítását a **Adam** optimalizáló algoritmussal végeztem, amely jól alkalmazható képfeldolgozási feladatokban, mivel adaptív tanulási rátát használ a különböző paraméterekhez. A kezdeti tanulási ráta (**learning rate**) értékét 0,001-re állítottam be, amely megfelelő egyensúlyt biztosít a gyors konvergencia és a stabilitás között.

A kategóriák közötti többosztályos klasszifikáció miatt a veszteségfüggvényként a **categorical_crossentropy**-t használtam. Az értékelési metrikák között a **accuracy**, **precision** és **recall** mutatókat is figyeltem, mivel ezek különösen fontosak lehetnek egy-egy ritkábban előforduló bőrbetegség esetében.

A modell **fit** függvény segítségével került betanításra, ahol a tanítóadatokat tartalmazó **train generatoron** végeztem a tanítást. A tanítás során összesen **100 epoch**-ot futtattam (amennyiben kevesebb epoch-ra volna szükség, a későbbiekben tárgyalt EarlyStopping leállítja), **shuffle=False** beállítással, hogy a képek sorrendje az egyes epoch-ok során változatlan maradjon.

A tanítási folyamat során több **callback** funkciót is használtam a hatékonyság és a megbízhatóság növelése érdekében:

- **EarlyStopping**: a tanítás megszakítása, ha a validációs veszteség több egymást követő epoch során nem javul.
- **ModelCheckpoint**: a legjobb teljesítményű modell mentése a validációs pontosság alapján.
- **ReduceLROnPlateau**: a tanulási ráta automatikus csökkentése, ha a validációs veszteség egy adott számú epoch után nem javul.

6.5.3. Adatelőkészítés és augmentáció

A képadatokat egységesen **224×224 pixeles** méretre skáláztam, amely szabványos képméret a modern konvolúciós neurális hálózatok esetében, így biztosítva a kompatibilitást a modell bemeneti rétegeivel. A képek színcsatornáit **RGB** formátumban használtam.

A képeket **ImageDataGenerator** osztály segítségével olvastam be és dolgoztam fel, amely lehetővé tette a képek **0 és 1 közötti normalizálását** a **1/255** szorzót alkalmazva. Ezzel biztosítható, hogy a bemeneti értékek az aktivációs függvények optimális működési tartományába essenek, ami gyorsítja a tanítást és stabilabb konvergenciát eredményez.

A tanító-, validációs- és teszthalmazokat **flow_from_dataframe** függvénnyel töltöttem be, ahol a képek elérési útját és a hozzájuk tartozó címkéket adatkeretekben (**DataFrame**) tároltam. A betöltés során a következő beállításokat alkalmaztam:

- **batch size**: 64
- **target size**: 224×224 pixel
- **class mode**: **categorical**, mivel többosztályos klasszifikációt végeztem
- **color mode**: **rgb**
- **shuffle**: a tanító és validációs halmazok esetében **True**, a teszthalmaznál **False**, hogy az eredmények összevethetők legyenek a címkékkel

A jelenlegi megvalósításban **adattágító (augmentációs)** technikákat nem alkalmaztam, mivel az ImageDataGenerator **CPU** alapon, működik ezáltal nagyon lelassította a tanulási folyamatot. A jövőben viszont célom, hogy egy **GPU** alapú módszerrel próbáljak meg augmentációt végezni az adatokon.

6.5.4. Fejlesztési környezet

A modell fejlesztését és tanítását a **Kaggle** felhőalapú gépi tanulási platformon végeztem, amely előre telepített könyvtárakkal és GPU-támogatással biztosít kényelmes környezetet a mélytanulási feladatokhoz. A platform választását az is indokolta, hogy a Kaggle környezetben elérhető **GPU-k egyenként 16 GB memóriával rendelkeznek**, valamint **29 GB rendszermemória (RAM)** áll rendelkezésre, ami elegendő erőforrást biztosít a nagyobb méretű képadatok feldolgozásához és a komplex modellek betanításához.

A tanítás során **GPU-gyorsítást** alkalmaztam a számítási idő csökkentése érdekében. A kísérletek alatt kipróbáltam **egy GPU-s** és **két GPU-s** konfigurációkat is. Azonban mivel a betöltéshez és előfeldolgozáshoz **ImageDataGenerator** osztályt használtam, amely szekvenciálisan olvassa be és dolgozza fel a képadatokat, a feldolgozás sebessége korlátozott maradt. Ennek következtében a két GPU-s konfiguráció érdemi gyorsulást nem eredményezett a tanítási idő tekintetében.

A jövőbeni fejlesztési irányok között szerepel hatékonyabb adatbetöltési és augmentációs módszerek, például **TensorFlow Dataset API** vagy **tf.data pipeline** használata, amelyek párhuzamos adatfeldolgozást és GPU-barát adatátvitelt biztosítanak, így jobban kihasználhatóvá téve a több GPU-s környezet előnyeit.

6.5.5. Összegzés

A jelenlegi modell körülbelül **70%-os pontossággal** képes bőrbetegségeket osztályozni képfelvételek alapján. Célom, hogy ezt az arányt **90% fölé** emeljem, növelve ezzel a rendszer megbízhatóságát. A további javítás érdekében tervben van **adattágítás**, a **tanítási pipeline optimalizálása**, valamint **transfer learning** módszerek alkalmazása is, amelyek várhatóan jelentős teljesítménynövekedést eredményezhetnek.

6.6. Tesztelés

A tesztelési stratégiám célja az alkalmazás megbízhatóságának és stabilitásának biztosítása, különös tekintettel a Spring keretrendszerben működő komponensekre. A tesztelési folyamat során kiemelt figyelmet fordítottam a szolgáltatási réteg (**Service Layer**) és az adatbáziskommunikáció ellenőrzésére, mivel ezek alapvető szerepet játszanak az üzleti logika és az adatkezelés szempontjából.

A megbízható tesztelés érdekében **unit** teszteket alkalmaztam, amelyek lehetővé tették az egyes osztályok és metódusok izolált vizsgálatát, függetlenül a külső tényezőktől, például az adatbázistól vagy más szolgáltatásoktól. Ennek eredményeként az egyes komponensek működését külön-külön ellenőriztem, így a tesztelési folyamat gyorsabbá és hatékonyabbá vált.

A tesztelési stratégiám magában foglalta a különböző mockolási technikák alkalmazását, hogy a tesztek során a külső függőségeket virtuális helyettesítőkkel váltsam ki. Ezenkívül integrációs teszteket is terveztem, hogy a különböző komponensek együttműködését ellenőrizzem valós környezetben.

6.6.1. Tesztelési módszerek és eszközök

A fejlesztési folyamat során kiemelt figyelmet fordítottam az alkalmazás különböző komponenseinek automatikus tesztelésére, hogy biztosítsam a megfelelő működést és minimalizáljam a hibalehetőségeket (lásd 6.9). Ehhez megbízható tesztelési módszereket és eszközöket alkalmaztam, amelyek lehetővé tették az egyes modulok és metódusok izolált vizsgálatát, valamint a rendszer egészének integrált ellenőrzését.

A tesztelés során a következő eszközöket, valamit technikákat használtam, hogy gördülékenyenség és hitelesség érdekében:

1. **JUnit 5**

   A JUnit 5 az egyik legelterjedtebb Java tesztelési keretrendszer, amelyet az egységtesztek (**unit** tesztek) és integrációs tesztek írására és futtatására használtam. A keretrendszer lehetőséget biztosított:

   - Metódusok és osztályok izolált tesztelésére, hogy az egyes komponensek helyes működését önállóan ellenőrizzem
   - Különböző tesztesetek definiálására, beleértve a normál működést és a hibás bemenetek kezelését
   - Annotációk használatára (pl. **@Test**, **@BeforeEach**, **@AfterEach**), amelyekkel a tesztelési folyamat strukturáltabbá és átláthatóbbá vált
   - Paraméterezett tesztek írására, amelyek segítségével különböző bemenetekkel vizsgálhattam az egyes metódusok működését

2. **Mockito**

   A Mockito egy népszerű tesztelési könyvtár, amely lehetővé teszi mock objektumok létrehozását. A tesztelési folyamat során a repository-k és külső szolgáltatások tesztelésére használtam a Mockito-t, így a valódi adatbázis vagy külső API-k elérése nélkül is tudtam ellenőrizni a szolgáltatási réteg működését.

3. **Mockito verify** – Metódushívások ellenőrzése

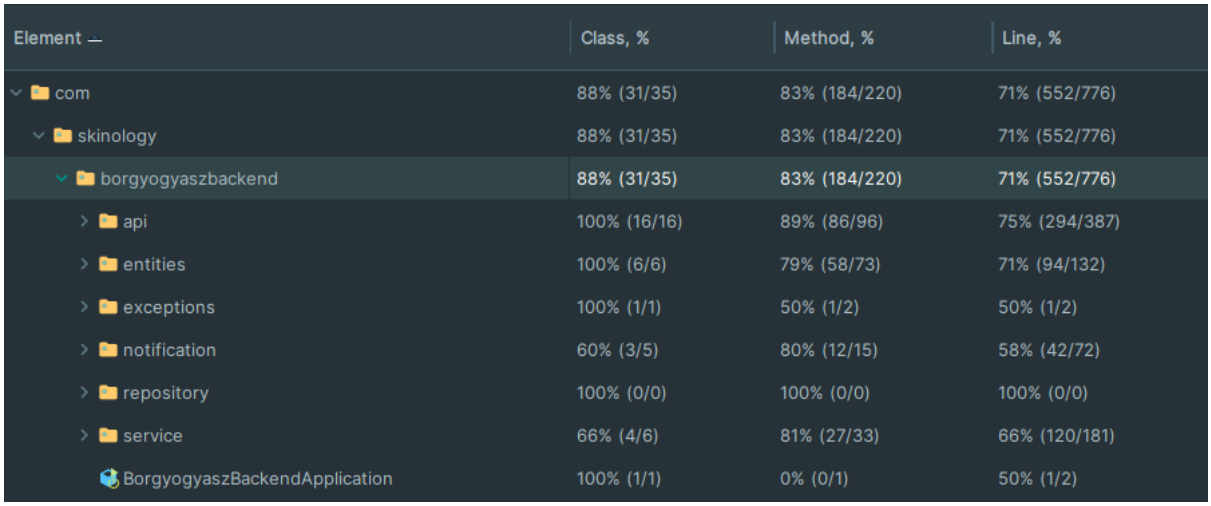
   A Mockito verify funkcióját arra használtam, hogy biztosítsam a megfelelő metódusok végrehajtását és azok helyes hívását a tesztelési folyamat során. A **verify** függvény segítségével a következőket tudtam megvalósítani:

   - Ellenőriztem, hogy egy adott metódus pontosan hányszor futott le, például a repository megfelelő hívását egy adott művelet végrehajtása során
   - Biztosítottam, hogy az alkalmazás nem hajt végre felesleges vagy nem kívánt műveleteket, például hogy egy adott metódus csak egyszer vagy a megfelelő paraméterekkel lett meghívva

4. **A tesztelés célja és megvalósítása**

   A tesztelés legfőbb célja az volt, hogy biztosítsam a szolgáltatási réteg metódusainak helyes működését, különböző bemeneti adatok és adatbázis-műveletek kezelése során. Ennek érdekében:

   - **Pozitív és negatív** teszteseteket is készítettem, hogy az alkalmazás megfelelően kezelje a helyes és helytelen bemeneteket
   - **Valós szituációkat** modelleztem, például azt, hogy mi történik, ha egy adott rekord nem található meg az adatbázisban, vagy ha egy lekérdezés hibás bemeneti paramétereket kap
   - Teszteltem az adatbázis-műveletek helyes végrehajtását, ellenőrizve, hogy az adatok megfelelően mentésre kerülnek, frissülnek vagy törlődnek az adatbázisból



Element	Class, %	Method, %	Line, %
com	88% (31/35)	83% (184/220)	71% (552/776)
skinology	88% (31/35)	83% (184/220)	71% (552/776)
borgyogyaszbackend	88% (31/35)	83% (184/220)	71% (552/776)
api	100% (16/16)	89% (86/96)	75% (294/387)
entities	100% (6/6)	79% (58/73)	71% (94/132)
exceptions	100% (1/1)	50% (1/2)	50% (1/2)
notification	60% (3/5)	80% (12/15)	58% (42/72)
repository	100% (0/0)	100% (0/0)	100% (0/0)
service	66% (4/6)	81% (27/33)	66% (120/181)
BorgyogyaszBackendApplication	100% (1/1)	0% (0/1)	50% (1/2)

6.9. ábra. Tesztek lefedettsége

6.6.2. Tesztelési eredmények

Controller tesztelés

A controller réteg tesztelése során arra törekedtem, hogy biztosítsam az API végpontok helyes működését. Ennek érdekében ellenőriztem, hogy a beérkező **HTTP** kérések megfelelően feldolgozásra kerülnek, és a válaszok a várt eredményeket adják vissza. A célom az volt, hogy garantáljam a források és a szolgáltatások megfelelő irányítását.

Tesztelt funkciók:

- **HTTP GET kérések:** Megvizsgáltam, hogy a controller megfelelő válaszokat ad-e a REST API végpontokra, különösen az összetevőinformációk lekérdezésekor
- **HTTP POST/PUT kérések:** Teszteltem, hogy az új összetevők hozzáadása, frissítése és törlése megfelelően működik-e a controller szintjén
- **Válaszok ellenőrzése:** Minden válaszkódot és adatot validáltam, hogy meggyőződjek arról, hogy a megfelelő státuszkódok és adatok kerülnek visszaküldésre
- **Mock szolgáltatások:** A szolgáltatásokat mock-oltam, hogy a controller helyes működését teszteljem anélkül, hogy a teljes szolgáltatási réteget újratesztelném

Service tesztelés

A service réteg tesztelése során arra összpontosítottam, hogy az üzleti logika és a különböző folyamatok megfelelően működjenek. Ehhez mock objektumokat használtam, hogy a teszteket izoláltan tudjam végrehajtani, kizárva a külső rétegek hatásait.

Tesztelt funkciók:

- **Funkcionális tesztek:** Az alkalmazás üzleti logikáját megvalósító metódusokat egyenként teszteltem, hogy megbizonyosodjak arról, hogy az elvárt eredményeket adják vissza

- **Mock repository és dependency injection:** A repository-k és más külső függőségek helyettesítésére mock objektumokat használtam, így biztosítva, hogy kizárólag a service réteg logikáját vizsgáljam, és ne történjenek valódi adatbázis-műveletek

A service rétegben több olyan tranzakció is található, amelyek módosítják az adatállapotokat. Ezért külön figyelmet fordítottam arra, hogy ellenőrizzem, hogy a megfelelő tranzakciók végrehajtódnak-e, és hogy hiba esetén a rendszer képes-e visszagörgetni a változtatásokat.

6.7. Scraping

Fontos volt számomra az, hogy az alkalmazás fejlesztése folyamán saját adatokkal tudjak dolgozni és ne függjek thrid-party API-októl, ezért is kapott kulcsfontosságú szerepet az adat scraping a tervezési fázis folyamán. Elsősorban a bőrápolási termékek összetevőiről volt szükség adatra, mint például az összetevők neve, leírása és funkcionalitása. E mellett szükséges volt azt is tudni, hogy az összetevők káros, semleges vagy pozitív hatással vannak a felhasználó bőrére. Mivel nem kaptam egy egységes és átfogó oldalt, amin minden információ megtalálható lett volna két fázisra osztottam le az adatgyűjtés folyamatát:

6.7.1. Az összetevőkről gyűjtött általános információk

Az összetevők az alkalmazás egyik legfontosabb részét képezik, ezért elengedhetetlen volt, hogy a lehető legtöbb adatot összegyűjtsem. Az adatgyűjtés során a következő fontos lépéseket és technológiákat használtam:

Funkcionalitások listájának összegyűjtése

- **Böngésző indítása:** A program a Puppeteer segítségével egy Chrome/Chromium böngészőt indít el automatizált módon. Ez lehetővé teszi, hogy a kód betöltse a kívánt weboldalakat, azokon különböző műveleteket hajtson végre, valamint az oldalon található adatokhoz hozzáférjen és azokat feldolgozza

- **Weboldal betöltése:** Miután a böngésző elindult, a kód megnyitja az EU COSING[20] adatbázisának funkcionalitásokat listázó oldalát, hogy adatokat nyerjen ki. adatbázis adott oldalát, amely a kozmetikai összetevők különböző funkcionalitásait tartalmazza. Ez az oldal táblázatos formában listázza az egyes összetevők funkcióit, például hidratáló, tartósítószer vagy antioxidáns szerepüket.

- **Funkcionalitások kivonása:** Az oldal betöltése után a Puppeteer segítségével a kód megkeresi a funkcionalitásokat tartalmazó táblázatot, majd kinyeri a táblázatban szereplő funkcionalitások neveit és eltárolja őket egy listában (lásd 6.37).

- **Adatok mentése:** Miután az összes funkcionalitást sikeresen kinyerte a rendszer, azokat JSON formátumban menti el egy fájlba. Ehhez a Node.js beépített fs (file system) modulját használja, amely lehetővé teszi fájlok létrehozását és módosítását.

[20] Cosmetic Ingredients Notification

Az adatok elmentése biztosítja, hogy a későbbiekben más alkalmazások vagy elemző eszközök is könnyedén feldolgozhassák azokat.

```
async function getFunctionsList(const URL) {
let browser;
try {
  browser = await puppeteer.launch(); // open the browser
  const page = await browser.newPage();
  await page.setViewport({ width: 1366, height: 768 }); // set the viewport
     (default)
  // set timeout to 2 minutes
  page.setDefaultNavigationTimeout(2 * 60 * 1000);
  // scrape data (functionalities) from a page
  async function scrapePage() {
    await page.waitForSelector("table a");
    const functions = await page.$$("table a");
    for (const element of functions) {
      const functionName = await element.evaluate((el) => el.innerText);
      functionsList.push(functionName); // store the functionality in the
         list
    }
  }
  // opening the page where the functionalities are
  await page.goto(URL);
  await scrapePage();
  // store the list in local storage
  fs.writeFile("functionsList.json", JSON.stringify(functionsList));
  await browser.close();
} catch (err) {
  console.log(err);
  if (browser) {
    await browser.close();
  }
}
}
```

6.37. kódrészlet. Funkcionalitások listájának összegyűjtése

Összetevők (ingredients) összegyűjtése

- **Böngésző indítása és oldal betöltése:** A kód a Puppeteer segítségével elindít egy Chrome/Chromium böngészőt, majd betölti az adott weboldalakat, amelyek az egyes funkcionalitásokhoz tartozó összetevőket tartalmazzák. Ez a lépés hasonló a korábban ismertetett funkcionalitásgyűjtési folyamathoz, azonban itt az összetevőkre fókuszál (lásd 6.38).
- **Összetevők kivonása:** Az oldal betöltése után a kód kinyeri az összetevők neveit és azokat egy Map struktúrában tárolja, ahol a funkcionalitások neveihez a hozzátartozó összetevők listája kapcsolódik.

- **Többoldalas tartalom kezelése:** Mivel egy-egy funkcionalitás több összetevőt is tartalmazhat, az információ gyakran több oldalon keresztül érhető el. A kód figyeli, hogy elérhető-e további oldal, és ha igen, akkor a Puppeteer segítségével automatikusan a következő oldalra lép. Ezáltal a program minden releváns adatot kinyer anélkül, hogy manuálisan kellene végiglapozni az oldalt.
- **Adatok mentése:** A kinyert adatokat a Node.js beépített fs (file system) modulja segítségével menti el. Az összetevőket tartalmazó Map struktúrát JSON formátumban rögzíti, amely könnyen feldolgozható más rendszerek vagy elemző eszközök számára.

```
  for (const functionName of [...functionsList]) {
    await page.goto(`${URL}/${functionName}`);
    await scrapePage(functionName);
  }
```

6.38. kódrészlet. Összetevők begyűjtése funkcionalítások alapján

Összetevők leírásainak összegyűjtése

- **Böngésző indítása és oldal betöltése:** A kód ismét a Puppeteer használatával navigál az INCIDecoder weboldalra, ahol az egyes összetevők részletes leírását próbálja megszerezni (lásd 6.39).
- **Leírások kivonása:** A Puppeteer segítségével a kód megpróbálja kinyerni a leírásokat különböző HTML elemekből. Ha talál leírást, azt eltárolja, ha nem, akkor hibakezelést végez.
- **Adatok mentése:** A kinyert leírásokat a PostgreSQL adatbázisba tölti fel a ***pg*** Node.js modul használatával, valamint a helyi fájlrendszerbe is menti őket JSON formátumban.
- **Hibakezelés:** A kód gondoskodik a hibák kezeléséről is, például, ha egy oldal nem tölt be vagy ha nincs elérhető leírás, a folyamat folytatódik a következő összetevővel.

```
async function getDescriptions() {
  try {
    // read data from local storage
    const data = await fs.readFile("ingredientsList.json", "utf8");
    const ingredients = JSON.parse(data);
    // call the scrapeDescription() function with the ingredients list
    await scrapeDescription(ingredients);
  } catch (err) {
    console.error("Error reading file or processing data:", err);
  }
}
```

6.39. kódrészlet. Összetevőkhöz tartozó leírások begyűjtése

Adatok felhasználása és további feldolgozása

- **Adatok adatbázisba töltése:** A kinyert funkcionalitásokat és hozzávalókat a PostgreSQL adatbázisba tölti fel (lásd 6.40), ahol a továbbiakban felhasználhatók lesznek. Ehhez a kód ismét a ***pg***, Node.js modult használja.
- **Kód optimalizálás:** A weboldalakról történő adatkinyerés (web scraping) esetén figyelembe kell venni a céloldalak teljesítményét és az esetleges szerverkorlátozásokat. Ha a kód túl gyorsan és túl sok kérést küld, az IP-cím tiltásához vagy a hozzáférés korlátozásához vezethet. A kód egy időzítési mechanizmust alkalmaz (**sleep** függvény), amely minden adatlekérés után egy kis várakozási időt iktat be. Ez csökkenti a szerver terhelését és segít elkerülni az esetleges kitiltást vagy blokkolást.

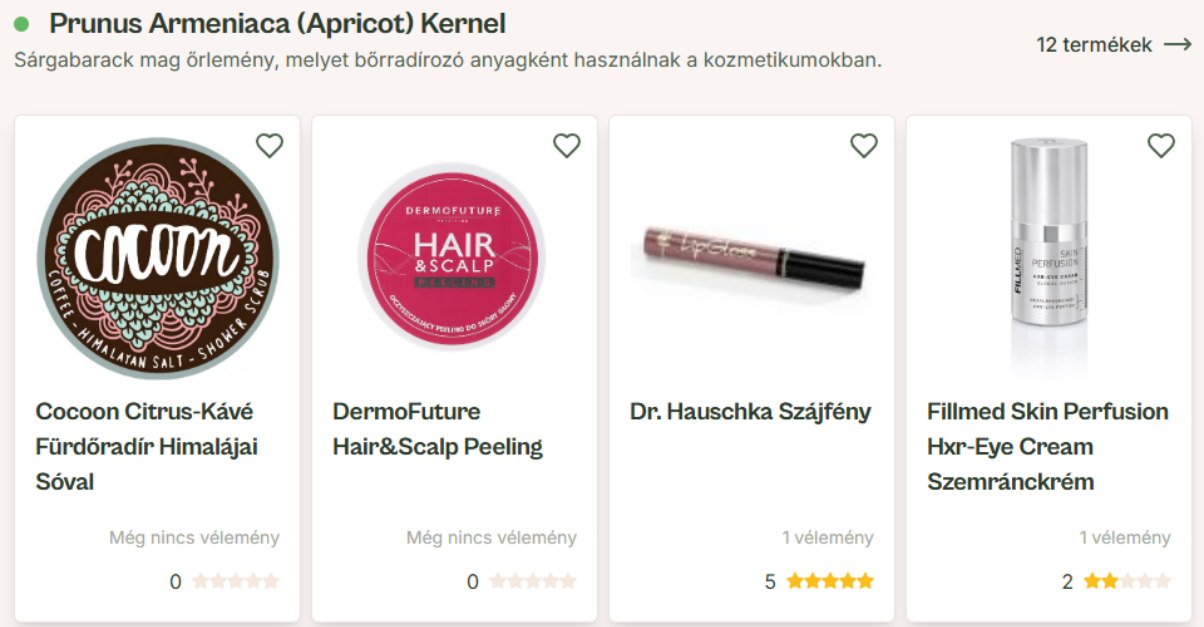
```
try {
   // connect to the database
   await client.connect();
   const functionsList = await readFunctionsList();
   // iterate through the list of functionalities and inserting them into the
      database
   for (const functionName of functionsList) {
     const query = `INSERT INTO functionality (name) VALUES
        ('${functionName}');`;
     await client.query(query);
   }
 } catch (err) {
   console.log(err);
 } finally {
   // close the connection
   await client.end();
 }
```

6.40. kódrészlet. Adatok beszúrása

6.7.2. Az összetevők hatása a bőrre nézve

Mivel az alapötlet az volt, hogy az alkalmazás tudja megítélni azt, hogy az adott bőrápolási termék összetevői közül melyik káros, semleges vagy akár pozitív a felhasználó bőrére, ezért szükségem volt minden egyes összetevőről, hogy milyen hatással van a bőrfelületre. Sajnos a fejlesztés kezdeti fázisában egy olyan oldal sem állt rendelkezésemre, ahol minden összetevőre fel lett volna tüntetve ez az információ, viszont egy kis idő elteltével - a Krémmánia oldal karbantartását és felújítását követően, megjelentek ezek az információk is az általános tudnivalók mellett. Örültem, hogy egy megbízható és magyar oldalról tudom az információkat kinyerni és nem kell ismeretlen adatforrásokban megbízzak, ahol az is lehet hogy nem hiteles és valid adatok vannak megjelenítve.

Amint az a 6.10 képen is látható az oldalon egyértelműen és intuitív módon van szemléltetve az, hogy az adott összetevő káros-e vagy sem a bőrre. A szürke pont a semelegességet, a zöld pont a pozitív, a piros a negatív, a sárga pedig a figyelmeztetett hatást szimbolizálja.



6.10. ábra. Példa egy összetevőre a Krémmánia oldalon

Mivel ebben az esetben az adatgyűjtés során nem volt muszáj semmilyen interakció az oldallal (például gombnyomás), így a BeautifulSoup Python könyvtárcsomagot használtam, mivel könyebbnek és gyorsabbnak tűnt, mint a Javascriptes megoldás.

Elsőként az oldalakon való navigálás tűnt a legkönnyebb megoldásnak, mivel a Krémmánia URI-jában tökéletesen lehet helyettesíteni azt az oldalszámot, amelyiket éppen szeretném. Például ha a harmadik oldalra szeretnék navigálni a következő URI-t kell használjam: **https://kremmania.hu/osszetevok?page=3**. Ezzel a megoldással az volt a probléma, hogy az oldalon csak 999 oldal tekinthető meg ilyen formában és így nem tudtam hozzáférni minden egyes összetevőhöz, amely az oldal adatbázisában szerepel. Alternatív megoldásként az oldal szűrőjét használtam ahhoz, hogy minden adatot elérjek. Időközben rájöttem, hogy nevenkénti szűrés esetén - az ABC betűin iterálva, minden összetevőt le tudok tölteni, így például a **https://kremmania.hu/osszetevok?alphabet=A** URI segítségével elérek minden A betűvel kezdődő összetevőt.

Pythonban elsősorban a Selenium csomag segítségével megvártam, hogy betöltődjön teljesen az oldal, majd azután kezdtem el letölteni az adatokat, ahogy az a 6.41 programrészben látható.

```
options = Options()
options.headless = True
driver = webdriver.Chrome(options=options)

for letter in alphabet:
    page = 1
    while True:
        print(f"Scraping page {page} for letter '{letter}'...")
        url = f"{base_url}?name=&functions=&alphabet={letter}
        &ewgScore=&page={page}"
        driver.get(url)
```

```
        driver.implicitly_wait(10)
```

6.41. kódrészlet. Selenium használata

Ezek után egy hosszú CSS szelektor (lásd 6.42) segítségével kihámoztam az aktuális oldalról a számomra releváns információkat és adattisztítást illetve előkészítést végeztem. Az adatbázisban már előzetesen felvezetett összetevők mellé kellett beszúrnom a státuszukat is és ez gondot okozott, mivel a Krémmánia és az EUCosIng nem mindig egyformán tartotta számon ugyanannak az összetevőnek a nevét és formázását. Az egyeztetést a RapidFuzz nevű csomaggal oldottam meg, amely gyors és hatékony **fuzzy string matching**[21] algoritmusokat biztosít. Főként akkor szokták használni, ha két szöveget kell összehasonlítani, amelyek nem teljesen egyeznek meg.

```
items = soup.select('#__next > div > main >
div.section-content.grid.grid-cols-1.pt-6.lg\\:grid-cols-ingredients-
page.lg\\:gap-10 > div.col-start-1 > section:nth-child(2) > div > div
> div > div > a > div > h3')
items_color = soup.select('#__next > div > main > div.section-content.
grid.grid-cols-1.pt-6.lg\\:grid-cols-ingredients-page.lg\\:gap-10 >
div.col-start-1 > section:nth-child(2) > div > div > div > a > div >
    div')
```

6.42. kódrészlet. CSS szelektor az adatgyűjtéshez

6.8. Work management

A fejlesztés során kiemelt figyelmet fordítottunk a csapatmunkára is, éppen ezért agilis módszerek segítségével menedzseltük a munka menetét. A SCRUM metodológiát követve a fejlesztési folyamatot - egy nagy, monolit szintű fejlesztés helyett több, kisebb munkakörre bontottuk le. Elsősorban - amikor elkezdtük az alkalmazás fejlesztését egy **kick-off**[22] meeting keretén belül megismertük egymást a csapaton belül.

Ahogy az a 6.11 ábrán is látható a termékfejlesztés folyamatát több **sprintre** bontottuk le. A sprintjeink a fejlesztés elején egy hetesek voltak, majd átálltunk két hetes intervallumokra, mivel az egyetem mellett sokkal kézenfekvőbbek voltak a hosszabb sprintek. A sprinteket tervezési megbeszélések előzték meg, ahol a csapat megvitatta azt, hogy a következő sprintben mit szeretne megvalósítani, milyen új funckionalitásokat szeretne lefejleszteni. Ilyenkor a **product backlogból**[23] válogattunk, prioritási sorrendnek megfelelően ötleteket és továbbfejlesztési lehetőségeket.

A sprint folyamán minden nap a csapattal megbeszéléseket tartottunk - ún. **daily scrum** meetingeket, amelyek keretén belül mindenki elmondta a csapatból, hogy éppen

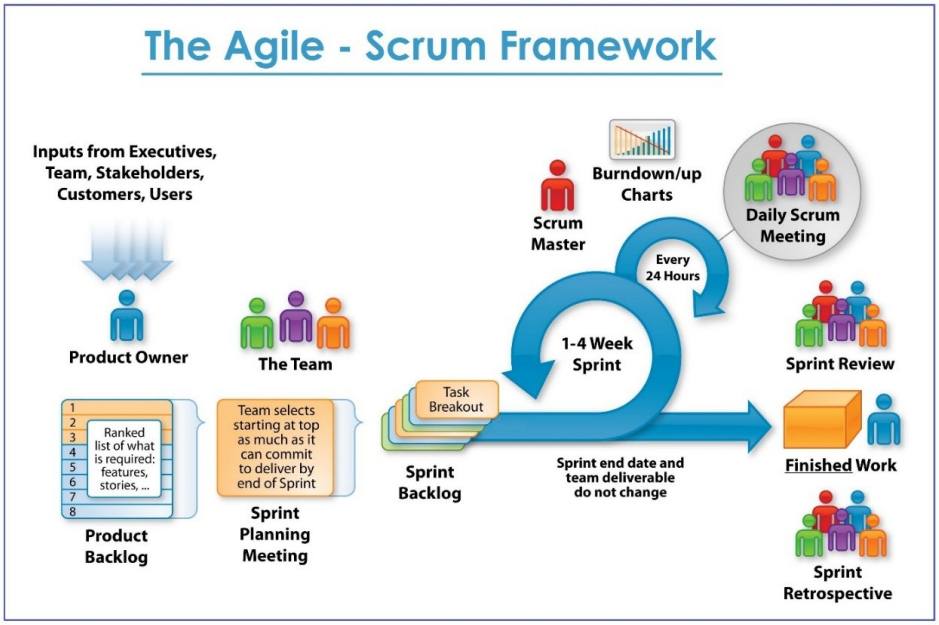
[21] A fuzzy string matching (homályos szövegillesztés) egy olyan módszer, amely két szöveg közötti hasonlóságot méri, még akkor is, ha azok nem teljesen egyeznek. Ez hasznos például elírások, rövidítések vagy különböző írásmódok esetén.

[22] A kick-off meeting egy projekt vagy együttműködés kezdeti megbeszélése, amelynek célja, hogy minden érintett fél egy közös alapra kerüljön a projekt céljaival, elvárásaival és következő lépéseivel kapcsolatban.

[23] A product backlog egy prioritizált feladatlista, amely a fejlesztendő termékhez kapcsolódó összes szükséges funkciót, fejlesztést és javítást tartalmazza.

min dolgozott az előző nap és mivel fog foglalkozni aznap. Ha valami elakadása vagy kérdése volt valakinek azt megpróbálta a csapat együtt megoldani, hogy gördülékenyen tudjon tovább haladni az alkalmazás fejlesztése. Próbáltuk a napi megbeszéléseket minél hatékonyabban lebonyolítani és minél rövidebb időn belül végezni velük, hogy felesleges megbeszélésekkel ne teljen a fejlesztésre szánt idő.

A sprintek végén **demó megbeszéléseket** tartottunk, ahol a csapat bemutatta azt, hogy min dolgozott az elmúlt időszakban, így pár hetente tudtuk aktualizálni az alkalmazás funckionalitás listáját. Mivel a mi esetünkben nem volt sem kliens, sem product owner a projektben, ezért a vezetőtanárunknak, illetve a mentorainknak demóztuk az előrehaladást.



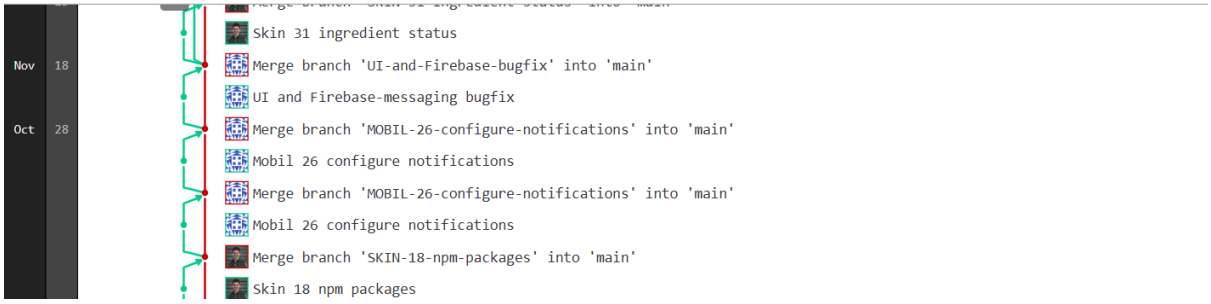
6.11. ábra. SCRUM keretrendszer [24]

6.8.1. Verziókövetés

Verziókövetőnek a Git technológiát használtuk és a GitLab platformon tároluk az alkalmazás forráskódját. A backend, frontend, illetve AI résznek három különböző repository van létrehozva annak érdekében, hogy átláthatóak és jól elkülöníthetőek legyenek az alkalmazás egyes részei. Amint az a 6.12 git fán is látható minden egyes funkcionalitásra külön branchet húztunk ki és azon fejlesztettünk. Miután az adott funkció fejlesztése megtörtént **merge requesteket** hoztunk létre, amelyeket egymásnak reviewztunk ki. A review alatt próbáltunk minden egyes kis részletre odafigyelni, hogy a lehető legrobosztusabb és leghatékonyabb legyen a kódunk. Ha úgy gondoltuk, hogy a csapattársunk kódja megfelel az elvárásainknak az approve segítségével jeleztük, hogy felőlünk bemehet a main branchre a módosítás. Amennyiben a CI/CD pipeline is sikeresen lefutott, a main ág kibővült az új kódrészlettel. Fontos megjegyezni azt, hogy a mellékelt ábrán úgy

[24]Forrás: SystemPlus, "SCRUM", elérve: 2025. március 12.

tűnik, mintha csak bugfix branchek lettek volna az októberi és novemberi hónapokban - mivel egyetlen commit van minden feature branchen, viszont ez nem így van. Mergelésnél mindig **squasholtuk**[25] a feature branch commitjait éppen azért, hogy utólag tiszta és átlátható legyen a git fánk.



6.12. ábra. Git fa

6.8.2. Issue tracking

Az átláthatóság érdekében a fejlesztési folyamat során **kanban issue boardot** használtunk, amelyen könnyedén tudtuk bármikor ellenőrizni, hogy a csapat éppen milyen feladattal foglalkozik. Ezt a táblát a Jira platformon tartottuk számon, mivel ez az egyik legelterjedtebb, leghasználtabb eszköz a szoftverfejlesztésben. E mellett azért esett a Jirára a választás, mert a platformon ingyenes tudtunk timelinet is csinálni, így egyszerűen tudtuk időzíteni az aktuális sprint feladatainak a leosztását és megoldását.

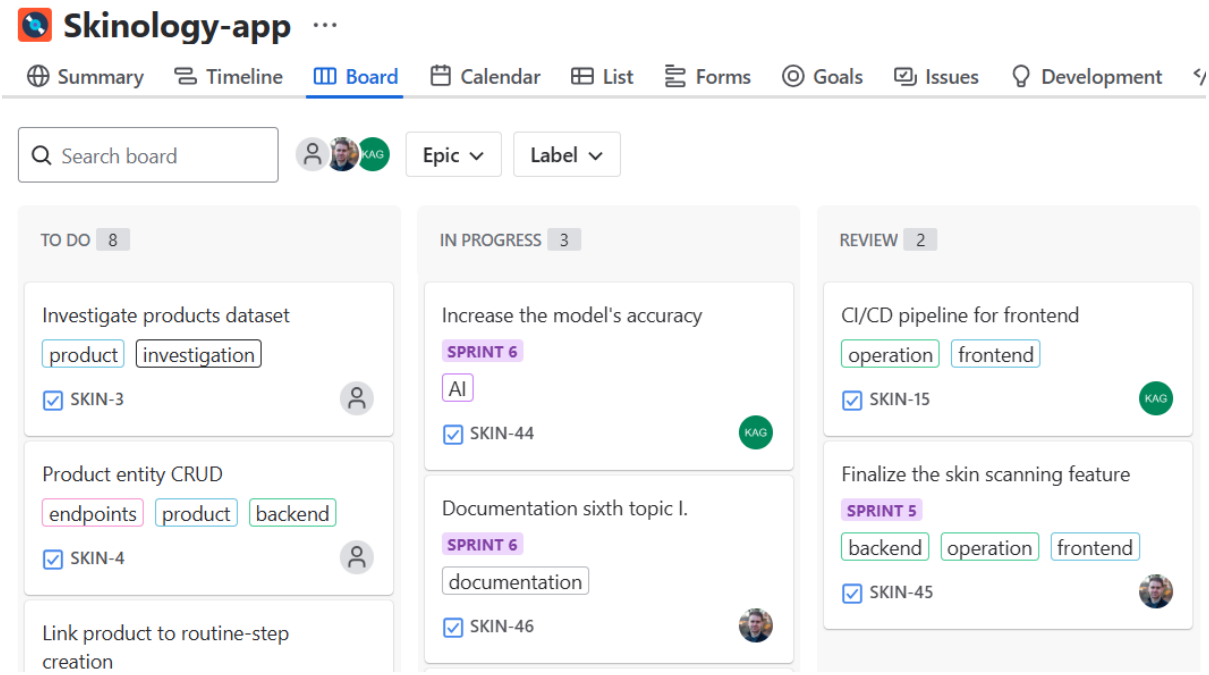
Amint az a 6.13 ábrán is látható a feladatoknak különböző címkéket használtunk, hogy kategorizálni tudjuk őket, mint például documentation, AI, product, operation stb. Továbbá a feladatoknak egy előre meghatározott névkonvenciót használtunk, amely a **SKIN-x** nevet viselte. Ez összhangban volt a GitLabon kezelt feature branchek neveivel is, hogy könnyedén követni tudjuk a haladást. A feladatokat négy nagy kategóriába soroltuk be: **to do**, **in progress**, **review** és **done**.

6.9. Funkcionalitások

6.9.1. Bőrbetegség analizálás

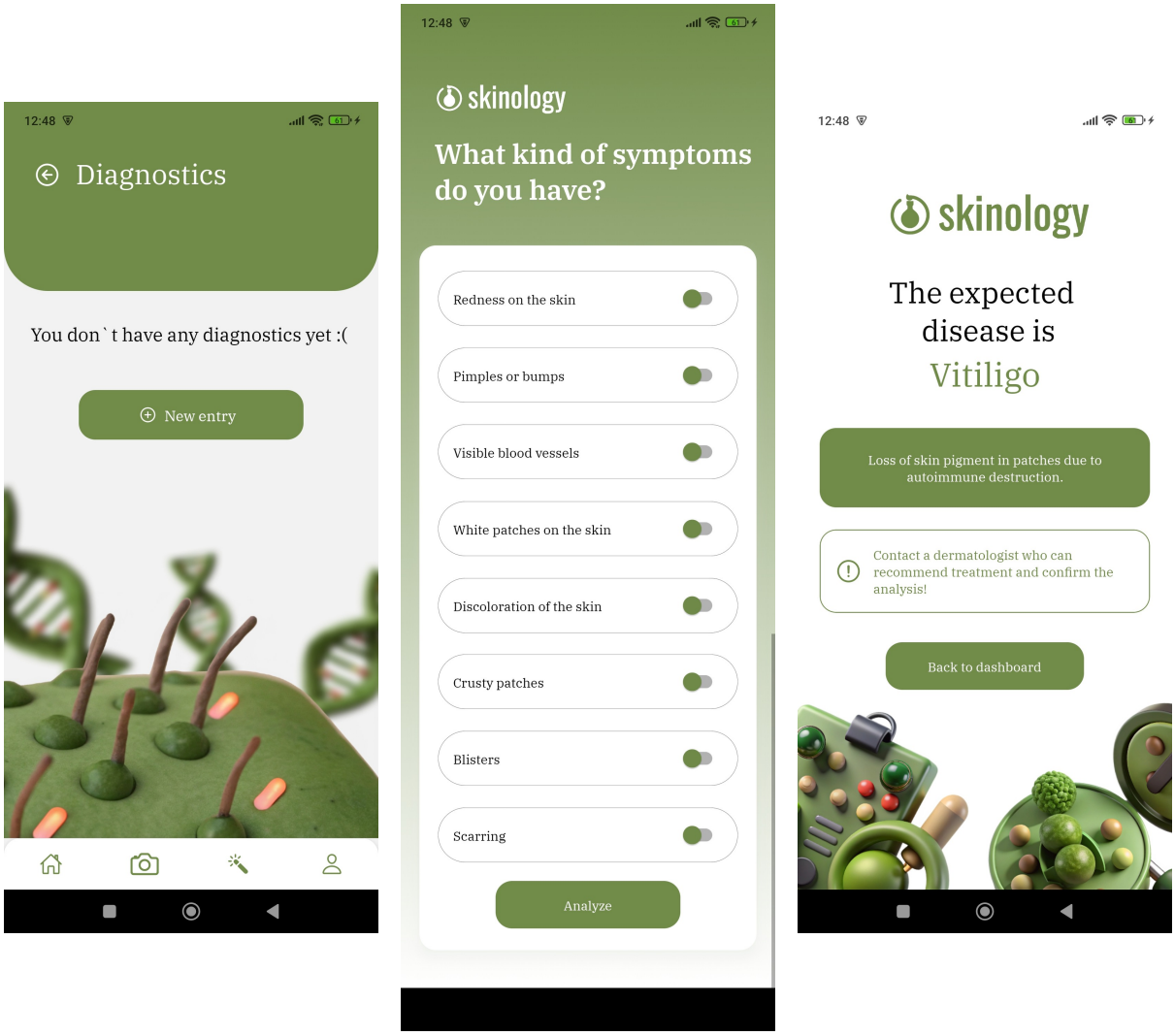
Az alkalmazás egyik fő funkciója a bőrbetegségek analizálása **AI-alapú** megoldásokkal. Amennyiben a felhasználó a PrePhotoScreenről átnavigál a bőrbetegség szkennelésre láthatja az előzetesen felvitt képeit, amelyek szekciókba rendezve vannak megjelenítve és minden képsorozat mellett írja a legutolsó diagnosztizálás eredményét. Abban az esetben, ha még egy diagnosztizálást sem hajtott végre a felhasználó egy placeholder jelenik meg a képernyőn, ahogy az a 6.14 fotón is látható. Figyeltem arra, hogy semmilyen esetben se fogadja a felhasználót üres kijelző, minden use-casere odafigyeltem. Ez amellett, hogy a felhasználói élményt javítja a designt sem rontja el a sarkallatos esetekben.

[25] A commit squash egy Git művelet, amely több commitet egyetlen commitba egyesít. Ez különösen akkor hasznos, ha egy fejlesztési ágon (branch-en) több kisebb, közbenső commit készült, de a végleges verzióban tisztább commit history-t szeretnénk látni.



6.13. ábra. Issue tracking a Jira segítségével

Ha a felhasználó egy már előre létrehozott szekciót szeretne ismét analizálni az AI segítségével, azt az **Analyze** feliratú gombbal megteheti. Abban az esetben, ha teljesen új szekciót szeretne létrehozni - mivel egy új bőrbetegség gyanúja vetődik fel, azt is megteheti a **New entry** gombbal. Ebben az esetben a kamera felület fogadja, ahol le tudja fotózni a sérült bőrfelületet. Ha már előzetesen megvan a fotó a bőrfelületről, abban az esetben a galériából is ki tudja választani. A fotó kiválasztása vagy elkészítése után egy töltőképernyő jelenik meg, amelyen egy személyre szabott animáció indul el, ezzel is javítva a felhasználói élményt, amíg az AI model visszaadja az eredményt a kliens oldalnak. Miután megjött a válasz az AI modeltől a képernyőn - ahogy azt a 6.14 fotó is ábrázolja egy tünetkiválasztó űrlap jelenik meg. Ezt azért gondoltam fontosnak leimplementálni, mivel az AI model által visszaadott három legvalószínűbb betegség tünetei jelennek meg rajta. A felhasználó ki tudja választani a tüneteit - ezzel is pontosítva az eredményt, mivel előzetes kutatásra alapozva a tünetkiválasztás egy fontos eleme a bőrbetegség diagnosztizálásnak. Miután kiválasztotta a tüneteit is az **Analyze** gombot megnyomva az eredmény kijelző jelenik meg az alkalmazásban (lásd 6.14 ábra), ahol a diagnosztizált betegség látható az AI model prediktálása és a tünetkiválasztás után. Fontos volt azt is megjeleníteni ezen a kijelzőn, hogy a páciens keressen fel egy hivatásos dermatológust, mivel az AI model analizálása **nem 100%-ig hiteles**. Amennyiben a felhasználó visszanavigál a bőrbetegségeket összegző oldalra, megjelenik az új szekció az új kezdő fotóval. Fontos megjegyezni, hogy az analizálásnál használtam egy küszöbértéket is. Az AI model, ha csak a küszöbérték alatti precizitással tudja megállapítani a bőrbetegség meglétét, abban az esetben elvetjük a predikció azon részét. Ha a model mind a három legvalószínűbb betegség meglétét csak a küszöbérték alá tudja megítélni, akkor a felhasználó az eredmény képernyőn egy mosolygós arcot lát, amely jelzi, hogy valószínűleg nincs diagnosztizált betegség.

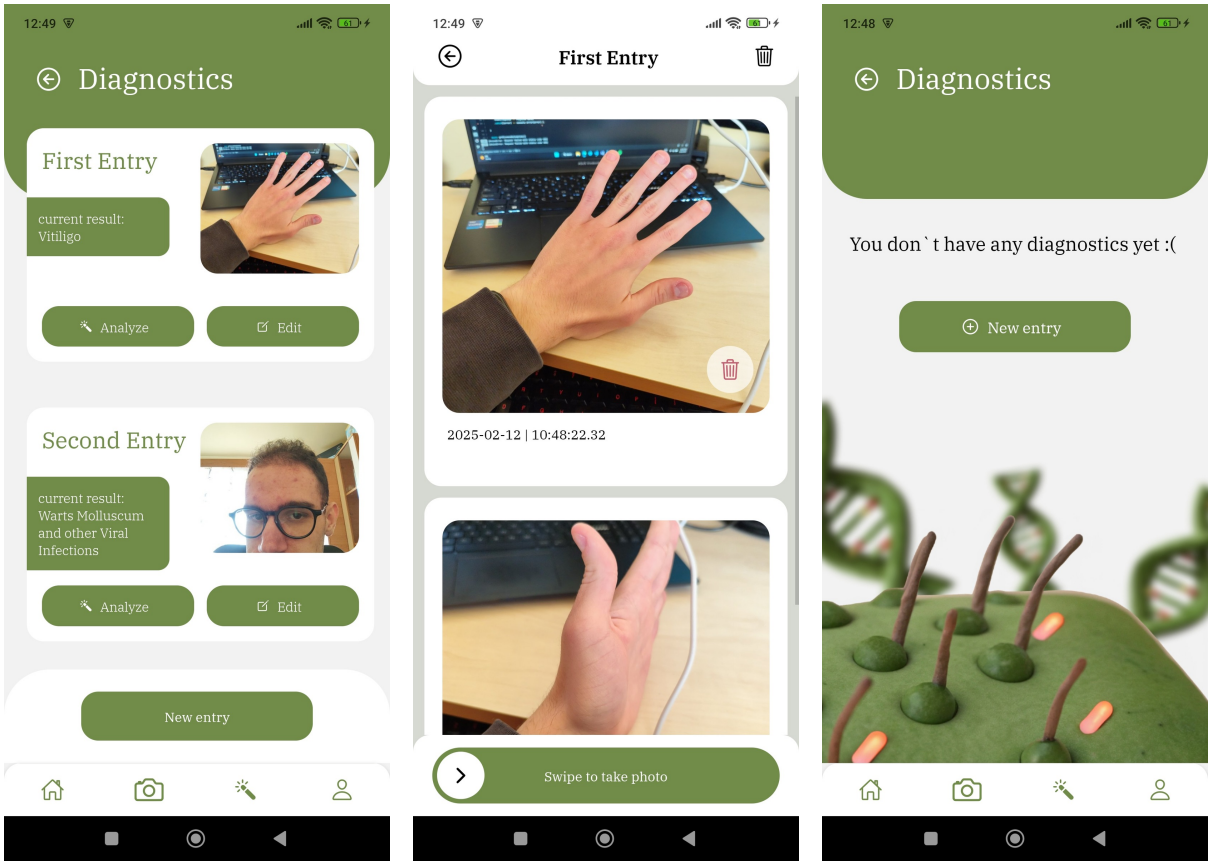


6.14. ábra. Skinology bőrbetegség analizálás funkció

Ha a felhasználó módosítani szeretné az előre létrehozott szekcióját azt az **Edit** gombbal megteheti (lásd 6.15), amiután a 6.15 ábrán látható képernyőn találja magát. Itt lehetősége van a szekciót módosítani, kitörölni fotókat, amelyek már nem relevánsak vagy éppen hozzáadni további képeket. Ezzel egyidejűleg a felhasználó egy naplószerű fotósorozatot lát a bőrfelülete alakulásáról, amely kronológikusan visszanézhető. Minden egyes fotó alatt megjelenik az elkészítés dátuma is. Ha új fotót ad hozzá az adott szekcióhoz végigmegy ismét az analizálás lépésein és a végén a hozzáadott fotóval kiegészült elemzés eredményét látja. Az elemzés folyamatát úgy oldottam meg, hogy irreleváns legyen az, hogy az AI model egy vagy éppen több képet kap a HTTP kérésben, mivel minden esetben végbe tud menni a prediktálás. Ha több fotó (azaz adat) van, hatékonyabb lesz az elemzés, ha kevesebb az input, kevésbé lesz pontos az analizálás.

Fontos megemlíteni azt, hogy a bőrbetegség analizálása bejelentkezéshez kötött, így csak azok a felhasználók férhetnek hozzá a funckióhoz, akiknek van fiókjuk. Amennyiben a felhasználó csak vendégként lép be az alkalmazásba, akkor a 6.15 látható képernyő fogja

fogadni, amely arra próbálja irányítani a felhasználót, hogy jelentkezzen be az applikációba.



6.15. ábra. Skinology bőrbetegség analizálás funkció 2

6.9.2. Bőrápolási rutin

A bőrápolás egyre inkább a modern öngondolkodás és egészségtudatosság szimbólumává vált, ahol a személyre szabott megoldások kulcsszerepet játszanak. A felhasználók nemcsak hatékony, hanem időhöz és életmódhoz igazodó rutinokat keresnek, amelyek rugalmasan alkalmazkodnak a mindennapok változásaihoz. Ezen igényekre válaszul fejlesztettem ki a bőrápolási rutin tervező modult, amely nem csupán egy egyszerű ütemtervező, hanem egy komplex platform, amely lehetővé teszi a felhasználók számára, hogy teljes mértékben saját kezűleg formálják meg bőrápolási szokásaikat. A modul középpontjában a maximális testreszabhatóság és a felhasználóbarát működés áll, miközben technikai hátterében egy robusztus, relációs adatbázis (PostgreSQL) biztosítja az adatok biztonságos és strukturált tárolását.

A piacon elérhető számos alkalmazással ellentétben, ahol a termékek előre definiált adatbázisából lehet választani, az én megoldásomban a felhasználók saját kozmetikumokat adhatnak meg szabad szöveges formában. Ez lehetővé teszi, hogy akár niche márkákat, házi készítésű termékeket vagy egyedi keverékeket is bevonjanak a rutinjaikba, ezzel is hangsúlyozva a teljes kreatív irányítást. Emellett a rutinok nem korlátozódnak általános időkeretekre (pl. „hétköznapok”), hanem pontosan meghatározott napokra (pl. hétfő,

szerda, péntek) tervezhetők, így akár váltakozó munkaidővel vagy egyedi szükségletekkel rendelkező felhasználók is könnyedén használhatják az alkalmazást.

Főbb funkcionalitások:

1. **Rutinok dinamikus détrehozása**:

   - **Napok és időszakok precíz beállítása**
     A felhasználó kiválaszthat konkrét napokat (pl. hétfő, csütörtök, vasárnap) és napszakokat, így a rutinok tökéletesen illeszkednek az egyedi napi rendhez.
   - **Időzítés és emlékeztetők**
     Minden lépéshez hozzárendelhető időtartam (pl. „Hidratálás: 5 perc"), és a rendszer automatikusan kiszámítja a teljes rutin időigényét. Opcionálisan minden rutinhoz külön be lehet állítani értesítést, ami a rutin kezdete előtt 5 perccel jelez a felhasználónak, ezáltal bizosítva, hogy az esetleges kozemtikumokat elő tudja készíteni és időben neki tudjon kezdeni a rutinnak

2. **Kozmetikumok szabad szöveges bevitele**

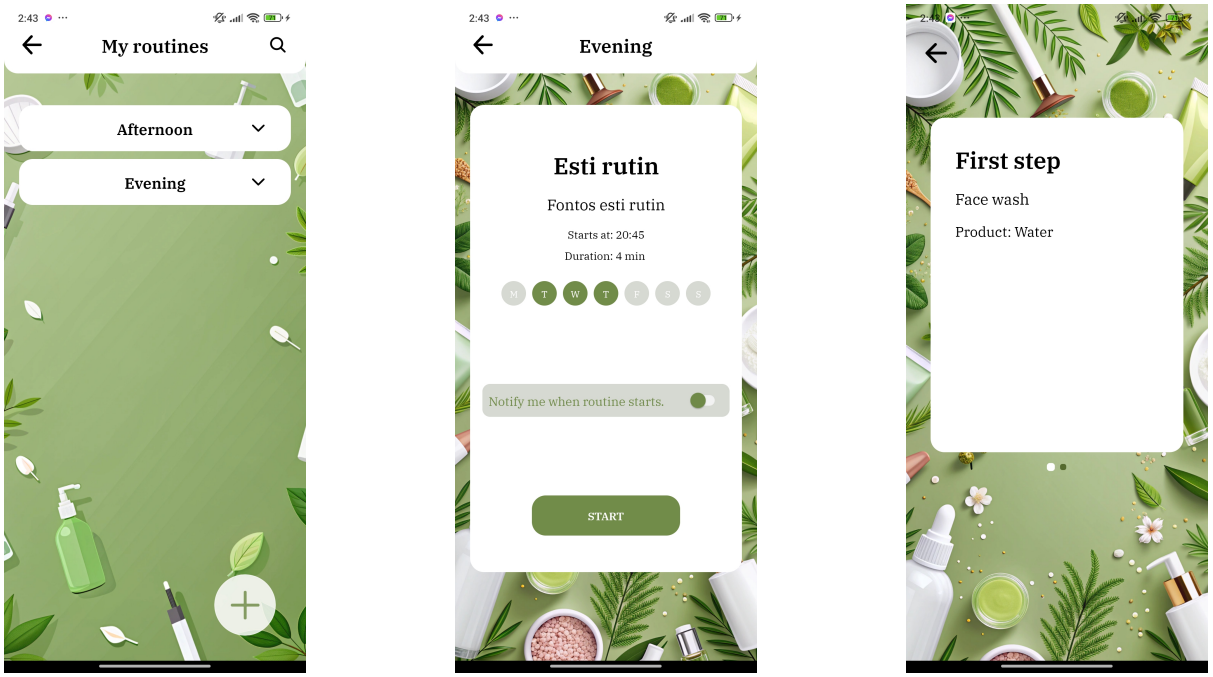
   A felhasználó kézzel írja be a használt kozmetikumok nevét, típusát és egyéb jellemzőit (pl. „C-Vitamin szérum – The Ordinary, 2 csepp"), így nincs korlátozva előre definiált terméklistákra. Ez a funkció kifejezetten értékes azok számára, akik niche márkákat vagy egyedi receptúrákat használnak.

3. **Lépésalapú szerkezet és interaktivitás**:

   - **Egyszerű és átlátható lépésfelépítés**:
     A rutinok létrehozása során a felhasználó egyértelműen hierarchikus szerkezetben definiálja a teendőket (6.16). Minden rutin egy egyedi névvel ellátott konténer (pl. „Relaxáló Esti Bőrápolás"), amelyhez közvetlenül kapcsolódnak az egyes lépések. A lépések nem alkotnak bonyolult fát, hanem sorrendben egymás után adhatók hozzá, így a felhasználó pontosan látja, milyen teendők végrehajtása szükséges a rutin teljesítéséhez
   - **Swipe-olható kártyák**:
     Amikor a felhasználó elindít egy rutint, a lépések vizualizálva jelennek meg kártyák formájában, amelyeket vízszintes gesztussal (balra/jobbra húzással) lehet lapozni. Ez a megoldás nemcsak esztétikus, hanem könnyen követhetővé teszi a folyamatot

4. **Szűrés, keresés és adatkezelés**

   A felhasználó szűrheti a rutinjait napszakok szerint és úgy keresheti a végrehajtani kívánt rutinját, vagy konkrétan rá is kereshet egy konkrét rutinra, ekkor értelemszerűen egyből hozzáfér a keresett rutinhoz



6.16. ábra. Bőrápolási rutinhoz kapcsolódó képernyők

6.9.3. Összetevő szkennelés

A kozmetikai ipar folyamatosan bővül, és a vásárlók egyre inkább érdeklődnek a termékek összetevőinek részletes információi iránt. Az összetevők listájának megértése elengedhetetlen, mivel sokan a bőrtípusuknak megfelelő kozmetikai termékeket keresnek. A célom egy olyan alkalmazás vagy rendszer fejlesztése, amely lehetővé teszi, hogy a felhasználók gyorsan és egyszerűen információt nyerjenek a kozmetikai termékek összetevőiről, kizárólag egy fénykép vagy galériaelem segítségével. A rendszer segítségével a felhasználók megismerhetik a termékek hatását a bőrükre, azok felhasználási területeit, valamint a potenciális előnyöket és hátrányokat.

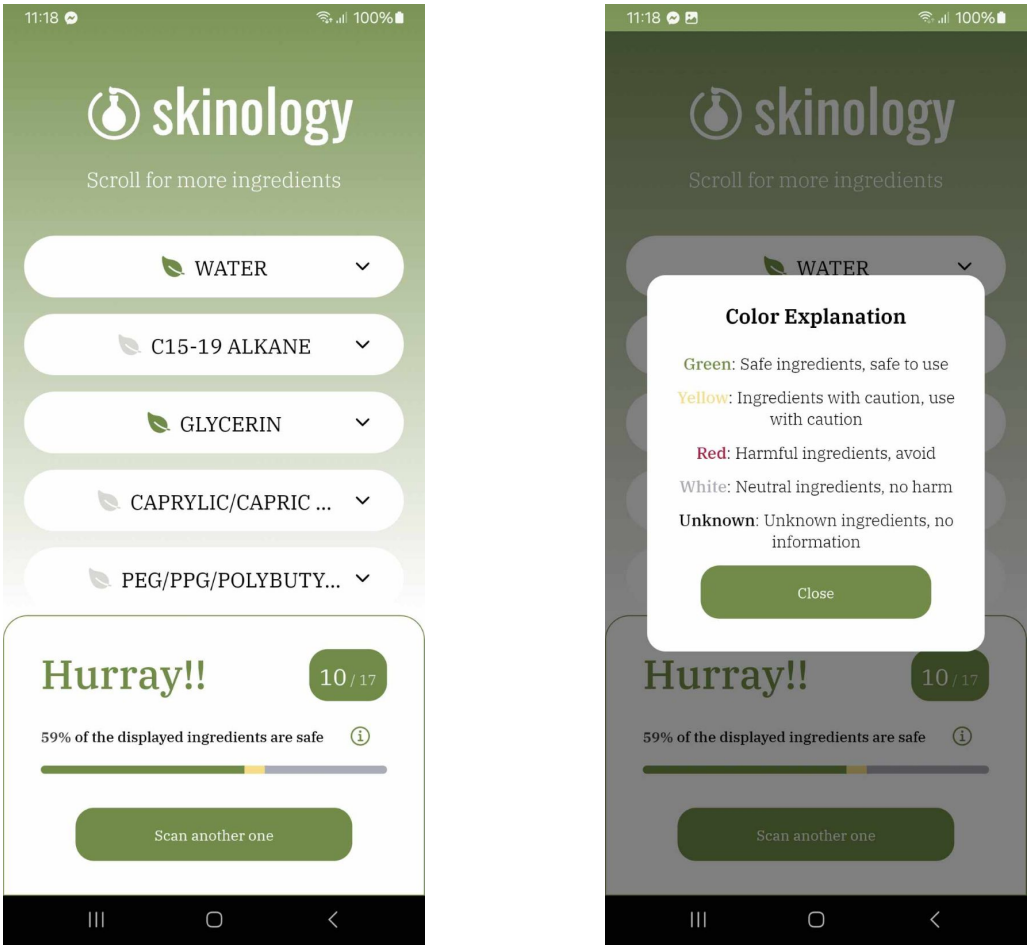
A funkció működése

A felhasználó a kamera segítségével fényképet készíthet egy kozmetikai termékről, amelynek címkéjén az összetevők listája található, vagy a galériából kiválaszthat egy meglévő képet. Ezt követően az alkalmazás az Optikai Karakterfelismerés (OCR)[26] technológia segítségével felismeri és kinyeri a képen található szöveget, amely tartalmazza az összetevők nevét. Miután a szöveg kinyerésre került, a rendszer szabályos kifejezésekkel (regex) dolgozik, amelyek lehetővé teszik az összetevők pontos azonosítását, függetlenül attól, hogy a szöveg milyen nyelven szerepel.

A rendszer a kinyert összetevők alapján információkat jelenít meg a felhasználónak, beleértve:

- **Leírás**:

[26]Az OCR (Optikai Karakterfelismerés) egy technológia, amely lehetővé teszi, hogy a számítógépek képeken vagy szkenneléseken szereplő szöveget felismerjenek és digitális szöveggé alakítsanak



6.17. ábra. Összetevő szkennelés eredmény

Minden kozmetikai összetevő különböző hatásokkal rendelkezik a bőrre, és a rendszer ezt az információt biztosítja a felhasználónak (lásd 6.17). A leírás általában az összetevő alapvető jellemzőit és előnyeit tartalmazza. Például egy hidratáló összetevő esetében a leírás arról tájékoztatja a felhasználót, hogy az összetevő képes megőrizni a bőr nedvességtartalmát, és segít megakadályozni a kiszáradást. Egy ránctalanító összetevő esetén a leírás kiterjedhet arra, hogy az összetevő segít csökkenteni a finom vonalak megjelenését, javítja a bőr rugalmasságát, és serkenti a kollagéntermelést. A leírás segít abban, hogy a felhasználók pontosan megértsék, hogy miért és hogyan használják az adott összetevőt

- **Felhasználás**:

  Az összetevők különböző problémákra és célokra alkalmazhatók, és a rendszer ezt az információt is megjeleníti. A felhasználási terület pontosan meghatározza, hogy az adott összetevő milyen problémák kezelésére ajánlott. Például, ha az összetevő pattanásos bőr kezelésére alkalmas, a rendszer információt adhat arról, hogy hogyan segíti a pórusok tisztítását, csökkenti a gyulladást, és segít megelőzni az újabb pattanások kialakulását. Ha az összetevő öregedésgátló hatással bír, a leírás kitérhet arra, hogy milyen módon csökkenti a ráncokat, javítja a bőr feszességét és ragyogá-

sát, vagy stimulálja a bőr megújulását. Ezen kívül az összetevő alkalmazható lehet például pigmentfoltok kezelésére, bőrnyugtatásra vagy hidratálásra is

- **Bőrtípusra gyakorolt hatás**:

  Mivel minden bőrtípus más-más igényekkel rendelkezik, az összetevők hatása is változó lehet. A rendszer az adott összetevő hatását a különböző bőrtípusokra is megjeleníti. Például egy olajmentes, pórusösszehúzó összetevő ideális lehet zsíros bőrre, mivel segít szabályozni a faggyútermelést, és megelőzni a pattanások kialakulását. Ezzel szemben egy gazdag hidratáló összetevő, mint a hialuronsav, száraz bőrre ajánlott, mivel segít megkötni a vizet a bőrben és biztosítja annak megfelelő nedvességtartalmát. Az érzékeny bőrre ajánlott összetevők gyakran nyugtató hatásúak, és segítenek csökkenteni az irritációkat, míg a normál bőr számára olyan összetevők javasoltak, amelyek fenntartják a bőr egészséges egyensúlyát.

7. fejezet

Összefoglaló

A projekt keretén belül egy olyan szoftverrendszer megvalósítására került sor, amely elősegíti a felhasználók mindennapjait, ha bőrápolásról van szó. Az alkalmazás segítségével a felhasználók tudatosabban és megfontoltan tudnak bőrápolási termékeket vásárolni és minél jobban óvni tudják a bőrüket. Megtekinthetik és kielemezhetik részletesen a bőrápolási termékeik összetevő listáját és többletinformációt nyerhetnek, arról hogy pontosan mi kerül a bőrükre.

Továbbá az alkalmazás lehetőséget nyújt a felhasználóknak, hogy gyorsan -pár fotót készítve, kielemezzék a bőrfelületüket és megelőzzék a felesleges időpontfoglalást a dermatológusuknál, ezzel időt spórolva maguknak. Amennyiben az alkalmazás bőrbetegséget diagnosztizál, csak abban az esetben küldi el a pácienst a megfelelő dermatológushoz.

Összegzésként egy olyan alkalmazás létrehozására került sor, amely emberközeli és segítőkész az emberek, a társadalom számára, mivel személyes tapasztalat alapján éreztük azt, hogy létfontosságú egy ilyen alkalmazás létrehozása és nagymértékben megkönnyiti az emberek hétköznapjait bőrápolási szempontból.

7.1. Továbbfeljesztési lehetőségek

Szeretném, ha az alkalmazás több lenne, mint egy bőrelemző eszköz – egy teljes körű bőrápolási partnerré válna, ahol a felhasználók nemcsak diagnózist kapnak, hanem közösséget, személyre szabott segítséget és motivációt is.

Első lépésként egy kölcsönös támogatást nyújtó közösségi platformot építenék ki. Itt a felhasználók megoszthatnák tapasztalataikat, kérdéseiket tehetnének fel, és akár szakértőktől is kérhetnének tanácsot. Egy egyszerű beszélgetőfelületen túl lehetőséget teremtenék orvosi konzultációra is, akár videóhíváson keresztül, hogy komolyabb problémák esetén professzionális segítséget lehessen kapni. Ezt kiegészíteném egy bőrnapló funkcióval, ahol a felhasználók nyomon követhetik az idő múlásával a bőrüket. Az AI nemcsak elemzi a változásokat, hanem hasznos tanácsokat is adhat.

Végül, hogy a felhasználók kitartóak legyenek, egy játékosított rendszert is beépítenék. Kihívások, jutalmak és egyértelmű célok segítenék őket abban, hogy hosszú távon is tartsák a bőrápolási rutinjukat.

Ábrák jegyzéke

Kódrészletek jegyzéke

Irodalomjegyzék

[BEH+13] Ann Chang Brewer, Dawnielle C. Endly, Jill Henley, Mahsa Amir, Blake P. Sampson, Jacqueline F. Moreau, and Robert P. Dellavalle. Mobile applications in dermatology. *JAMA Dermatology*, 149(11):1300–1304, 11 2013.

[Coo23] Anabel Cooper. 2023's most popular beauty google searches, 2023.

[FHR+10] J.J.J. Fu, G.G. Hillebrand, P. Raleigh, J. Li, M.J. Marmor, V. Bertucci, P.E. Grimes, S.H. Mandy, M.I. Perez, S.H. Weinkle, and J.R. Kaczvinsky. A randomized, controlled comparative study of the wrinkle reduction benefits of a cosmetic niacinamide/peptide/retinyl propionate product regimen vs. a prescription $0 \cdot 02\%$ tretinoin product regimen. *British Journal of Dermatology*, 162(3):647–654, 03 2010.

[GKYH20] Manu Goyal, Thomas Knackstedt, Shaofeng Yan, and Saeed Hassanpour. Artificial intelligence-based image classification methods for diagnosis of skin cancer: Challenges and opportunities. *Computers in biology and medicine*, 127:104065, 2020.

[WYP+24] Shannon Wongvibulsin, Matthew J. Yan, Vartan Pahalyants, William Murphy, Roxana Daneshjou, and Veronica Rotemberg. Current state of dermatology mobile applications with artificial intelligence features. *JAMA Dermatology*, 160(6):646–650, 06 2024.