TDK-dolgozat

Gordos Bence

Koordináta-alapú szoftverfejlesztés a precíziós mezőgazdaság támogatására

Gordos Bence, IV. évf. gazdaságinformatikus BSc
Témavezető: Dr. habil. Kertész Attila
Szegedi Tudományegyetem
Természettudományi és Informatikai Kar, Szoftverfejlesztés Tanszék

Tartalomjegyzék

1. BEVEZETŐ

A mezőgazdaság területén történő digitalizáció és technológiai fejlődés növekvő jelentőséggel bír. A precíziós gazdálkodás lehetővé teszi a gazdák számára az erőforrások hatékonyabb felhasználását, ami hatékonyabb termeléshez vezet. Kutatások azt is bizonyítják, hogy a precíziós mezőgazdasággal elősegíthető a gazdaságok külső inputjainak környezetkímélő kezelése helyspecifikus adatok felhasználásával [1]. Ez a megközelítés lehetővé teszi a műtrágyák, vetőmagok, rovar- és gyomirtók célzott, szükségletekhez igazított alkalmazását, minimalizálva a felesleges felhasználást, miközben biztosítja a gazdálkodás jövedelmezőségét. Például a nitrogén helyspecifikus menedzsmentje csökkentheti a felhasznált nitrogén mennyiségét, míg a rovar- és gyomirtók csak ott kerülnek alkalmazásra, ahol valóban indokolt.

Magyarországon a precíziós gazdálkodás alkalmazása más fejlett országokhoz viszonyítva rendkívül alacsony. A Központi Statisztikai Hivatal (KSH) adatai szerint 2020-ban a gazdaságok mindössze 12%-a használt valamilyen precíziós eszközt, ez az arány Dániában 23%-volt 2018-ban [2]. A precíziós gazdálkodási technikák közül az egyik legnépszerűbb a mezőgazdasági gépek automata kormányzását elősegítő rendszerek. A KSH felmérései szerint 2023-ban a magyar gazdaságok 5,4%-a használt ilyen technológiát [3], ez a 2020-as értékekhez viszonyítva 1,4%-os növekedést mutat. Ez az arány azonban az USA-ban működő legkisebb gazdaságok körében is 25% körüli volt 2019-ben, a nagyobb gazdaságok körében pedig 64.5%-72.9% becsülik ugyanabban az időben [4].

A KSH eredményei rávilágítanak arra is, hogy a gazdák egyre több százaléka tartaná hasznosnak és jövedelmezőnek a precíziós gazdálkodási technológiák használatát, de ezt különböző hátráltató tényezők miatt nem tették meg. A legnagyobb akadályt a magas belépési, illetve működtetési költségek, valamint az ilyen eszközökről formált tapasztalatok, információk terjedésének alacsony mértéke gazdálkodók között jelentik [5,6]. A precíziós eszközök használata sokszor magas háttértudást igényel a felhasználóktól, amire azok nincsenek felkészítve. Ez a gazdaságok számára azt jelenti, hogy bizonyos technológiák bevezetésekor, a belépési költség mellett, a meglévő munkaerő betanításából adódó költségek is felmerülhetnek, vagy akár új munkaerő felvétele szükséges a precíziós eszközök használatához. Mindez ahhoz vezet, hogy kisebb méretű gazdaságoknak sokszor nem megtérülő befektetés a precíziós mezőgazdaság.

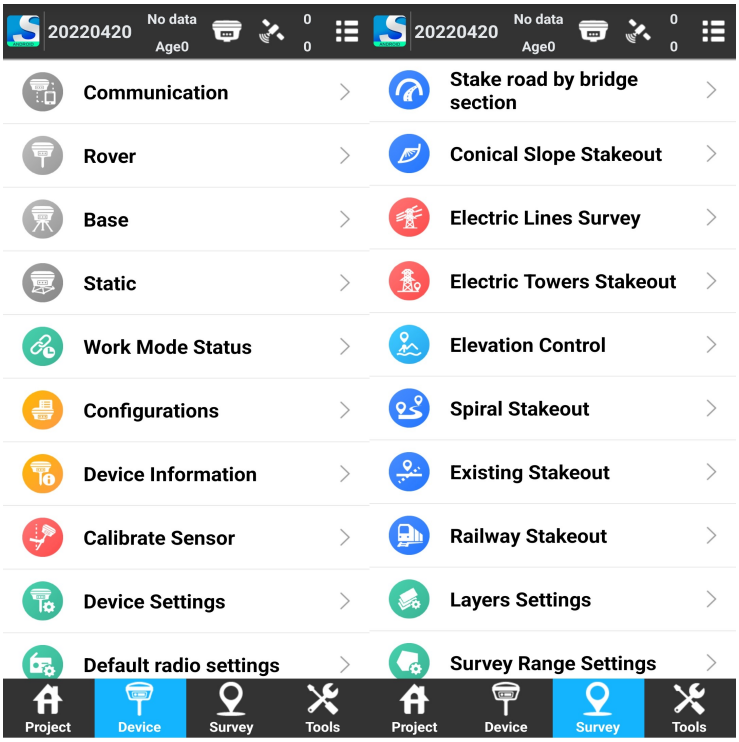
A földrajzzal és mezőgazdasággal kapcsolatos háttértudásra nem csak a felhasználóknak, hanem a szoftverfejlesztőknek is rendelkezniük kell, hogy ilyen rendszereket

fejlesszenek. Az ebből adódó fejlesztési nehézségek hátráltathatják innovációt és új megoldások kialakulását.

A dolgozat célja egy olyan precíziós mezőgazdasági projekt megvalósítása, ami egyszerűsített felülettel lecsökkenti betanulásból és működtetésből eredő költségeket a gazdaságok számára, valamint kiemelni egy ilyen projekt megvalósításakor felmerülő szoftverfejlesztési akadályokat, azokra megoldást kínálni.

A modern gazdálkodást segítő alkalmazások alapját a mezőgazdasági földterületekről származó adatok képzik. Ezért az agrárszektorban alkalmazott mezőgazdasági szoftvereknek, amelyek földrajzi koordinátákkal dolgoznak, kulcsfontosságú szerepük van a precíziós gazdálkodásban. A projekt célkitűzése egy olyan szoftver csomag megvalósítása, ami képes földrajzi koordináták bemérésére, azokkal történő számolásra, valamint a termőföldekről mezőgazdasági szempontból hasznos adatok tárolására.

A koordináták pontos beméréséhez egy földmérő szoftverre van szükség. A jelenleg piacon lévő ilyen alkalmazások nem kimondottan mezőgazdasági felhasználásra lettek tervezve, hanem általános földmérési feladatok ellátására. Ezek az alkalmazások természetesen használhatók mezőgazdasági területek beméréséhez is, de a célközönségük földmérő szakemberek. Ebből adódóan az alkalmazások beállítások és többlet funkciók sokaságával vannak felszerelve, ami egy laikus felhasználó számára összezavaró és egy agrár szempontból felesleges bonyolultsági réteget ad az alkalmazásnak. Ezeket a beállításokat és többletfunkciókat mutatja az 1.1. Ábra egy népszerű, több mint százezer letöltéssel rendelkező földmérő alkalmazásból [7].



1.1. Ábra. SurPad4 alkalmazás képernyőképei

Mindezen felül ezek az alkalmazások nem alkalmasak mezőgazdasági szempontból fontos adatok rögzítésére, például a termőföldbe vetett takarmánynövény típusának rögzítésére. Ezeket az adatokat egy külön alkalmazásban kell rögzíteni, ami azt jelenti, hogy a nehezen bemért földrajzi adatokat kettő vagy több különböző alkalmazásban kell vezetni és több különböző felületet kell megtanulnia a felhasználónak használnia.

2. FŐBB FUNKCIÓK

2.1. Mobilalkalmazás

A mobilalkalmazás fő célja a földrajzi koordináták pontos meghatározása, bemérése, födlmérési funkciókat is megvalósítva. Az alkalmazás csak a mezőgazdasághoz fontos földmérő funkciókat látja el, ezzel egyszerűsítve a felületét egy hagyományos földmérő alkalmazáshoz képest. Egy hagyományos földmérő alkalmazással ellentétben képes kommunikációt folytatni webalkalmazás párjával, amivel a rögzített adatokat szinkronizálhatja. Három fő képernyőből áll: Projekt, Mérés és Eszközök. Projekt oldalon találhatók a pontok CRUD operációi és webszerver szinkronizáció, Mérés fülön a földmérés, az Eszközök fülről pedig a koordináta mérések (távolság-, területszámítások, …), NTRIP- és általános beállítások, valamint a webszerver be- és kijelentkezés funkciók érhetők el.

A mobilalkalmazás felületének kialakításában a felhasználók várható életkorát is figyelembe kell venni. Magyarországon a gazdatársadalom sajnos idősödő. A gazdaságirányítók 60%-át 55 vagy annál idősebb személyek alkotják [3]. A gazdaságirányítók körében a 65 vagy annál idősebbek a legnépesebb réteg, 36,9%-kal. Az alkalmazásnak olyan felülettel kell rendelkeznie, ami egy ilyen célközönség számára is könnyen kezelhető, értelmezhető. Mobilalkalmazásoknak célszerű ilyen esetben többek között nagyobb méretű gombokat, feliratokat, extra megerősítő ablakokat használniuk [8].

2.1.1 NTRIP kliens

A centiméteres pontosságú helymeghatározás elérése elengedhetetlen a precíziós mezőgazdaságban. Az okostelefonok GPS pontossága általában 5 méter [9], ezért önmagában nem alkalmas földmérési célokra. A centiméteres pontosság eléréséhez RTK korrekciót kell alkalmazni [10].

Az RTK egy differenciális GNSS **(**Global Navigation Satellite System**)** technológia, amely magas szintű pontosságot biztosít. Az RTK GPS rendszerek fix helyzetű bázisállomásokról érkező adatokat használnak a pontosság növelése érdekében. A GPS jelek tartalmaznak bizonyos hibákat, amelyek csökkentik a pontosságot, például az atmoszférikus interferencia, amely zavarhatja a jel útját a műholdtól a vevőig. Ha rendelkezésre áll egy rögzített pozíciójú bázisállomás, aminek ismerjük a pontos koordinátáit, akkor ez a hiba kiszámítható a pontos és a mért koordináták különbségéből. Ha ezt a hibát, vagyis korrekciót valós időben juttatjuk el egy ismeretlen ponton mozgó vevőbe, akkor ez az eljárás egy valós idejű kinematikus eljárás (Real Time Kinematic - RTK) lesz [11]. Az RTK korrekciók jellemzően internetkapcsolaton keresztül jutnak el a vevőhöz. A korrekciós adatokat RTCM (Radio Technical Commission for Maritime Services) formátumban kell továbbítani NTRIP (Networked Transport of RTCM via Internet Protocol) protokoll segítségével.

Összegzésképpen, a teljes folyamathoz szükséges egy fix helyzetű és ismert pozíciójú bázisállomás, egy szerver, amely képes a bázisállomásról érkező RTCM jelek fogadására és továbbítására a vevő felé, valamint egy vevőegység, amely képes ezeknek a jeleknek a fogadására és a pontos pozíció meghatározására. A vevőnek rendelkeznie kell egy RTK-kompatibilis GPS egységgel, azaz egy dual-frequency GNSS vevőegységgel, amelyet a legtöbb okostelefon nem támogat. Ezért az okostelefonon kívül szükséges egy külső vevőegység is, amely az RTCM jelek alapján kiszámítja a pontos helyzetet. Ezzel a külső vevőegységgel a mobilalkalmazás Bluetooth-on keresztül kommunikál. A mobilalkalmazás ilyen módon egy

NTRIP kliens, ami az RTCM jeleket megkapja a szervertől, Bluetooth-on keresztül továbbítja azokat a külső vevőegység felé, majd visszakapja a pontosított lokációkat, amiket földmérés során használni tud.

Az okostelefonon kívül szükséges eszközök/szolgáltatások beszerzése történhet több módon is. RTK bázis állomás és NTRIP Caster (szerver) beszerzése akár ingyenes is lehet (RTK2go.com [12]), de fizetős szolgáltatások is léteznek. Magyarországon ilyet a Lechner Tudásközpont szolgáltat [13], a dolgozat írásakor éves 150.000 Ft + ÁFA áron, de más árkonstrukciók is léteznek. A legnagyobb befektetést igénylő eszköz az RTK Reciever, azaz az eszköz, amire az okostelefon Bluetooth-on keresztül csatlakozik. Az ilyen eszközök akár az 1000-4000 eurós árat is elérhetik [14], de léteznek olcsóbb megoldások is, például Arduino kitek, amik 200-700 eurós ártartományban mozognak [15].

2.1.2 Földmérés

A földmérés funkció lehetővé teszi pontos helymeghatározás alapján koordináták felvételét és kitűzését. A felhasználó pozícióját és a már korábban elmentett koordinátákat az alkalmazás egy műholdas képű térképen jeleníti meg. A műholdas kép nagyban segíti a földmérését, hiszen így látszódnak termőföldek és az őket elválasztó földutak, ami egy hagyományos térképen, vagyis utcai nézetű térképen csak egyszerű zöld területként jelenne meg.

Egy koordináta kitűzésekor az alkalmazás segít a kitűzött pont lokalizálásában és centiméter pontos helymeghatározásában. Ilyenkor a bal alsó sarokban megjelenik egy ábra, ami mindig az aktuális pozícióhoz képest mutatja, hogy a kitűzött pont milyen irányba és távolságra helyezkedik el. Ahogy a felhasználó egyre közelebb ér a ponthoz a térkép nézet kicserélődik két másik képernyőre a kitűzött pont közelségétől függően. Erre azért van, szükség, mert egyrészt jelzés értékű a felhasználó számára, másrészt a térképeken nem lehet annyira közelíteni, hogy a deciméteres és centiméteres távolságok jól látszódjanak. A koordinátákra minden CRUD művelet megvan valósítva. Ezeket a funkciókat a térkép felületen kívül a Projekt oldalon is elérjük. Lehetőség van továbbá .csv fájlból történő importálásra és .csv fájlba történő exportálásra is. A koordinátákat bejelentkezés után szinkronizálhatjuk a webalkalmazással, vagy teljesen felülírhatjuk a mobilalkalmazásban tárolt pontokat a webalkalmazástól kapott pontokkal.

Az alkalmazásban lehetőség nyílik különböző mérések elvégzésére. A már korábban felvett koordinátákkal számolva meghatározható két pont közötti távolság, két pont osztópontja (például felezőpontja), két pontból húzott egyenes északhoz viszonyított szöge, valamint

poligon területe. Ezek a számolások mind mezőgazdasági szempontból hasznos információk lehetnek. A terület és távolság meghatározás hasznos pusztán informatív jelleggel is, de a többi számítás is hasznos többek között például önvezető mezőgazdasági gépek beállításához.

2.2. Webalkalmazás

A webalkalmazás fő funkciója mezőgazdasági adatok rögzítése, megjelenítése, valamint a koordinátákkal történő számolások. Webalkalmazásban van lehetőség a rögzített pontokat föld egységekbe szervezni. Ezekről a föld egységekről számos hasznos információ rögzíthető például a bele vetett növény típusa, permetezése, aratás stb. A földeket és a különálló pontokat is, a mobilalkalmazáshoz hasonlóan, egy műholdképes térképen jeleníti meg. A térkép felületen lehetőség van továbbá új pontok felvételére is, és a pontokkal való számításokra is.

A webalkalmazásban is megtalálható minden olyan számolási funkció, mint a mobilalkalmazásban, azaz két pont közötti távolság, két pont osztópontja (például felezőpontja), két pontból húzott egyenes északhoz viszonyított szöge, valamint poligon területe. A mobilalkalmazással ellentétben ezen a felületlen sokkal kényelmesebben végezheti a felhasználó ezeket a műveleteket. Természetesen előnyt jelent a nagyobb képernyő és egér használata, de ezen felül a webalkalmazás kirajzolja a kiválasztott pontok által bezárt poligont vagy szakaszt, amivel számolni szeretne a felhasználó, ezzel egyértelműbbé téve a folyamatot.

További különbség a mobilalkalmazással szemben, hogy a térképre történő kattintással fel lehet venni új pontokat, anélkül, hogy a felhasználó tudná annak koordinátáit vagy kimenne terepre bemérni a mobilalkalmazással. Ezek a pontok használhatók számításokhoz, és el is menthetőek az adatbázisba. Ezzel a módszerrel akkor is biztosítva van egy kevésbé pontos számolási, mérési lehetőség, ha a felhasználónak nincs lehetősége kimenni terepre a pontos koordináták beméréséhez.

A webalkalmazásban minden fontos mezőgazdasági művelethez és földegységhez tartozó adat CRUD operációja megvannak valósítva. Ezeket az adatokat Form és Datatable elemeken keresztül módosítani, listázni. Ezek az adatok a következők: Földekhez-, Pontokhoz-, Gépekhez-, Aratásokhoz-, Vetésekhez-, Permetezésekhez-, Talajmunkákhoz-, Trágyázásokhoz tartozó adatok.

3. ARCHITEKTURÁLIS TERV

3.1. Webalkalmazás

A webalkalmazásnak képesnek kell lennie komplexebb JavaScript elemek megjelenítésére (például a térkép), valamint képesnek kell lennie egy mobilalkalmazás kiszolgálásra is. Ezért célszerű külön frontendre és backendre bontani.

A backend-et Python-ban, Django Rest Framework keretrendszer segítségével valósítottam meg. A nyelv kiválasztásában nagy szerepet játszottak azok a könyvtárak, amik nagyban segíthetik a földrajzi adatok feldolgozását. A tudományos közösség, beleértve a geográfusokat is előszeretettel használják a Python nyelvet, ezért a legnépszerűbb földrajzzal kapcsolatos könyvtárak zöme ezen a nyelven van megvalósítva. Mindemellet a Python a webfejlesztők körében is népszerű nyelv, sok webfejlesztési keretrendszer áll a fejlesztők rendelkezésére. A három legnépszerűbb ilyen keretrendszer jelenleg a Django Rest Framework, Flask-RESTful és a FastAPI. Ezek a keretrendszerek közül a FastAPI és a Flask-Restful a könnyűsúlyúbb rendszerek, a kettő közül FastAPI a leggyorsabb és legegyszerűbb. FastAPI a projekt kitűzött céljainak eléréséhez elegendő lenne, de a robusztusabb Django Rest Framework olyan, a projekt szempontjából is fontos előnyökkel, valamint fejlesztői kényelmi funkciókkal rendelkezik, amik gyorsíthatják a fejlesztést és könnyedén lehetővé teszik az esetleges jövőbeli fejlesztéseket. Ilyen előny az out-of-the-box admin felület, felhasználókezelés, autentikáció és a magas skálázhatóság [16]. A legnagyobb előnye viszont az, hogy könnyedén konfigurálható GIS adatok támogatására. A GIS adatok hasznosak lehetnek például nedvesség kimutatására bizonyos földterületeken és növekvő igény mutatkozik olyan precíziós mezőgazdaságban használható alkalmazásokra, amik képesek ilyen adatokkal dolgozni [17]. Ezeket az adatokat adatbázisban is képes tárolni és ezeken az adatokon beépített query-ket futtathatunk, például két koordináta közötti távolágra szűrhetünk. Mindez könnyedén elérhető, pár sornyi kóddal, ha az alkalmazás beállításaiban bekonfiguráljuk a GeoDjango-t [18]. A projekt jelenlegi célkitűzéseihez nem volt szükség GIS adatok tárolására, valamint nagy hangsúlyt fektetve a precíziós számolásokra a GeoDjango beépített koordinátákkal való számolásaira sem. Jövőbeli fejlesztések során azonban a GIS adatok rendkívül egyszerű kezelése nagy előnyt jelent más keretrendszerekkel szemben. Az adatbázis kiválasztását is leegyszerűsíti, ha a jövőben elképzelhetően GIS adatokat tárolna az alkalmazás. A Django Rest Framework támogatja a PostgreSQL, MariaDB, MySQL, Oracle, illetve SQLite adatbázisokat, de a GeoDjango legtöbb műveletéhez PostgreSQL adatbázist igényel. Megkönnyítve egy esetleges későbbi migrációt,

az API PostgreSQL adatbázist használ. Az API fejlesztésekor arra is gondolni kellett, hogy bizonyos számolási műveletek időigényesek lehetnek. Ilyen művelet lehet az, ha egyszerre több termőföld területét kell kiszámolni, például, ha földeket CSV fájlból importálja a felhasználó. Az ilyen feladatok végrehajtására aszinkron feladat kezelő rendszert kell használni. Pythonban, mint minden másra, ezen feladat ellátására is több opció van. Én a Celery mellett döntöttem, mert könnyen integrálható Django-val, valamint ideális nagyobb terhelés ellátására is, ha a jövőben erre szükség lenne. Szükség van egy üzenetközvetítőre is az API és a Celery között, hogy az aszinkron feladatkezelés működhessen. Itt is több opció közül kellett dönteni, a legnépszerűbb választások hasonló felhasználási területeken a RabbitMQ és a Redis. A RabbitMQ funkciógazdag, robosztus üzenetkezelést biztosít. Sok funkcióra viszont nincs szüksége a projektnek és csak extra komplexitást adna az integrálás során. A Redis ezzel szemben egyszerűbb és gyorsabb. A Redis fő felhasználása a gyorsítótárazás, de emellett üzenetközvetítőként is használható. Ez egy plusz előnye a RabbitMQ-val szemben, hiszen így egy eszközt kell integrálni gyorsítótárazásra és üzenetközvetítésre is. Bár az alkalmazásnak jelenleg nincs szüksége gyorsítótárra, de a jövőben, például GIS adatok használatakor még szüksége lehet. Ezért a Redis-t választottam üzentközvetítésre.

A webalkalmazás másik része a frontend. A frontend keretrendszer kiválasztása nincs hatással az alkalmazás érdemi működésére, ezért ez személyes preferencia alapján került kiválasztásra. A React.js mellett döntöttem, mert jó dokumentációval rendelkezik és alkalmas térképek, formok, datatable-ek megjelenítésére.

3.2 Konténerizáció

A könnyebb és portabilisabb kitelepítés miatt a webalkalmazás dockerizálva lett. A Docker egy nyílt forráskódú platform, amely lehetővé teszi alkalmazások konténerizált környezetben történő futtatását. A konténerizálás azt jelenti, hogy az alkalmazás és annak összes függősége egy elszigetelt, könnyen hordozható egységben fut, amit konténernek nevezünk. Ennek köszönhetően a Docker lehetővé teszi az alkalmazások gyorsabb, egyszerűbb és megbízhatóbb telepítését bármilyen környezetben, legyen az fejlesztési, tesztelési vagy éles környezet. A Docker egyik fő előnye az, hogy a konténerek könnyen áthelyezhetők egyik rendszerről a másikra, mivel magukban hordozzák az alkalmazás összes szükséges komponensét. Ez azt jelenti, hogy a konténer mindenhol ugyanúgy működik. Emellett a Docker használata segít a rendszer erőforrásainak hatékonyabb felhasználásában, mivel a konténerek gyorsabban indulnak, és kevésbé terhelik a rendszert, mint a hagyományos virtuális gépek. A konténerek könnyen skálázhatók, és ezen kívül a Docker integrálható CI/CD (folyamatos

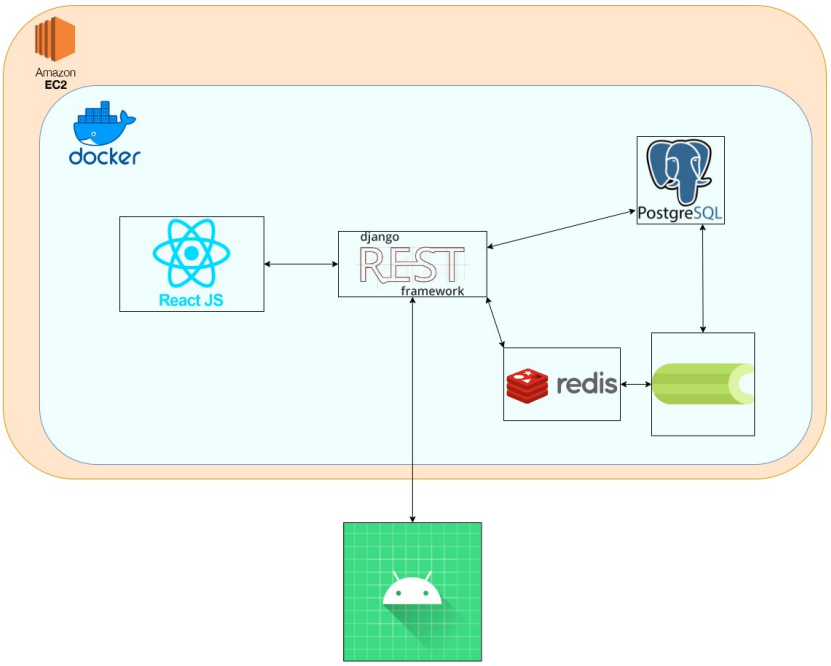
integráció és folyamatos telepítés) folyamatokba, megkönnyítve ezzel az automatikus tesztelést és telepítést. Összességében a Docker megbízhatóságot, gyorsaságot és egyszerű kezelhetőséget biztosít, mely előnyök mind hozzájárulnak a hatékonyabb fejlesztési és üzemeltetési folyamatokhoz.

3.3. Mobilalkalmazás

A mobilalkalmazás egy natív Android alkalmazás. Az alkalmazás Java nyelven lett megvalósítva az Android SDK használatával. Java-t választottam, mert ezen a nyelven is megtalálható pár olyan könyvtár, amik a földrajzi adatokkal való számolást segíteni tudják. Ezen felül az NTRIP kliens megvalósításában is nagy segítséget nyújt egy könyvtár, ami csak Java nyelven érhető el. Az alkalmazás nem használ saját relációs adatbázist. Tud kommunikálni az API-val, a mobileszközön tárolt adatokat pedig az alkalmazás saját memóriájában tárolja.

3.4. Összegzés

A webalkalmazás egy Django REST framework-ben megvalósított REST API-ból és egy React.js-ben megvalósított frontendből áll. Adatbázisként PostgreSQL-t használ, az aszinkron feladatkhoz Celery worker-eket használ, amikkel Redis-en keresztül kommunikál. Mindez dockerizálva fut a felhőben. Kitelepítéshez Amazon Web Services szolgáltatásit vettem igénybe, ezt a folyamatot a 6. fejezetben fogom kifejteni. A mobilalkalmazás egy natív Android alkalmazás, Javában írva. Saját adatbázist nem használ, a működéshez szükséges adatokat az alkalmazás saját memóriájába menti el, valamint a kitelepített REST API-val kommunikál és szinkronizálja az adatait. Az összesített tervet a 3.1 Ábra mutatja.

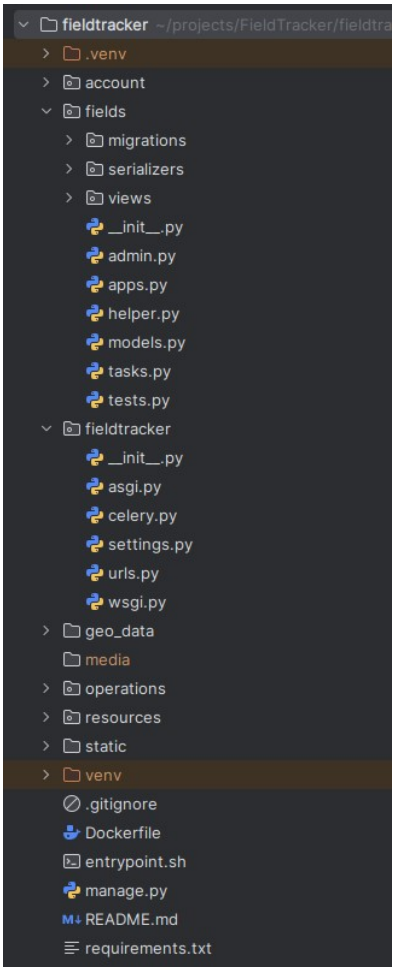


3.1 Ábra. Architektúrális terv

4. Webalkalmazás megvalósítása

4.1. Backend

A backend a Django és Django Rest Framework projekt kezdetekor elérhető legfrissebb verzióira épül. A projekt Django 5.0.4, Django Rest Framework 3.14.0 és Python 3.12 verziókkal lett megvalósítva. Django alkalmazások különálló alkalmazásokból vagy úgynevezett app-okból állnak. Az app-ok strukturálása alapvető fontosságú a kód áttekinthetősége, karbantarthatósága és moduláris felépítése szempontjából. Django appoknak köszönhetően a projekten belüli funkciókat különálló, logikailag egységes komponensekre tudjuk bontani, amelyek könnyebben kezelhetőek és újrahasznosíthatóak. A backendet négy fő alkalmazásra osztottam fel, amelyek mindegyike saját felelősségi körrel rendelkezik, így biztosítva a projekt átláthatóságát: fields (a földek és koordináták, ezekkel való számolások), operations (mezőgazdasági események, például aratás), resources (készletek, például vetőmag készletek), account (felhasználókezelés). Minden Django projektben található egy fő alkalmazás is, ami általában a projekt nevét viseli. Ez az alapalkalmazás tartalmazza a projekt beállításait és URL konfigurációit. Az így keletkezett mappa struktúrát a 4.1 Ábra mutatja.



4.1 Ábra. A webalkalmazás mappastruktúrája

A beállításokat a settings.py fájl határozza meg, amely magába foglalja többek között az alkalmazások regisztrálását (INSTALLED_APPS szekció), middleware-eket, az adatbázis beállításokat, a statikus fájlok kezelését, és az egyéb külső szolgáltatások integrációját. Az urls.py fájlban határozzuk meg a projekt fő URL struktúráját, ide kerülnek a gyökér végpontok, és innen lehetőségünk van az egyes appok urls.py fájljait is becsatolni. Ez a megközelítés biztosítja, hogy minden alkalmazás saját URL-konfigurációval rendelkezzen, lehetővé téve a végpontok logikus és jól szervezett felépítését.

4.1.1 Adatrögzítés

Minden alkalmazás rendelkezik modellekkel, kontrollerekkel és serializerekkel. Ezek az elemek segítenek az adatbázisban tárolt adatok kezelésében, a HTTP kérések kiszolgálásában, és az adatok JSON formátumba alakításában, amit az API végpontokon keresztül továbbítunk. A modellek a Django alkalmazás adatbázisának szerkezetét határozzák meg. Az egyes appok models.py fájljában találhatók, és definiálják az adatbázisban tárolt táblák felépítését. Egy modell egy osztály, amely az adatokat és azok tulajdonságait írja le. Django-

ban a modellek a django.db.models.Model osztályból öröklődnek, ami biztosítja, hogy a Django automatikusan létrehozza a megfelelő adatbázis táblákat a definiált mezőkkel. Egy ilyen osztályt mutat be a 4.2 Ábra.

```
class Point(models.Model):  13 usages  b3n3c
    name = models.CharField(max_length=150)
    longitude = models.FloatField()
    latitude = models.FloatField()
    altitude = models.FloatField(null=True, blank=True)
    order = models.SmallIntegerField(null=True, blank=True)
    updated_at = models.DateTimeField(auto_now=True)
    field = models.ForeignKey(
        Field, on_delete=models.SET_NULL, related_name="field_points",
        null=True, blank=True
    )
```

4.2 Ábra. Példa egy Model osztályra

A Django ORM (Objektum-relációs leképezés) segítségével a modellek objektumait könnyen kezelhetjük, anélkül, hogy SQL lekérdezéseket írnánk. Az ORM lehetővé teszi az adatok Python objektumként való kezelését, de a háttérben a Django SQL parancsokat futtat az adatbázison. Az ORM használatával a fejlesztők egyszerűen dolgozhatnak az adatokkal, elkerülve a manuális SQL kódolást. A projektben a különböző alkalmazások modell osztályai a következő adatokat írják le: földek, pontok (koordináták), vetések, aratások, talajmunkák, talajmunka típusok, trágyázások, permetezések, növények, vetőmagok, trágyák, permetek, mezőgazdasági gépek. Ahhoz, hogy a modellekből adatbázis táblák legyenek először adatbázis migrációkat kell létrehozni, majd ezeket futtatni. Ezeket a migrációkat Django automatikusan létrehozza a modell osztályok alapján *makemigrations* parancs terminálból való kiadása után, majd a *migrate* paranccsal hajtjuk végre őket.

A kontroller feladata a logika kezelése és az API végpontok kiszolgálása. Django-ban a view-k felelnek ezért a szerepért. A Django REST keretrendszerben (DRF) a view-k különböző osztályokból öröklődhetnek, például APIView-ből vagy egyszerű CRUD (Create, Read, Update Delete) műveletek esetén ModelViewSet-ből, melyek mindegyike különböző szintű testreszabhatóságot biztosít. Az APIView alapvető vezérlést nyújt, ahol külön definiálható a get, post, put, delete függvények viselkedése, míg a ModelViewSet olyan előre definiált CRUD műveleteket biztosít, amelyeket a List, Retrieve, Create, Update és Destroy metódusokkal könnyedén el lehet érni. Egy ilyen egyszerű viewset osztályt mutat be a 4.3 Ábra. A view-k vagy viewset-ek az API végpontokon érkező kérések logikáját határozzák meg, és a megfelelő válaszokat küldik vissza a kliensnek.

```
class SeedViewSet(viewsets.ModelViewSet):  2 usages  b3n3c
    permission_classes = [IsAuthenticated]
    queryset = Seed.objects.all()
    serializer_class = SeedSerializer
```

4.3 Ábra. ModelViewSet példa

Az API végpontok URL-jeit a fő alkalmazásban található urls.py fájlban határozzuk meg, ahol a *path()* vagy *re_path()* függvényekkel adunk meg útvonalakat, és a router használatával a viewset-eket gyorsan és egyszerűen bejegyezhetjük az URL-konfigurációba. A válaszok JSON formára alakításához, illetve a JSON kérések feldolgozásához a kontrollerek serializereket használnak.

A serializer-ek a modellekből származó Python objektumokat alakítják át JSON formátumú adathalmazokká és fordítva, így a szerver és a kliens között könnyen kezelhető adatcsere jöhet létre. A serializer-ek általában a serializers.ModelSerializer osztályból öröklődnek, amely egyszerűsíti a modellek és azok mezőinek JSON reprezentációját, lehetővé téve számukra az adatfeldolgozást a bejövő és kimenő HTTP kérések során. Egy serializer példát mutat be a 4.4 Ábra. Az ilyen architektúra biztosítja, hogy a Django REST Framework képes legyen a háttérrendszerből érkező adatok validálására és kezelésére, ezáltal hatékonyan támogatva a biztonságos és jól strukturált API-k fejlesztését.

```
class SeedSerializer(serializers.ModelSerializer):  5 usages  b3n3c
    plant = serializers.PrimaryKeyRelatedField(queryset=Plant.objects.all(),
                                               allow_null=True)
    class Meta:  b3n3c
        model = Seed
        fields = '__all__'

    def to_representation(self, instance):  b3n3c
        self.fields['plant'] = PlantSerializer(read_only=True)
        return super(SeedSerializer, self).to_representation(instance)
```

4.4 Ábra. ModelSerializer példa

4.1.2 Felhasználókezelés

A projekt a Django által nyújtott felhasználókezelést alkalmazza, de kiterjesztett modellekkel és extra biztonsági funkciókkal, például meghívásos regisztrációval, jelszó-visszaállítással és JWT (JSON Web Token) alapú autentikációval lett kibővítve. A JWT tokent használó rendszerekben a felhasználók egy egyszeri tokennel azonosítják magukat, amely időkorlátos, és lehetőséget nyújt az access és refresh tokenek kezelésére. A JWT tokent egyedi

és titkosított formában állítjuk elő, ami jelentősen megnehezíti a visszaéléseket. Az access tokenek időkorlátosak, így, ha valaki megszerez egy tokent, az csak rövid ideig érvényes. A frontendről érkező minden kéréshez csatolni kell a tokent, így a JWT autentikáció egyaránt kompatibilis a webes és mobil alkalmazásokkal. A projekt a tokenek kezeléséhez a Simple JWT plugint használja. A projektben a felhasználókat az Account modell kezeli, amely egy Django osztályból, az AbstractUser-ből származik, így támogatja a felhasználónév és e-mail alapú azonosítást is. A felhasználókat meghívások útján lehet regisztrálni: a saját megvalósítású RegisterView biztosítja, hogy a felhasználók csak érvényes meghívó token birtokában hozhatnak létre fiókot. Új felhasználóknak e-mail cím alapján generálható meghívó. A meghívó tokenek adatbázisban tárolódnak, egy e-mail címhez csak egy token tartozhat és ezek egy napig érvényesek. Miután a token lejár lehetőseg van újat generálni. A meghívókat a backend e-mailben küldi el a meghívott e-mail címekre. Ha egy felhasználó elfelejti a jelszavát, a rendszer lehetővé teszi a jelszó visszaállítását, melyhez egy egyórás érvényességű PasswordResetToken-t használunk. Ez a token is e-mailben kerül eljuttatásra a felhasználóhoz. A felhasználói meghívások és jelszó visszaállítási folyamatok mindegyike egyedi tokeneket használ. Ezeket UUID-ként generálja a rendszer, amelyeket nehéz kitalálni, így védettek az esetleges visszaélésektől.

4.1.3 Koordinátákkal való számolások

A koordinátákkal történő pontos számítások kivitelezéshez térképvetületekkel és koordináta rendszerekkel kapcsolatos háttértudás szükséges. A programozói munka megkönnyítése érdekében létrehoztam egy Python könyvtárat *geo-measurements* néven [19], amit nyílt forráskódúvá tettem és egy GitHub repo-ból elérhető [20]. A könyvtár minden függvényt tartalmaz a már említett funkciók eléréshez, azaz a távolság, terület, osztópont, illetve szög számolásokhoz. Ezeken felül segédfüggvényeket is tartalmaz ország és kontinens megállapításra koordináták alapján, offline módon. Minden funkciónak megvan a saját függvénye, ami inputként megfelelő mennyiségű pontot vár. A könyvtár felhasználja PyProj, utm, GeoPandas és geographiclib Python könyvtárakat.

A GeoMeasurements könyvtárban számolás első lépése a legnagyobb pontosságot elérő CRS (Coordinate reference system) automatikus meghatározása az input pontok alapján. A CRS egy koordináta rendszert definiál. A Föld, gömbölyű alakja miatt, nem vetíthető síkba az alakzatok (területek, távolságok, szögek) torzítása nélkül. Ezért léteznek különböző térképvetületek, amik más-más karakterisztikával rendelkeznek, bizonyos területeket arányaiban máshogy jelenítenek meg. Egy CRS azt írja le koordináták segítségével, hogy egy

2 dimenziós, sík térkép vetületen bizonyos pontok hol helyezkednek el a Földön. A CRS-ek lehetnek úgynevezett GCS (Geographic Coordinate Systems), azaz földrajzi (geodéziai) koordináta rendszerek vagy PCS (Projected Coordinate System), amik karteziánus koordináta rendszerek. A GCS rendszerek szélesség, hosszúság és magasság párokkal dolgoznak, még a PCS rendszerek egyszerű x,y koordinátákkal, azaz úgynevezett easting (K-NY) és northing (É-D) párokkal. A földmérésre használt eszközök általában a PCS rendszereket használják, mert ezek megőrzik a távolságot. Az ilyen rendszerek azonban nagy területre már kevésbé pontosak. A legnépszerűbb ilyen CRS az UTM (Universal Transverse Merctor), ami a földet 60 egyenlő zónára osztja fel, ezzel nagyfokú pontosságot biztosítva a zónákon belül. Léteznek országspecifikus CRS-ek is, amik általában az adott országban a legnagyobb pontosságot adják és általában az országon belül a hivatalos szervek által használt koordináta rendszereként szolgálnak. Ezek a CRS-ek csak egy adott ország határain belül értelmezhetőek, és általában bizonyos UTM zónákat pontosítanak. A GeoMeasurements könyvtár függvényeinek megadhatók paraméterben ilyen nemzeti CRS rendszerek, ebben az esetben ezeket fogja megpróbálni használni.

Különböző CRS-ek használatával tehát, ugyanarra az inputra más-más eredményeket kaphatunk koordinátákkal való számoláskor, ezért fontos, hogy mindig az inputhoz mérten a legpontosabbat válassza ki a rendszer. CRS meghatározásakor a GeoMeasurements könyvtár függvényei legelső lépésben ellenőrzik, hogy az input pontok egy ország határain belülre esnek-e, és tudja-e kezelni az ország specifikus koordináta rendszert (kapott-e ilyet paraméterben), mert ez adja a legnagyobb pontosságot. Abban az esetben, ha nem egy ország határain belül vannak pontok, vagy olyan országhoz tartoznak, amire a rendszer nincs felkészítve, a rendszer kiválasztja azt az UTM zónát, amibe az input pontok tartoznak és abban fogja végrehajtani az számolásokat. Az az eset is előfordulhat azonban, hogy az input pontok különböző UTM zónákba esnek. Ilyen esetben, ha az összes pont egy kontinensen belül található, akkor a kontinens specifikus CRS-t fogja használni. A könyvtár az összes kontinensre fel van készítve, de az alapértelmezett CRS-ek felül is írhatóak kontinensenként. Azokban a szélsőséges esetekben, amikor a pontok különböző kontinenseken találhatók, a rendszer WGS84-et fog használni. Ilyen eset előállása precíziós mezőgazdasági felhasználás esetén nagyon ritka, de nemzeti CRS megadásával lehetetlen. A WGS84 (The World Geodetic System 1984) egy GCS, azaz szélesség és hosszúság párokkal dolgozik, a fentebb említett rendszerek közül ez adja a legpontatlanabb számítási eredményeket. A WGS84 azonban az egyik legnépszerűbb koordináta rendszer, mert a Föld egész területére értelmezhető és a térképek (például a Google

Maps is) előszeretettel használják. Ezért a számítások input koordinátái is ebben a formátumban lesznek.

A CRS meghatározása után az input koordinátákat át kell váltani az input WGS84 rendszerből a meghatározott legoptimálisabb rendszerre. Ehhez a művelethez nemzeti vagy kontinentális CRS esetén a *pyproj* könyvtár Tranformer osztályát használja a rendszer, egyszerű UTM konverzió esetén pedig az *utm* Python könyvtárat. Projected Coordiante Reference System használata esetén a számolások elvégzése egyszerű x,y koordináta rendszerben történik, tehát egyszerű koordináta geometriai képletek használatával kapjuk meg a megoldásokat. Két pont közötti távolságot Euklideszi távolság kiszámításával lehet megkapni, poligon területét cipőfűző formulával vagy másnéven Gauss területtel számoljuk, két pont felezőpontját szakasz osztópont képlettel kapjuk meg, két pont által bezárt északhoz viszonyított szöget pedig atan2 függvény segítségével kapjuk meg. Abban az esetben, ha nem sikerült optimális CRS-t meghatározni és WGS84 rendszerben kell számításokat végezni, a képletek már nem ilyen egyszerűek. WGS84 koordináták ugyanis nem karteziánus koordináták, azaz nem értelmezhetők egy egyszerű x,y koordináta rendszerben. A WGS84 rendszer a Földet forgási ellipszoidként modellezi, ahol a szélesség a Föld egy adott pontjától az Egyenlítőig mért szög, a hosszúság pedig a greenwichi délkörhöz (nulladik hosszúsági kör) viszonyított szög. Ez azt jelenti, hogy ezekkel a koordinátákkal történő számítások képletei meglehetősen bonyolultak és sok trigonometrikus számításból állnak. A számítások implementálásnak megkönnyítése érdekében több Python könyvtárat vettem igénybe, ezek a geographiclib, shapely és pyproj könyvtárak.

A GeoMeasurements könyvtár pontosságát unit tesztek mellett valós mérésekkel is ellenőriztem. Terepen mérőszalaggal távolságokat mértem 4.5 Ábrán látható módon, majd a kijelölt pontok koordinátáit a projekt keretein belül megvalósított mobilalkalmazással határoztam meg.

4.5 Ábra. Adatgyűjtés terepen

8 távolságot ellenőriztem: 10cm, 20cm, 50cm, 1m, 2m, 3m, 4m, 5m. Mind a nyolc csoportot három alkalommal mértem különböző pontokkal. Nemzeti CRS használatával a könyvtár *calculate_distance* függvénye átlagosan 4,18 centimétert tévedett. Az eredményeket a 4.6 Ábra szemlélteti.

Távolság (cm)	Koordináta 1 (szélesség, hosszúság)	Koordináta 2 (szélesség, hosszúság)	Számolt távolság (cm)	Eltérés abszolút értéke (cm)	Átlagos eltérés (cm)
10	45.80694309166667, 20.08992326166667	45.806944376666664, 20.08992356333333	14.46877706	4.468777064	3.615986313
	45.802712289999995, 20.08952961166667	45.80271095166667, 20.089528936666667	15.7682801	5.768280096	
	45.799175671666674, 20.089187325000005	45.79917482666667, 20.089187335000005	9.38909822	0.61090178	
20	45.80694309166667, 20.08992326166667	45.806946106666665, 20.08992382	33.77942026	13.77942026	6.766341316
	45.802712289999995, 20.08952961166667	45.80271004, 20.08952929	25.12437865	5.124378651	
	45.799175671666674, 20.089187325000005	45.799173746666675, 20.089187256666666	21.39522504	1.395225037	
50	45.80694309166667, 20.08992326166667	45.806948025000004, 20.08992345	54.83376005	4.833760048	3.263563617
	45.802712289999995, 20.08952961166667	45.80270749333334, 20.089529121666665	53.43154876	3.431548757	
	45.799175671666674, 20.089187325000005	45.79917106, 20.089186628333334	51.52538205	1.525382047	
100	45.802712289999995, 20.08952961166667	45.80270312, 20.089528554999998	102.2180933	2.218093259	1.263799855
	45.799175671666674, 20.089187325000005	45.799166613333334, 20.08918659	100.8089101	0.808910123	
	45.799175671666674, 20.089187325000005	45.79918472833333, 20.089187998333333	100.7643962	0.764396182	
200	45.80694309166667, 20.08992326166667	45.806961533333336, 20.089925958333332	205.9738946	5.973894592	4.297134125
	45.802712289999995, 20.08952961166667	45.80269393166667, 20.089527620000005	204.5655215	4.565521514	
	45.799175671666674, 20.089187325000005	45.79919344, 20.089188536666665	197.6480137	2.351986267	
300	45.80694309166667, 20.08992326166667	45.80697004666666, 20.0899264	300.4880021	0.488002133	3.904570948
	45.802712289999995, 20.08952961166667	45.80268454166667, 20.089527196666666	308.8819733	8.881973278	
	45.799175671666674, 20.089187325000005	45.799202423333334, 20.089189356666665	297.6562626	2.343737434	
400	45.80694309166667, 20.08992326166667	45.806979580000004, 20.08992738	406.6826528	6.682652828	5.477972446
	45.802712289999995, 20.08952961166667	45.80267608166667, 20.089526283333335	403.1408226	3.140822605	
	45.799175671666674, 20.089187325000005	45.799211045, 20.089189481666665	393.3895581	6.610441906	
500	45.80694309166667, 20.08992326166667	45.80698808499999, 20.08992841166667	501.5197507	1.519750662	4.848935121
	45.802712289999995, 20.08952961166667	45.8026668, 20.089525595	506.4016105	6.401610547	
	45.799175671666674, 20.089187325000005	45.79922002333333, 20.08919041333333	493.3745558	6.625444153	
					4.179787968

4.6 Ábra. Mérési eredmények nemzeti CRS használatával

A mérés közben felmerülő hibák több forrásból származhatnak. Pár centiméternyi hiba keletkezhetett a koordináták meghatározásakor is. A mérőszalaggal kimért pontokat és a földmérő eszközt nehéz összeegyeztetni. Magának a földmérő eszköznek is van egy átmérője,

ami megnehezíti a centiméter pontos mérést, valamint a mérőeszköz enyhe megdöntése is hibás eredményhez vezethet. Ezen felül az RTK módszer is megenged körülbelül 1 centiméternyi hibát. A távolság számolást elvégeztem nemzeti CRS használata nélkül is, azaz egyszerű UTM zóna használatával. Ezek eredményeit a 4.7 Ábra mutatja.

Távolság (cm)	Koordináta 1 (szélesség, hosszúság)	Koordináta 2 (szélesség, hosszúság)	Számolt távolság (cm)	Eltérés abszolút értéke (cm)	Átlagos eltérés (cm)
10	45.80694309166667, 20.08992326166667	45.806944376666664, 20.08992356333333	14.47	4.468776911	3.615986267
	45.802712289999995, 20.08952961166667	45.80271095166667, 20.089528936666667	15.77	5.768280297	
	45.799175671666674, 20.089187325000005	45.79917482666667, 20.089187335000005	9.39	0.610901594	
20	45.80694309166667, 20.08992326166667	45.806946106666665, 20.08992382	33.78	13.77942028	6.766341352
	45.802712289999995, 20.08952961166667	45.80271004, 20.08952929	25.12	5.124378744	
	45.799175671666674, 20.089187325000005	45.799173746666675, 20.089187256666666	21.40	1.395225034	
50	45.80694309166667, 20.08992326166667	45.806948025000004, 20.08992345	54.83	4.833759957	3.263563582
	45.802712289999995, 20.08952961166667	45.80270749333334, 20.089529121666665	53.43	3.431548751	
	45.799175671666674, 20.089187325000005	45.79917106, 20.089186628333334	51.53	1.525382038	
100	45.802712289999995, 20.08952961166667	45.80270312, 20.089528554999998	102.22	2.218093354	1.278637859
	45.799175671666674, 20.089187325000005	45.799166613333334, 20.08918659	100.81	0.808910111	
	45.799175671666674, 20.089187325000005	45.79918472833333, 20.089187998333333	100.76	0.808910111	
200	45.80694309166667, 20.08992326166667	45.806961533333336, 20.089925958333332	205.97	5.9738946	4.297134126
	45.802712289999995, 20.08952961166667	45.80269393166667, 20.089527620000005	204.57	4.565521613	
	45.799175671666674, 20.089187325000005	45.79919344, 20.089188536666665	197.65	2.351986165	
300	45.80694309166667, 20.08992326166667	45.80697004666666, 20.0899264	300.49	0.488002133	3.904570946
	45.802712289999995, 20.08952961166667	45.80268454166667, 20.089527196666666	308.88	8.881973371	
	45.799175671666674, 20.089187325000005	45.799202423333334, 20.089189356666665	297.66	2.343737333	
400	45.80694309166667, 20.08992326166667	45.806979580000004, 20.08992738	406.68	6.682652828	5.477972443
	45.802712289999995, 20.08952961166667	45.80267608166667, 20.089526283333335	403.14	3.140822694	
	45.799175671666674, 20.089187325000005	45.799211045, 20.089189481666665	393.39	6.610441806	
500	45.80694309166667, 20.08992326166667	45.80698808499999, 20.08992841166667	501.52	1.519750662	4.848935181
	45.802712289999995, 20.08952961166667	45.8026668, 20.089525595	506.40	6.401610741	
	45.799175671666674, 20.089187325000005	45.79922002333333, 20.08919041333333	493.37	6.625444141	
					4.18164272

4.7 Ábra, mérési eredmények nemzeti CRS használata nélkül

A minimális eltérést a két eredmény között a pontok lokációja magyarázza. Ezek a koordináták Szerbiában találhatók. Szerbia teljes területe egy UTM zónába tartozik (Magyarországgal ellentétben), ezért a nemzeti CRS (EPSG:8682), különösen olyan sík területen, ahol ezek pontok találhatók, nem ad számottevő korrekciót az egyszerű UTM zónához viszonyítva.

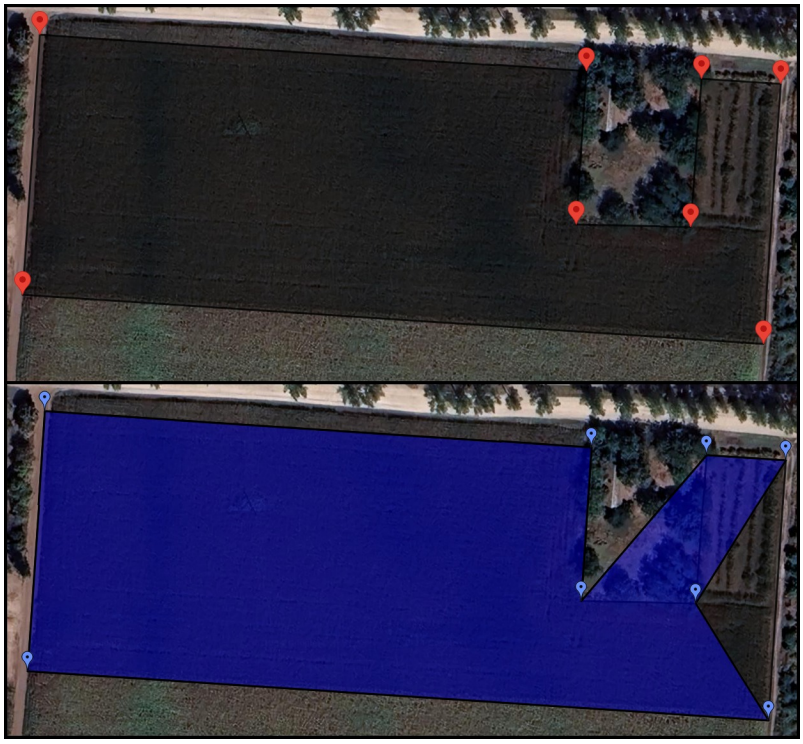
4.1.4 Földek és pontok importálása

Lehetőség van földeket és pontokat csv fájlból importálni, ezzel megkönnyítve új felhasználóknak a rendszer adatokkal való feltöltését. Logikailag (és adatbázis szinten is) a pontok vagy különállóak vagy földekhez tartoznak. Ezért célszerű előbb földekkel feltölteni a rendszert, majd a hozzájuk tartozó pontokkal. Ezt két különböző csv használatával lehet megtenni, az egyik a földeket tartalmazza, a másik a pontokat.

Föld objektumok létrehozása a backenden egyszerűen történik, az importhoz tartozó view a csv ellenőrzése után a földeknek megfelelő modell osztályokból álló listát hoz létre majd ezeket a rekordokat a *model.objects.bulk_create* metódussal létrehozza az adatbázisban. Pontok létrehozásakor, pontosabban pontok földekhez való csatolásakor, már több tényezőt kell figyelembe venni.

Ha a pont egy föld objektumhoz tartozik, és az érintett föld objektum létrehozásakor a felhasználó nem adott meg területet, akkor a rendszer automatikusan meg fogja határozni a föld területét a hozzárendelt pontok alapján. Egy föld területe, amennyibben nincs felhasználó által megadott érték, mindig újraszámolódik, amikor a hozzárendelt pontok változnak. Ez a kiszámolt területet adatbázis szinten is tárolódni fog. Ahhoz azonban, hogy egy poligon területét meghatározzuk a 4.1.3-as fejezetben taglalt módon, fontos meghatározni a földhöz tartozó koordináták sorrendjét. Ez a sorrend írja le, hogy a koordinátákat milyen sorrendben kell összekötni ahhoz, hogy a kívánt poligont megkapjuk. Ha a koordinátákat rossz sorrendben kötjük össze, akkor egy a termőföld valódi alakjától teljesen különböző alakzatot kapunk és így a terület számítása is rossz lesz, mert ennek a hibás alakzatnak fogjuk megkapni a területét. A felhasználótól nem várható el, hogy a csv fájlban a pontokat a helyes sorba rendezze, ezért ezt a számítást a backend-nek kell elvégeznie. Ennek számításnak a későbbiekben is hasznát veszi a rendszer, amikor a pontok egyesével kerülnek hozzáadásra/törlésre a földhöz, valamint a földek frontend-en való megjelenítésekor is fontos szerepet fog játszani a kiszámolt sorrend.

A GeoMeasurements könyvtár *order_points_for_polygon* függvénye képes elvégezni ezt a feladatot. A mezőgazdasági termőterületek bármilyen sokszög alakzatot felvehetnek. Az alakzatok lehetnek szabálytalan, konvex, de akár konkáv sokszögek is. Ahhoz, hogy egy sokszöget pontosan meg tudjunk rajzolni a csúcsokon kívül ismernünk kell az éleket vagy a szögeket is [21]. Ezen információk nélkül nem adható olyan algoritmus, ami mindig a helyes megoldást fogja adni egy input koordináta listára. Azonos koordináták különböző összekötését mutatja be a 4.8 Ábra. Olyan algoritmus azonban adható, ami az egyszerűbb alakzatokra helyes megoldást ad. A mezőgazdasági területek bár sokféle alakzatot felvehetnek, legtöbb esetben téglalap, vagy ahhoz hasonló alakúak [22]. Az *order_points_for_polygon* függvény egy ilyen algoritmust használ.



4.8 Ábra. Probléma a sokszög éleinek meghatározásakor

A függvény az input rendezetlen koordináta listát már PCS rendszerben várja, hogy x,y koordináta rendszerben dolgozzon. Első lépésben meghatározza a pontok által behatárolt terület közepét átlag számítást használva. Az input pontok átlagos x,y koordinátái megadják középpontot. Ez után minden pontnak meghatározza a középponthoz mért szögét. Ha ez a szög szerint rendezzük a pontokat, akkor egy olyan sorrendet kapunk, amivel a legtöbb termőföld helyesen leírható.

Létezhetnek olyan bonyolult alakzatú földek, amikre a fenti algoritmus nem ad megfelelő megoldást. Ezekben a szélsőséges esetekben a frontend fog segítséget nyújtani a felhasználónak, aki manuálisan felülírhatja a hibás sorrendet. Importáláskor figyelembe kell venni, hogy ezek a számítások több száz rekord esetén időigényesek lehetnek. Ha fájl feldolgozása túl sokáig tart, a frontendről érkező kérés timeout-olhat. Ezért ezt a folyamatot optimalizálni kell és workerek-be szervezni az aszinkron feldolgozáshoz. Ezt a feladatot a 3.1 fejezetben taglalt módon Celery workerek segítségével oldottam meg.

A Celery worker egy háttérben futó folyamat, amely várakozik és feldolgozza a Celery-hez küldött feladatokat. A workerek képesek párhuzamosan több feladatot is végrehajtani, ami jelentősen növeli a teljesítményt, különösen időigényes műveleteknél. Ezekben a workerekben pedig batch processing-et használok. A worker először feldogozza a csv-ben található pontokat, amiket 200-as méretű batch-ekben ment el. A pontokhoz meghatározza azokat a földeket, amiknek a területét frissíteni kell. A földeknek 20-as méretű batch-ekben fogja kiszámolni az

új területét és újra rendezni a pontjait. A workerekhez az üzenet a Django-tól Redis-en keresztül jut el. Redis egy óráig tárolja a workerek azonosítóját és státuszát is. Ahhoz, hogy a frontend értesülhessen a fájl feldolgozásának eredményéről a backend biztosít egy endpoint-ot, ahol azonosító alapján lekérdezhető a worker státusza.

4.2. Frontend

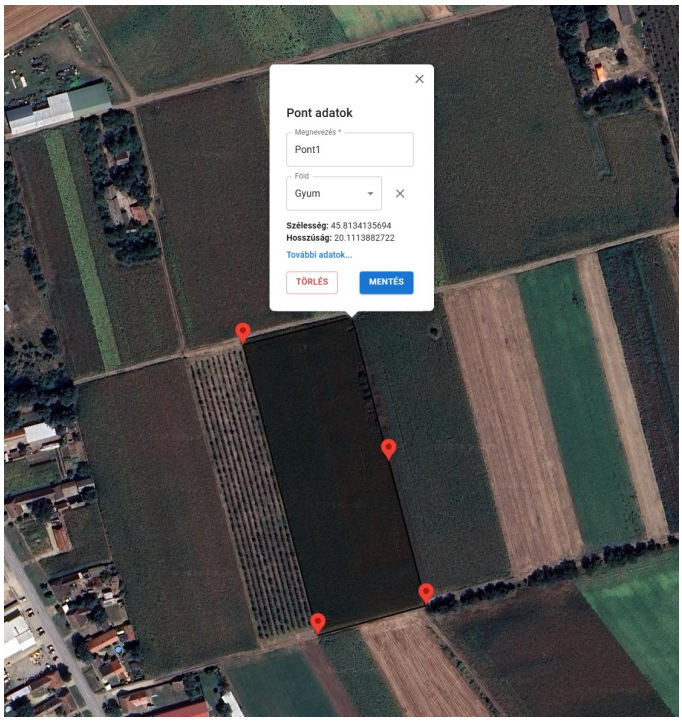
A projekt frontendje React.js és Vite kombinációjával készült, ahol a felhasználói felület kialakításában a Material UI elemek dominálnak. A Vite választása jelentős előnyökkel jár a projekt szempontjából. A Vite egy modern, gyors build-eszköz, amely azonnali szerverindítást biztosít, és olyan fejlesztési élményt nyújt, amely különösen nagyobb projektek esetén érezhetően hatékonyabbá teszi a munkát. A Vite előnyei közé tartozik az azonnali hot-reload, amely gyorsítja a fejlesztést, és a modul-alapú build rendszer, amely lehetővé teszi, hogy csak az éppen szükséges részek kerüljenek betöltésre. Ezáltal a Vite nemcsak gyorsabb fejlesztést, de kisebb végleges csomagméretet is biztosít, ami javítja az alkalmazás teljesítményét.

A projekt az API-hívásokhoz Axios-t használ, amelyhez egy JWT-alapú autentikációs interceptor került hozzáadásra. Az useAxiosWithJwtInterceptor egyedi hook, amely lehetővé teszi az Axios konfigurációját, hogy minden kérés automatikusan tartalmazza az aktuális hozzáférési tokent. Az autentikációs token a localStorage-ban van tárolva, és az interceptor az összes kérést ezzel a tokennel látja el a "Bearer" típusú autentikációs fejléccel. Az interceptor emellett figyeli az API-válaszokat is. Ha egy kérés során a szerver 401-es státuszkódot küld vissza (ami az érvénytelen vagy lejárt tokenre utal), akkor automatikusan megkísérli a hozzáférési token frissítését a háttérrben. A token/refresh végponton keresztül kér egy új hozzáférési (access) tokent a frissítési (refresh) token alapján, majd elmenti az új tokent a localStorage-ba, és megpróbálja újraküldeni az eredeti kérést az új tokennel. Ha a token frissítése sikertelen, az alkalmazás automatikusan a bejelentkezési oldalra irányítja a felhasználót, ezzel jelezve, hogy az autentikációra újra szükség van.

Minden backenden definiált modellhez tartozik egy-egy űrlap (form), amely lehetővé teszi az adatok felvitelét és módosítását. Az adatok listázásához a Material UI DataGrid komponensét használjuk, amely hatékony és rugalmas adatkezelést biztosít, különösen nagyobb mennyiségű adat megjelenítésekor. A fejlesztés során külön figyelmet fordítottam arra, hogy a komponensek újrahasznosíthatóak legyenek, így nemcsak az alkalmazás kinézete vált egységessé, hanem a kódolás folyamata is letisztultabbá és hatékonyabbá vált. Az egységesen megtervezett komponensek biztosítják, hogy a formok és a datatable-ek konzisztens

megjelenésűek és könnyen használhatóak legyenek, minimalizálva a duplikált kódot és egyszerűsítve a karbantartást.

A frontend másik fontos funkciója az adatrögzítésen kívül a földek és pontok térképen való megjelentítése. A térképhez a GoogleMaps-et használ műholdas nézetben. A térképeket API-kon keresztül lehet implementálni alkalmazásokban és ezek az apik sok esetben fizetős szolgáltatások. GoogleMaps alternatívája lehetne a MapBox API. Műholdas térképek megjelenítésekot azonban egyik API sem költségmentes. Mind a két API rendelkezik egy ingyenes sávval, amin belül bizonyos számú kérést ingyenesen kiszolgál. Ez a sáv egy kis felhasználóbázis kiszolgálására alkalmas, és fejlesztés alatt sem kell félni az ingyenes sáv túllépést illetően. Én a GoogleMaps API szolgáltatásait választottam a jó dokumentációja és széleskörű támogatottsága miatt. A térkép megjelenítését a GoogleMap komponens végzi a @react-google-maps/api npm csomagból. A frontend lekérdezi az adatbázisban tárolt földeket, pontokat, és a legutoljára végzett mezőgazdasági műveleteket a földeken. A földeket a térképen poligon elemekként jeleníti meg. A poligon definiálásakor fontos a 4.1.4-es fejezetben kifejtett sorrend meghatározása, ugyanis az npm csomag Polygon komponense ez a sorrend szerint rajzolja meg a földet. Földet a bele vetett takarmánynövény szerint színezi is, abban az esetben, ha van benne elvetve ilyen, egyéb esetekben fekete színre színezi. Az egyes növényekhez tartozó színt form elemen keresztül tudja a felhasználó felvenni, hasonlóképpen a vetés eseményét, és az aratást is. Magukat a pontokat is lehetőség van megjeleníteni Marker komponensekkel. Ha a felhasználó bal klikkel rákattint egy pontra vagy földre, egy DialogBox komponensben megjelennek a releváns információk (például név, terület, szélesség, hosszúság stb.) és a releváns műveletek (például törlés, módosítás, pont földhöz rendelése stb.). Ezt a működést mutatja be a 4.9 Ábra.

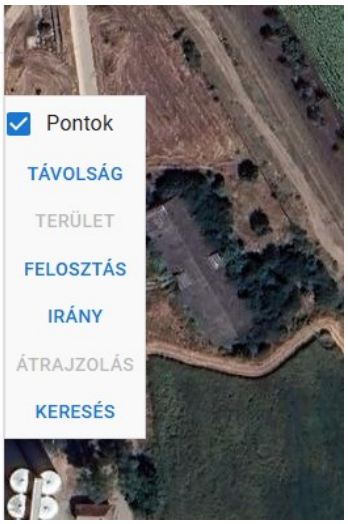


4.9 Ábra. Földek és pontok megjelenítése frontend-en

Ha a felhasználó bal klikkel a térképre kattint, új pontokat vehet fel. Ezeket a pontokat a saját DialogBox komponensükből tudja elmenteni, elmentés nélkül csak ideiglenesen lesznek a térképen. Az ideiglenes (még el nem mentett) pontok zöld színnel jelennek meg a térképen. A pontokat jobb klikkel lehet kijelölni számolások elvégzésére, ilyenkor a pontokhoz tartozó Marker kék színre vált, ezzel jelezve a kijelölést. A kijelölt pontok mennyisége alapján a térképen kirajzolódik a pontok által behatárolt alakzat. Ezt a funkcionalitást a 4.10 Ábra mutatja be.

4.10 Ábra. Pontok kijelölése számoláshoz

Ez lehet szakasz vagy poligon. Számolásokat a térkép komponens bal oldalán található eszközsávban található gombok segítségével végezhet a felhasználó. Ezek a gombok a backend által számolt műveletek: távolság, terület, szög és osztópont. Az eszközsávot a 4.11 Ábra szemlélteti.



4.11 Ábra. Eszközsáv

Egy ilyen gomb megnyomásakor a frontend egy POST kérést küld el a backendnek a kiválasztott pontokkal a számoláshoz megfelelő endpoint-ra, ami vagy egy szám típusú eredménnyel fog válaszolni, vagy osztópont számoláskor a kiszámolt koordinátákkal. Az eredményt a frontend egy dialgoboxban vagy az új koordináta felvételével jeleníti meg. Ugyanebben az eszközsávban van lehetőség rosszul behatárolt földek újra rajzolására is. Ezt egy föld kiválasztásával és az Újrarajzolás gomb lenyomásával teheti meg a felhasználó. Újrarajzoló módban csak a földhöz tartotó pontok jelennek meg, amiket jobb klikkel a helyes sorrendben történő kiválasztással a föld poligonja újrarajzolódik. Ilyenkor a frontend meghívja a megfelelő API endpointot az új sorrenddel, ami elmentődik az adatbázisban és az érintett föld területe újraszámolódik.

5. Mobilalkalmazás megvalósítása

5.1 Földmérés

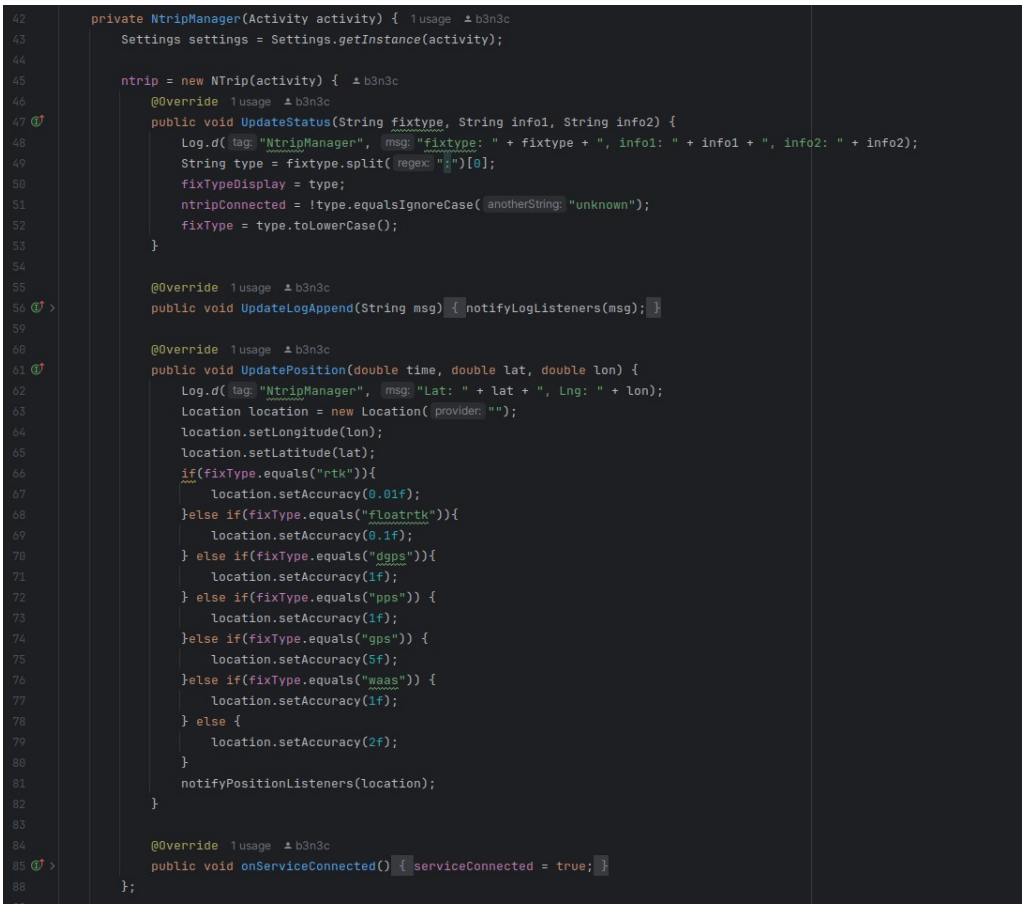
Az alkalmazás földmérő funkciója a Mérés fülről érhető el a MainActivity-n belül. Amikor a felhasználó a Mérés fülre navigál, a SurveyFragment Fragment osztály fog megjelenni. Ezen a képernyőn megjelenik egy GoogleMaps térkép műholdas nézetben. Az elmentett pontok Marker-ek formájában jelennek meg a térképen. A képernyő jobb alsó sarkában található egy iránytű, jobb felső sarkában pedig a műveleti gombok. Az iránytűt egy nyilvános GitHub repository felhasználásával implementáltam. Ez az iránytű a beállításokban

ki is kapcsolható. A képernyő tetején található Toolbar-ban az eszköz aktuális lokációja és annak pontossága kerül kiírásra.

5.1.1 NTRIP RTK korrekciók

Ahogy azt már 2.1.1 fejezetben részleteztem, a mobiltelefon saját GPS adatai nem elég pontosak földmérési adatok ellátásra. Az alkalmazásnak képesnek kell lennie RTCM üzenetek fogadására egy RTK szervertől NTRIP protokollon keresztül, ezeket a jeleket továbbítania Bluetooth segítségével egy vevő egységbe, ami visszaküldi az alkalmazásnak centiméter pontos lokációt. Azaz az alkalmazásnak NTRIP kliens szerepet kell betöltenie.

Ezen feladatkör ellátására egy nyilvános GitHub repository-t használtam alapul. Ez egy Pedro Nunes Institute által fejlesztett nyilvános MIT licenszű projekt, ami szintén egy másik nyílt forráskódú projektre épül [23]. Ez a projekt azonban Android fejlesztési szempontból réginek minősül (a szakdolgozat íráskor 11 éves), ami azt jelenti, hogy bizonyos felhasználói engedély kérési funkciókkal korszerűsíteni kellett. Az NTRIP kliens használatához létrehoztam egy NtripManager singleton osztályt. Ez az osztály felelős az NTRIP és Bluetooth kapcsolatok létrehozására, azok lezárására. A mögöttes logikát a Pedro Nunes Institute repo-jából átmásolt NTrip és NTRIPService osztályok végzik. Az NtripManager inicializálja ezeket az osztályokat ügyelve a megfelelő felhasználói engedélyek meglétére is. Bluetooth kapcsolatok és eszközök kezelésére a felhasználónak meg kell adnia bizonyos engedélyeket az alkalmazásnak. Ezeket az engedélyeket az AndroidManifest.xml fájlban is deklarálni kell. Ugyanitt deklarálnunk kell service-ként az NTRIPService osztályt, hogy az a háttérben dolgozhasson. Az NTrip osztály egy absztrakt osztály, ami elindítja az NTRIPService-t. Az NTrip osztályt az NtripManager valósítja meg. AZ NTrip osztály megvalósítását az 5.1 ábra mutatja be.

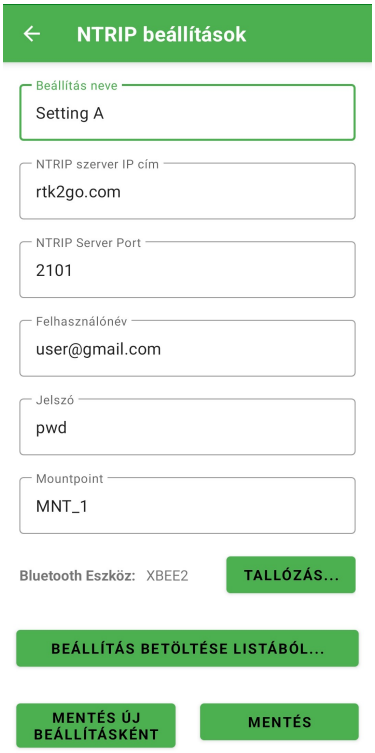


```
42      private NtripManager(Activity activity) {  1 usage  b3n3c
43          Settings settings = Settings.getInstance(activity);
44
45          ntrip = new NTrip(activity) {  b3n3c
46              @Override  1 usage  b3n3c
47              public void UpdateStatus(String fixtype, String info1, String info2) {
48                  Log.d( tag: "NtripManager",  msg: "fixtype: " + fixtype + ", info1: " + info1 + ", info2: " + info2);
49                  String type = fixtype.split( regex: " ")[0];
50                  fixTypeDisplay = type;
51                  ntripConnected = !type.equalsIgnoreCase( anotherString: "unknown");
52                  fixType = type.toLowerCase();
53              }
54
55              @Override  1 usage  b3n3c
56              public void UpdateLogAppend(String msg) { notifyLogListeners(msg); }
59
60              @Override  1 usage  b3n3c
61              public void UpdatePosition(double time, double lat, double lon) {
62                  Log.d( tag: "NtripManager",  msg: "Lat: " + lat + ", Lng: " + lon);
63                  Location location = new Location( provider: "");
64                  location.setLongitude(lon);
65                  location.setLatitude(lat);
66                  if(fixType.equals("rtk")){
67                      location.setAccuracy(0.01f);
68                  }else if(fixType.equals("floatrtk")){
69                      location.setAccuracy(0.1f);
70                  } else if(fixType.equals("dgps")){
71                      location.setAccuracy(1f);
72                  } else if(fixType.equals("pps")) {
73                      location.setAccuracy(1f);
74                  }else if(fixType.equals("gps")) {
75                      location.setAccuracy(5f);
76                  }else if(fixType.equals("waas")) {
77                      location.setAccuracy(1f);
78                  } else {
79                      location.setAccuracy(2f);
80                  }
81                  notifyPositionListeners(location);
82              }
83
84              @Override  1 usage  b3n3c
85              public void onServiceConnected() { serviceConnected = true; }
88          };
```

5.1 Ábra. NTrip osztály megvalósítása

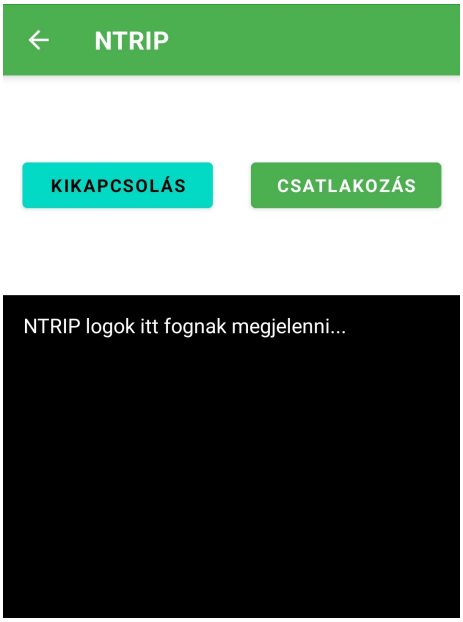
Az NTRIP státusz eseményekre, azaz új pontosított pozíció érkezésre, más osztályok az NtripManager-en Listener osztályokon keresztül iratkozhatnak fel. Az NtripManager a listener osztályokat egy listában tárolja és pozíció frissítéskor értesíti őket. Az NtripManager-ben kétféle Listener osztályt definiáltam, az egyik pozíciókra figyel, a másik log-okra.

NTRIP kapcsolatot és beállításokat az alkalmazás Eszközök fülén lehet létrehozni. Ezt a felületet mutatja be az 5.2 Ábra. NTRIP kapcsolat létrehozásához először létre kell hozni egy NTRIP beállítást. Egy beállítás több adatból áll: NTRIP szerver IP címe, NTRIP szerver port, felhasználónév, jelszó, mountpoint és egy bluetooth eszköz MAC címe. Ezen adatok mindegyike szükséges egy NTRIP kapcsolat létrehozásához. A Bluetooth eszköz MAC címének megadásában az alkalmazás segíti a felhasználót. A korábban párosított Bluetooth eszközök listájából ki lehet választani a kívánt eszközt, az alkalmazás automatikusan kezeli a MAC címet. Egy NTRIP beállításnak lehetőség van nevet is adni és ezzel el lehet menteni későbbi használatra. Lehetőség van korábban elmentett beállítások listából való betöltésére is. Ezeket a beállításokat (más beállításokkal együtt) a saját megvalósítású Settings singleton osztály kezeli, ami az alkalmazás saját memóriájába menti őket SharedPreferences használatával.



5.2 Ábra. NTRIP beállítások létrehozására szolgáló felület

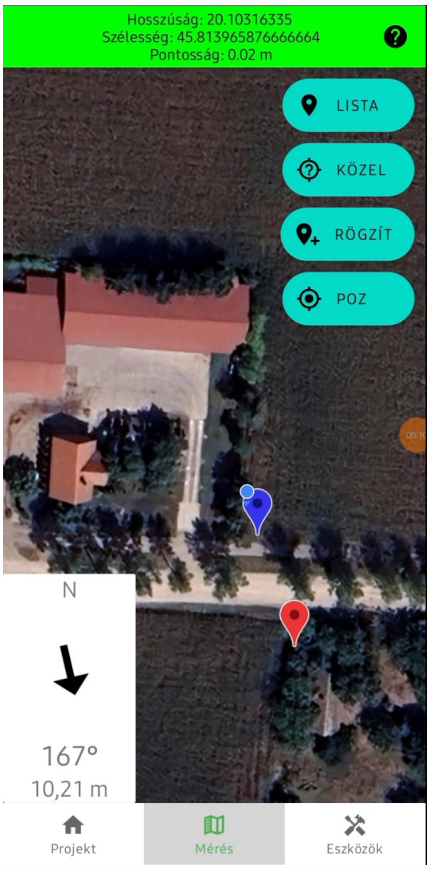
Az Android alkalmazásban a SharedPreferences osztály egy könnyű, kulcs-érték alapú adattárolási lehetőséget biztosít. Az adatok perzisztensen tárolódnak, így azok megmaradnak az alkalmazás újraindítása után is. NTRIP kapcsolat az Eszközök > NTRIP csatlakozás pontból inicializálható. Ezen a képernyőn található egy Csatlakozás és Kikapcsolás gomb, valamint egy LOG-ok megjelenítésére alkalmas felület. A képernyőt az 5.3 Ábra szemlélteti. A csatlakozás/kiakpcsolás gombok az NtripManager osztály connect és disconnect metódusait hívják meg. A logokat pedig az NtripManager log frissítéseire, NtripLogListener interface megvalósításával, feliratkozva figyelik.



5.3 Ábra. NTRIP csatlakozás az alkalmazásban

5.1.2 Pontok kitűzése

Pontok kitűzése a mérési folyamat kezdetét jelenti. Három módon lehetséges megtenni, mindhármat az alkalmazás Mérés fülén. Az egyik lehetséges mód a térképen egy pontra kattintás után előhívott DialogFragment-en a kitűzés gombra kattintás. A másik két mód a képernyő jobb felső részén található műveleti gombok használata. Lehetőség van pontot kitűzni listából, illetve a jelenlegi pozícióhoz mérten legközelebbi pont automatikus kitűzésére. Egy pont kitűzésekor a bal alsó sarokban megjelenik egy információs panel. Ezen az információs panelen látható a kitűzött pont és az eszköz aktuális helyzete közötti távolság, illetve, a két pontból húzott egyenes északhoz viszonyított szöge és ennek iránya. Ez mutatja, hogy az eszközt északra fordítva, a kitűzött pont milyen távolságra és irányba van a jelenlegi pozíciónkhoz. A képernyőt az 5.4 Ábra szemlélteti.

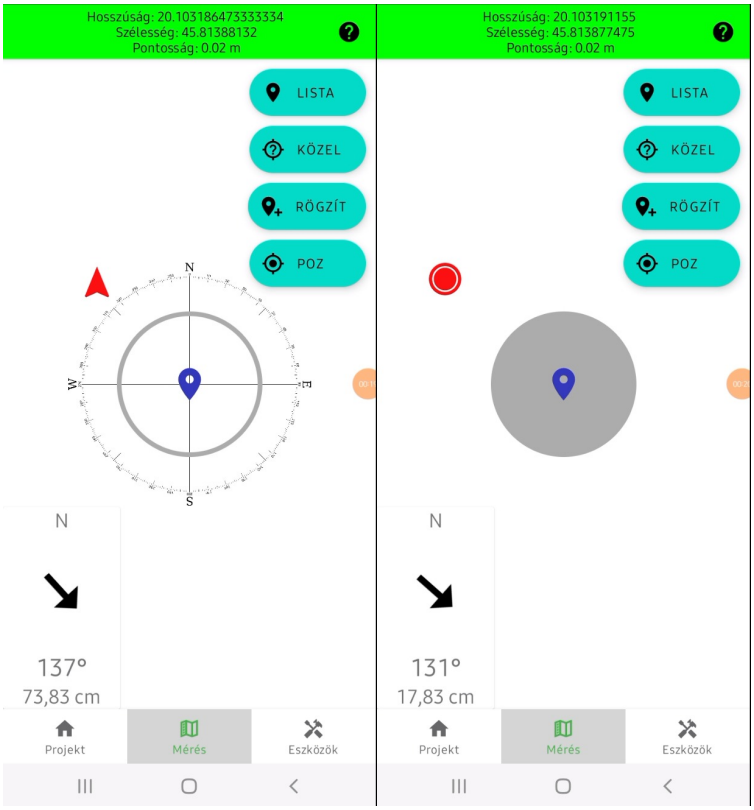


5.4 Ábra. Mérés képernyő kitűzött ponttal

Fontos megemlíteni, hogy földmérés közben nem történik semmilyen koordináta konverzió. Földméréskor az Android beépített Location osztályát és ennek *distanceTo*, illetve *bearingTo* metódusait használja az alkalmazás. A Location osztály az NtripManager listener-en keresztül kapott WGS84 koordinátákkal dolgozik. A korábbi fejezetekben már kifejtettem, hogy WGS84 koordináták nem alkalmasak pontos számításokra, mert a velük végzett műveletekor figyelembe kell venni a Föld gömbölyű alakját. A gömbölyű alakból származó hiba azonban kis távolságok mérésekor elenyészően kicsi, centiméteres távolságok meghatározásakor teljesen figyelmen kívül hagyható. Ez azt jelenti, hogy ha a kitűzött pont kilométerekre található az eszköz lokációjától, a távolság mérés tévedhet pár métert, de ez földméréskor nem releváns. A földmérés célja egy koordináta centiméter pontos meghatározása a való világban. Arról, hogy ismerjük az eszköz pontos koordinátáit az NTRIP gondoskodik, és miután az eszköz koordinátája már pár méteres távolságon belülre kerül a kitűzött koordinátához, az alkalmazás már centiméter pontossággal meg tudja határozni a távolságot. Ha centiméter pontos távolságot akkor is tudni szeretnénk, ha a kitűzött pont kilométeres távolságra van az eszközhöz, akkor a lokációkat valamilyen PCS rendszerbe kellene konvertálni és ebben a rendszerben végezni a számolásokat. Ezek a műveletek azonban rendkívül leterhelőek lennének az eszköz számára, ha minden akár centiméternyi pozíció váltáskor ki kellene számolni őket. Ha távoli pontok

egymás közötti távolságának mérése a cél, akkor a koordináta számolások funkciót fogja használni a felhasználó.

Amikor az eszköz már egy méteren belülre kerül a kitűzött ponthoz a térkép kicserélődik egy új képernyőre. Egy méteres távolságnál a térképet már nem lehet eléggé kinagyítani ahhoz, hogy a kitűzött pont pontosan meghatározható legyen a terepen. Ezt a logikát a CloseUpFragment-ben valósítottam meg. A CloseUpFragment 2 különböző UI-t jelenít meg a távolság alapján, ezzel tovább segítve a felhasználó munkáját. Az első UI 100-31 cm távolságon belül, a második 30-0 cm távolságon belül. Mind a kettő UI felépítése hasonló, csak a távolság nagyítása különböző. Mind a két képernyő közepén található egy fix pozíciójú Marker ikon, ami a kitűzött pontot jelzi. A felhasználó pozícióját egy piros ikon jelzi. Ezeket a képernyőket mutatja be az 5.5 Ábra.



5.5 Ábra. CloseUpFragment

Ennek a képernyőnek azt a feladatot kellett megoldania, hogy térkép nélkül, egyszerű Android layout elemekkel jelenítse meg 2 koordináta viszonyát: a kitűzött pont és az eszköz pozíciójának viszonyát. Valamilyen módon vizualizálni kell a két pont közötti távolságot és szöget/irányt. Természetesen figyelembe véve azt is, hogy a layout-ok minden Android eszközön a kijelző méretétől függően más-más módon jelenhetnek meg. A feladat megoldására legalkalmasabb eszköznek a ConstraintLayout-ot találtam. Ez a Layout típus az elemek

pozícióját úgynevezett constraint-ekkel határozza meg, azaz egymáshoz relatív módon tudjuk elhelyezni az elemeket. Például beállítható az, hogy *elem B* a jobb oldalán legyen *elem A*-nak, 10 pixeles margóval. A ConstraintLayout-nak egy olyan funkciója is van, aminek segítségével az elemeknek kör alapú contrainteket is adhatunk. *Elem A* definiálható a kör középpontja ként, és *elem B* pozíciója definiálható a középpontból vett sugár nagyságával és szögével. Mivel a kitűzött pont koordinátái földméréskor fix-ek, ezért a kitűzött pont megjeleníthető a kör középpontjaként, a képernyő közepén egy fix pontban. A felhasználó koordinátái ezzel szemben folyamatosan változnak és a kitűzött pontot szeretnék elérni, ezért a felhasználó helyzetét reprezentáló ikon a kör középpontjához relatív módon van elhelyezve a két koordináta távolsága és szöge alapján. Amikor a felhasználó 5cm belüli távolságra kerül a kitűzött a ponthoz az alkalmazás hangjelzést ad ki.

Ahhoz, hogy a távolságok és ikonok méretei minden Android eszközön egyformán jelenjenek meg, az eszközök felbontásától függetlenül, a layout-ot dp (density-independent pixels) mértékegységgel kell leírni. Ahhoz, hogy a távolság megfelelő módon legyen nagyítva a képernyőn először meg kell határozni, hogy a valóságban 1cm távolság hány dp-nek felel meg az aktuális eszközön. Ennek a kiszámítása a CloseUpFragment onCreateView metódusában történik, az 5.6 Ábrán látható módon.

```
Configuration configuration = getActivity().getResources().getConfiguration();
int screenWidthDp = configuration.screenWidthDp;
radius = (float)screenWidthDp/2;
cm_to_dp_100 = (radius / 100);
cm_to_dp_30 = (radius / 30);

density = getResources().getDisplayMetrics().density;
```

5.6 Ábra. Mértékegység kiszámítása

A UI-t minden lokáció változáskor frissíteni kell, hogy a ConstraintLayout tükrözze a távolság és szög változásokat. Amikor az eszköz 30 centiméteren belül van a célhoz, akkor a távolság a *cm_to_dp_30* változóval kerül kiszámolásra, ellenkező esetben pedig a *cm_to_dp_100* változóval. A *densitiy* számítására azért van szükség, mert a Layout tulajdonságok Java kódból csak pixel mértékegységgel módosíthatók, így a kiszámolt távolság értéket a *density* változó segítségével dp mértékegységből pixelre kell váltani. A UI frissítéséért felelős logikát az 5.7 Ábra mutatja.

```
private void updateUI(){
    boolean close;
    try {
        int dist;
        if(mLiveLocation.distanceTo(mDestLocation)*100 < 30){
            close = true;
            mCompass.setVisibility(View.INVISIBLE);
            mRadius.setImageDrawable(getActivity().getDrawable(R.drawable.circle_grey));
            dist = (int)((mLiveLocation.distanceTo(mDestLocation)*100)*cm_to_dp_30);
            mLocationImageView.setImageDrawable(getActivity().getDrawable(R.drawable.ic_mylocation));
        }else{
            close = false;
            mCompass.setVisibility(View.VISIBLE);
            mRadius.setImageDrawable(getActivity().getDrawable(R.drawable.circle_outline));
            dist = (int)((mLiveLocation.distanceTo(mDestLocation)*100)*cm_to_dp_100);
            mLocationImageView.setImageDrawable(getActivity().getDrawable(R.drawable.ic_navigation));
        }
        ConstraintLayout.LayoutParams layoutParams = (ConstraintLayout.LayoutParams) mLocationImageView.getLayoutParams();
        layoutParams.circleRadius = (int) (dist * density);
        float bearing = mDestLocation.bearingTo(mLiveLocation);
        if(bearing < 0){
            bearing+=360;
        }
        layoutParams.circleAngle = bearing;
        mLocationImageView.setLayoutParams(layoutParams);
        if(dist <= 75 && close){
            mToneGenerator.startTone(ToneGenerator.TONE_CDMA_PIP);
            mRadius.setImageDrawable(getActivity().getDrawable(R.drawable.circle_green));
        }else if(dist > 75 && close){
            mRadius.setImageDrawable(getActivity().getDrawable(R.drawable.circle_grey));
            mToneGenerator.stopTone();
        }else
            mToneGenerator.stopTone();
    }catch (Exception e){
        Log.d( tag: "CloseUpFragment", e.toString());
    }
}
```

5.7. Ábra. UI frissítése

5.1.3 Pontok kezelése

A térképen a Markerekre kattintva előhozható egy form-ot tartalmazó dialogbox. A from tartalmazza a pontok adatait: név, szélesség, hosszúság, magasság, kód. Ezeket az adatokat a form segítségével megváltoztathatjuk. Egy műveleti gomb segítségével új pontot is fel tudunk venni, ami az eszköz aktuális koordinátáival kerül inicializálásra. A pontok az eszköz saját memóriájába kerülnek mentésre SharedPreferences használatával. A SharedPreferencesből történő kiolvasásra és a SharedPreferces-be írásra létrehoztam egy PointBase singleton osztályt, ami elvégzi az adatbázis műveleteket. Első inicializáláskor (az alkalmazás indulásakor) beolvassa az elmentett pontokat memóriába és egy listában tárolja őket, a gyors eléréshez.

5.2 Számolások

Minden a webalkalmazás által megvalósított koordináta számolásra a mobilalkalmazás is képes. A műveleteket lokálisan is ki tudja számolni, de az API megfelelő endpointjait

meghívva is lehetőség van számításokat végezni. A beállításokban döntheti el a felhasználó, hogy melyik megoldást válassza. Alapértelmezetten lokálisan történnek a számítások. A webalkalmazáshoz hasonlóan az első lépés itt is a megfelelő CRS kiválasztása. A mobilapp azonban nem tudja automatikusan kiválasztani a legoptimálisabb CRS-t, ezt a felhasználó megteheti egy listából a beállítások fülön. Az alapértelmezett CRS az UTM, ami elég nagyfokú pontosságot biztosít. A felhasználó így akkor is pontos eredményeket kaphat, ha nincs háttértudása a CRS-ekről vagy nincs hozzáférése az API-hoz. A koordináták WGS84-ből UTM-re való átváltásához saját fejlesztésű UTM osztályt használok, pár internetes fórum bejegyzést alapul véve. Nemzeti CRS-ek konverziójára Proj4j könyvtárat használom. A számoláshoz használt képletek ugyanazok, mint a webalkalmazás esetében. A számolásokat elvégezni a felhasználó az Eszközök > Koordináta mérések képernyőn tudja végezni. Ezen a képernyőn a Mérés képernyőhöz hasonlóan egy GoogleMaps térkép van műholdas nézetben. A korábban elmentett pontok itt is Markerekkel vannak jelezve. A Markereket rájuk kattintva kiválaszthatjuk és a képernyő jobb alsó részén elhelyezkedő gombokkal végezhetjük el a különböző számolási feladatokat: távolság, terület, északhoz viszonyított szög, osztópont.

5.3 Kommunikáció az API-val

Az API-val való kommunikációt okhttp3 és retrofit2 könyvtárak segítségével valósítottam meg. Retrofit segítségével az alkalmazásban a pontok reprezentálására használt modell osztály egyszerűen inicializálható JSON formájú API válaszokból és egyszerűen alakítható át JSON formájú alakba HTTP kérésekhez. RetrofitAdapter osztály létrehoz egy Retrofit példányt, amely az API BASE_URL alapcímre csatlakozik. A RetrofitAdapter a Gson-t használja JSON adat feldolgozására, beleértve az OffsetDateTime típusú adatokat, amelyeket egy egyedi JSON deszerializáló segítségével alakít át. A RetrpfitAdapter osztályt az 5.8 Ábra szemlélteti.

5.8 Ábra. RetrofitAdapter

Az API használatához a felhasználónak először be kell jelentkeznie. Ezt az Eszközök fülön keresztül elérhető bejelentkezés oldallal teheti meg. Bejelentkezéskor a szervertől visszakapott refresh és access Bearer tokeneket SharedPreferences-ben tárolja az alkalmazás. Erre a feladatra a React apphoz hasonlóan itt is saját interceptort használok. A JwtInterceptor osztály felelős a JWT token kezeléséért és a hitelesítési folyamat kezeléséért. Az osztály az okhttp3 könyvtár Interceptor interfészét implementálja. Az interceptor minden kérés előtt lekéri a hozzáférési tokent SharedPreferences-ből, és ezt hozzáadja a kérés Bearer fejlécéhez. Ha a válasz 401-es hibakódot ad vissza, ami azt jelzi, hogy a token lejárt, akkor az intercept metódus próbálja frissíteni a tokent. Ha a frissítési kísérlet sikeres, az interceptor újraküldi az eredeti kérést az új tokent tartalmazó fejléccel. Ha a token frissítése sikertelen, akkor a felhasználót kijelentkezteti, és egy üzenetet jelenít meg, amely tájékoztatja, hogy újra be kell jelentkeznie a funkció eléréséhez. A felhasználónak lehetősége van lekérni a szerveren tárolt pontokat, ezekkel felülírni a saját eszközön tárolt pontokat, lehetősége van a pontokat szinkronizálni a mobil és web felületek között, illetve minden számolási funkcióhoz igénybe veheti az API-t. Szinkronizáció során az alkalmazás először lekérdezi az összes szerveren tárolt pontot. Majd a pontok updatedAt dátum adattagja alapján eldönti, hogy mely pontokat kell saját eszközön frissíteni és melyeket a szerveren. A szerveren történő frissítéshez meghívja a releváns API endpointokat PATCH, POST és DELETE kérésekkel.

6. Webalkalmazás kitelepítése

Annak érdekében, hogy a mobilapp és a webapp tudjanak egymással telepíteni, webapp-ot ki kell telepíteni. Az alkalmazás kitelepítését nagyban segíti a 3.2 fejezetben taglalt Dockerizáció. A webapp két Docker-konfigurációval rendelkezik: egy fejlesztői (dev) környezettel és egy kitelepítésre alkalmas gyártási (prod) környezettel. Mindkét konfiguráció több konténerből áll: django, celery, redis, react, nginx, és postgresql. A konfigurációkat docker-compose segítségével futtathatjuk, amelyhez egy yml fájl adja meg a szükséges konténereket, a kapcsolódó Docker Image-eket, környezeti változókat, illetve a volumeneket, amelyek az adatok tárolására szolgálnak a konténerek számára. A konténerek képfájljai részben Docker Hub-ról származnak, mint a redis és postgresql, míg más konténerek esetében saját .Dockerfile konfigurációkat használunk, amelyeket helyileg kell buildelni.

A fejlesztői környezetben különösen fontos, hogy a kódbeli változtatások azonnal megjelenjenek, anélkül, hogy újra kellene buildelni a képfájlokat. Ezt a különböző szerver indítási parancsok és Chokidar polling beállításával érem el, amely folyamatosan figyeli a kód változásait, és azonnal frissíti az alkalmazást. Ez viszont megnövekedett erőforrás-igénnyel jár, ezért a prod környezetben nem szükséges.

Mind a dev, mind a prod környezetben kritikus szerepet játszik az NGINX konténer, amely reverz proxyként szolgál, a bejövő kéréseket pedig az adott szolgáltatáshoz irányítja. Az NGINX biztosítja, hogy a felhasználói kérések – legyenek azok az admin felület, az API, vagy a websocket kapcsolat kérései – megfelelően kerüljenek a django backendhez, míg minden más kérést a react frontendhez irányít. Emellett az NGINX a statikus fájlokat is kezeli, így a static és media kötetekből közvetlenül kiszolgálja ezeket, amely jelentős teljesítményjavulást eredményez a Django szerver tehermentesítésével. A gyártási környezet további optimalizációkat is tartalmaz, hogy kitelepítésre alkalmas legyen. A gyártási NGINX konfiguráció a gyorsítótárazást és a proxy beállításokat is optimalizálja a teljesítmény és a biztonság érdekében. A websocket támogatás például valós idejű kommunikációt tesz lehetővé a felhasználói felület és a backend között, amely elengedhetetlen a valós idejű értesítések és interakciók esetén. Továbbá a *proxy_set_header* direktívákkal lehetővé teszi, hogy a felhasználó IP-címét a backend számára is továbbítsa, ami fontos biztonsági és naplózási szempontból. Prod környezetben az NGINX beállítása tehát nemcsak a teljesítményhez, hanem a biztonsághoz is nagyban hozzájárul, hiszen védi a belső szolgáltatásokat a közvetlen hálózati hozzáférés elől, és a statikus fájlokat gyorsítótárként kiszolgálva optimalizálja a válaszidőket.

A webalkalmazás React konténere a frontend kiszolgálására szolgál. Fejlesztői környezetben a szerver *npm run dev* parancs segítségével indul, amely figyeli a fájlok módosításait és automatikusan újratölti az alkalmazást, amikor változások történnek. Ez különösen hasznos fejlesztés során, mivel a kódbeli módosítások azonnal megjelennek, anélkül, hogy a konténert újra kellene indítani. Prod környezetben a react konténer optimalizálva van a gyors és hatékony kiszolgálás érdekében. Itt a React alkalmazás egy statikus HTML, CSS, és JavaScript fájlokból álló csomaggá buildelődik az *npm run build* parancs segítségével, amely minifikált és optimalizált fájlokat generál. Ez a build folyamat csökkenti a fájlméretet és gyorsítja a betöltési időt. A *node_modules* és a *build* mappa a buildelés után törlésre kerül, hogy csak a szükséges optimalizált statikus fájlok maradjanak meg a konténerben, így jelentősen csökkentve annak méretét és javítva a telepítés hatékonyságát.

Fejlesztői környezetben a Django szerver a következő paranccsal indul: *python manage.py runserver 0.0.0.0:8000*. Ez a beépített Django fejlesztői szervert indítja el, amely egyszerűen és gyorsan elérhetővé teszi az alkalmazást a fejlesztők számára. A *runserver* parancs segítségével a fejlesztői környezetben végzett változtatások azonnal láthatók, mivel a szerver automatikusan újraindul, ha bármilyen kódbeli módosítás történik. Ez a szerver azonban nem optimalizált a nagy terhelés kezelésére, és nem képes megfelelően kiszolgálni több párhuzamos kérést, ezért kizárólag fejlesztési célokra alkalmas. Ezzel szemben a prod környezetben a következő parancsot használják: *CMD ["gunicorn", "fieldtracker.wsgi:application", "-b","0.0.0.0:8000","--access-logfile","-"]*. Ebben az esetben a gunicorn (Green Unicorn) alkalmazásszerver szolgálja ki a Django alkalmazást. A gunicorn egy WSGI szerver, amely kifejezetten gyártási környezethez van optimalizálva, így képes magas terhelés alatt is hatékonyan kezelni a beérkező kéréseket. Több processzt és szálat is képes párhuzamosan futtatni, ami javítja a teljesítményt és a stabilitást. A --access-logfile "-" paraméter pedig lehetővé teszi a kérések naplózását, így részletes információkat kapunk a szerver használatáról és terheléséről.

Dockerizált alkalmazás kitelepítésére több platform létezik. A legnépszerűbb felhőszolgáltatók az Amazon Web Services (AWS), Microsoft Azure és a Google Cloud. Kisebb szolgáltatók is léteznek például a Heroku vagy a DigitalOcean. Ezek természetesen mind fizetős szolgáltatások, de nagyobb szolgáltatók egyes termékeit jellemzően egy bizonyos limitig ingyenesen lehet használni. Én az AWS szolgáltatásait választottam, mert 12 hónapig ingyenesen biztosít egy linux gépet, amit kényelmesen linux terminál segítségével lehet kezelni. Ez a termék az Amazon Elastic Compute Cloud (Amazon EC2) nevet viseli. Több instance típus létezik belőle különböző felszereltségekkel. A legnagyobb teljesítményű instance, ami

még ingyenesen hozzáférhető a t3.micro nevet viseli. Ez az instance típus 2 vCPU-val (virtual central processing unit) rendelkezik és 1 GB RAM-mal. Ez a felszereltség a prod docker konfigurációval elegendő a fejlesztéshez és egy kis felhasználó bázis kiszolgálásához.

Az alkalmazás kitelepítéséhez a linux szerveren kelleni fog a docker compose konfigurációt tartalmazó yml fájl. Ezt legegyszerűbben git-en keresztül lehet eljuttatni a szerverre. A docker image-ek buildelését saját gépen kell futtatni, mert a buildelés leterheli a szervert és nem rendelkezik elegendő erőforrással. A lokálisan buildelt Image-eket fel kell tölteni Dockerhub-ra. Dockerhubon ingyenesen létrehozható egy privát repository, ahova bejelentkezés után feltölthetőek az image-ek. A szerveren lévő docker-compose.yml fájlban ezekre az image-kre kell hivatkozni. Ezért a prod docker környezetnek két yml fájlt csináltam, egyet a lokális buildelésre, ami lokális Dockerfile-okra hivatkozik, egyet pedig a szerveren történő futtatásra. Kitelepítéskor az alkalmazás lokális buildelése után a *docker push* paranccsal feltöltöm az új képeket a DockerHub repoba. Ezt követően a szerveren *docker pull* paranccsal letöltöm őket. A prod docker konfiguráció futtatása után a webalkalmazás nyilvánosan elérhető válik az Amazon EC2 instance nyilvános IP címén keresztül.

7. ÖSSZEFOGLALÁS

Az agrárszektor digitalizációja és a precíziós gazdálkodás térnyerése kulcsfontosságú a fenntartható és hatékony termelés megvalósításában. Magyarországon a precíziós mezőgazdaság elterjedése még gyerekcipőben jár, főként a magas költségek és az alacsony információáramlás miatt. A dolgozat célja egy olyan szoftverfejlesztési megoldás bemutatása volt, amely egyszerűsített felületével csökkenti a technológiai akadályokat, és a gazdák számára könnyen hozzáférhetővé teszi a helyspecifikus adatok felhasználását.

A dolgozat során bemutatott projekt azt is bizonyította, hogy meglévő nyílt forráskódú könyvtárak és keretrendszerek hatékony eszközök lehetnek mezőgazdasági alkalmazások fejlesztésére. A projektben kifejlesztett GeoMeasurements könyvtár lehetővé teszi földrajzi adatok pontos kezelését, anélkül, hogy a fejlesztőnek speciális földrajzi szaktudással kellene rendelkeznie. Ez nemcsak a fejlesztési időt és költségeket csökkenti, hanem szélesebb körben is hozzáférhetővé teszi az ilyen típusú megoldásokat.

A dolgozat során megvalósított projekt úgy lett kialakítva, hogy könnyen továbbfejleszthető legyen. GeoDjango integrálásával a webalkalmazás egyszerűen felkészíthető GIS adatok tárolására, amit szeretnék elvégezni a jövőben. A webalkalmazás által gyűjtött adatokból könnyen döntéstámogató rendszert készíthető, akár mesterséges intelligencia

használatával. A mobilalkalmazás által bemért koordináták pedig alapjául szolgálhatnak más precíziós mezőgazdasági rendszereknek. Ilyenek lehetnek például az kormányzást segítő rendszerek vagy a drónnal működő rendszerek.

Irodalomjegyzék

[1] Bongiovanni, R., Lowenberg-Deboer, J. Precision Agriculture and Sustainability. Precision Agriculture, 5, 359–387 (2004). DOI: 10.1023/B .0000040806.39604.aa.

[2] www.ksh.hu, Központi Statisztikai Hivatal Agrárcenzus eredmények (2020). Elérhetőség: https://www.ksh.hu/docs/hun/xftp/ac2020/agrardigitalizacio/index.html#apreciseszkzkelterjedtsgehaznkbanalacsonyszint (Megtekintés dátuma: 2024. november 17.).

[3] www.ksh.hu, Központi Statisztikai Hivatal Agrárium 2023 végleges adatok. Elérhetőség: https://www.ksh.hu/s/kiadvanyok/agrarium-2023-vegleges-adatok/index.html (Megtekintés dátuma: 2024. november 17.).

[4] McFadden, J., Njuki, E., & Griffin, T. Precision agriculture in the digital era: Recent adoption on U.S. farms. Economic Information Bulletin No. (EIB-248). U.S. Department of Agriculture (2023). Elérhetőség: https://www.ers.usda.gov/publications/pub-details/?pubid=105893 (Megtekintés dátuma: 2024. november 17.).

[5] Balogh P., Bai A., Czibere I., Kovách I., Fodor L., Bujdos Á., Sulyok D., Gabnai Z., Birkner Z. Economic and Social Barriers of Precision Farming in Hungary. Agronomy, 11(6), 1112 (2021). DOI: 10.3390/agronomy11061112.

[6] Schimmelpfennig, D. Crop production costs, profits, and ecosystem stewardship with precision agriculture. Journal of Agricultural and Applied Economics, 50(1), 81–103 (2018). DOI: 10.1017/aae.2017.23.

[7] surpadapp.com, SurPad földmérő alkalmazás weboldal. Elérhetőség: https://surpadapp.com/ (Megtekintés dátuma: 2024. november 17.).

[8] García-Peñalvo, F. J., Conde, M. Á., & Matellán-Olivera, V. Mobile apps for older users – The development of a mobile apps repository for older people. Lecture Notes in Computer Science, vol 8524 (2014). DOI: 10.1007/978-3-319-07485-6_12.

[9] www.gps.gov, GPS Accuracy. Elérhetőség: https://www.gps.gov/systems/gps/performance/accuracy/ (Megtekintés dátuma: 2024. november 17.).

[10] Xu, H. Application of GPS-RTK technology in the land change survey. Procedia Engineering, 29, 3454–3459 (2012). DOI: 10.1016/j.proeng.2012.01.511.

[11] intelligeo.hu, Mi is az az RTK korrekció?. Elérhetőség: https://intelligeo.hu/20241021/mi-az-az-rtk-korrekcio (Megtekintés dátuma: 2024. november 17.).

[12] rtk2go.com, How it works. Elérhetőség: http://rtk2go.com/how-it-works/ (Megtekintés dátuma: 2024. november 17.).

[13] gnssnet.hu, Tájékoztató weboldal. Elérhetőség: https://www.gnssnet.hu/index.php?r=site%2Frealtime (Megtekintés dátuma: 2024. november 17.).

[14] globalgpssystems.com, RTK GNSS Receivers price comparison. Elérhetőség: https://globalgpssystems.com/gps-receivers/rtk-gnss-receivers/ (Megtekintés dátuma: 2024. november 17.).

[15] ardusimple.com, RTK Starter Kits. Elérhetőség: https://www.ardusimple.com/rtk-starter-kits/ (Megtekintés dátuma: 2024. november 17.).

[16] developer.mozilla.org, Django introduction. Elérhetőség: https://developer.mozilla.org/en-US/docs/Learn/Server-side/Django/Introduction (Megtekintés dátuma: 2024. november 17.).

[17] Sood, K., Singh, S., Rana, R. S., Rana, A., Kalia, V., & Kaushal, A. Application of GIS in precision agriculture. Centre for Geo-Informatics Research and Training. CSK Himachal Pradesh Agriculture University (n.d.).

[18] docs.djangoproject.com, GeoDjango. Elérhetőség: https://docs.djangoproject.com/en/5.1/ref/contrib/gis/ (Megtekintés dátuma: 2024. november 17.).

[19] pypi.org, Python Package Index geo-measurements. Elérhetőség: https://pypi.org/project/geo-measurements/ (Megtekintés dátuma: 2024. november 18.).

[20] github.com, GeoMeasurements nyilvános GitHub repository. Elérhetőség: https://github.com/b3n3c/geo_measurements (Megtekintés dátuma: 2024. november 18.).

[21] Disser, Y., Mihalák, M., & Widmayer, P. A polygon is determined by its angles. Computational Geometry, 44(8), 418–426 (2011). DOI: 10.1016/j.comgeo.2011.04.003.

[22] earthobservatory.nasa.gov, Agricultural patterns NASA Earth Observatory. Elérhetőség: https://earthobservatory.nasa.gov/images/6605/agricultural-patterns (Megtekintés dátuma: 2024. november 17.).

[23] github.com, api-ntrip-java-client. Elérhetőség: https://github.com/OneStopTransport/api-ntrip-java-client (Megtekintés dátuma: 2024. november 17.).