BOTANIAI

Projektna dokumentacija

2022/2023

UM FERI

Lahovnik, Borak, Drozg, Jevšnik

[Kazalo slik 3](#_Toc137639148)

[Uvod 4](#_Toc137639149)

[Namen 4](#_Toc137639150)

[Izbrane tehnologije 4](#_Toc137639151)

[Razdelitev dela 4](#_Toc137639152)

[Organizacija dela 4](#_Toc137639153)

[Predvideni uporabniki 5](#_Toc137639154)

[Struktura podatkovne baze 7](#_Toc137639155)

[Podatki o rastlini 7](#_Toc137639156)

[Podatki o uporabniku 8](#_Toc137639157)

[Glavne aktivnosti uporabnikov 10](#_Toc137639158)

[Klasifikacija rastline 10](#_Toc137639159)

[Virtualni vrt 11](#_Toc137639160)

[Dodajanje rastline v virtualni vrt 12](#_Toc137639161)

[Urejanje rastline v virtualnem vrtu 13](#_Toc137639162)

[Izbris rastline iz virtualnega vrta 14](#_Toc137639163)

[Pregled podrobnosti rastline 15](#_Toc137639164)

[Beleženje zalivanja rastlin 16](#_Toc137639165)

[Rezultati in produkti 17](#_Toc137639166)

[Napovedni model 17](#_Toc137639167)

[Metrike napovednega modela 17](#_Toc137639168)

[Validacijska točnost in izguba 19](#_Toc137639169)

[Uporabljeni sloji 19](#_Toc137639170)

[Zaledni sistem 20](#_Toc137639171)

[Glavne knjižnice 20](#_Toc137639172)

[Mobilna aplikacija 21](#_Toc137639173)

[Glavne knjižnice 21](#_Toc137639174)

[Scraper vsebin 21](#_Toc137639175)

[Izzivi, na katere smo naleteli tekom razvoja 21](#_Toc137639176)

[Nekompatibilnost knjižnic v pythonu 21](#_Toc137639177)

[Nekompatibilnost knjižnic v mobilni aplikaciji 21](#_Toc137639178)

[Augmented reality 22](#_Toc137639179)

[Namestitev 22](#_Toc137639180)

[Zaledni sistem 22](#_Toc137639181)

[Postopek namestitve 22](#_Toc137639182)

[Mobilna aplikacija 24](#_Toc137639183)

[Lokalna uporaba 24](#_Toc137639184)

[Postopek namestitve 25](#_Toc137639185)

[Klasifikacijski model 25](#_Toc137639186)

[Postopek namestitve 25](#_Toc137639187)

[Zagotavljanje kakovosti 26](#_Toc137639188)

[SonarQube 26](#_Toc137639189)

[Zaledni sistem 26](#_Toc137639190)

[Mobilna aplikacija 27](#_Toc137639191)

[Klasifikacija 29](#_Toc137639192)

[Testi in CI/CD 30](#_Toc137639193)

[Testi 30](#_Toc137639194)

[Avtomatizirano izvajanje testov 30](#_Toc137639195)

# Kazalo slik

[Slika 1: Prvotni diagram primerov uporabe 5](#_Toc137639196)

[Slika 2: Prilagojen diagram primerov uporabe 6](#_Toc137639197)

[Slika 3: Struktura dokumenta Plant 7](#_Toc137639198)

[Slika 4: Struktura dokumenta User 8](#_Toc137639199)

[Slika 5: Struktura objektov v polju wateringArray 9](#_Toc137639200)

[Slika 6: Diagram aktivnosti klasifikacija rastline 10](#_Toc137639201)

[Slika 7: Diagram aktivnosti virtualni vrt 11](#_Toc137639202)

[Slika 8: Diagram aktivnosti dodajanje rastline 12](#_Toc137639203)

[Slika 9: Diagram aktivnosti urejanje rastline 13](#_Toc137639204)

[Slika 10: Diagram aktivnosti izbris rastline 14](#_Toc137639205)

[Slika 11: Diagram aktivnosti pregled podrobnosti rastline 15](#_Toc137639206)

[Slika 12: Diagram aktivnosti beleženje zalivanja rastline 16](#_Toc137639207)

[Slika 13: Končno število učnih primerkov glede na razred 17](#_Toc137639208)

[Slika 14: Metrike napovednega modela 17](#_Toc137639209)

[Slika 15: Matrika zmede testnih podatkov 18](#_Toc137639210)

[Slika 16: Pravilna vrsta 18  
Slika 17: Napovedana vrsta 18](#_Toc137639211)

[Slika 18: Graf točnosti in izgube 19](#_Toc137639212)

[Slika 19: Graf točnosti in izgube nad neprilagojenimi učnimi podatki 19](#_Toc137639213)

[Slika 20: Metrike kakovosti nad neprilagojenimi podatki 19](#_Toc137639214)

[Slika 21: Zaporedje slojev v napovednem modelu 20](#_Toc137639215)

[Slika 22: Ponudbe Render-a 22](#_Toc137639216)

[Slika 23: Izberemo željen repozitorij 23](#_Toc137639217)

[Slika 24: Izpolnimo informacije novega Web Service-a 23](#_Toc137639218)

[Slika 25: Izpolnimo še preostale informacije 23](#_Toc137639219)

[Slika 26: Možnosti okoljskih spremenljivk 24](#_Toc137639220)

[Slika 27: Vnos okoljske spremenljivke 24](#_Toc137639221)

[Slika 28: Expo.dev in stanje prevajane aplikacije 25](#_Toc137639222)

[Slika 29: Prvi zagon SonarQube-a na kodi zalednega sistema 26](#_Toc137639223)

[Slika 30: Prikaz enega izmed varnostno kritičnih delov kode 26](#_Toc137639224)

[Slika 31: Zagon analize po spremembi zalednega sistema 27](#_Toc137639225)

[Slika 32: Prvi zagon SonarQube-a na kodi mobilne aplikacije 27](#_Toc137639226)

[Slika 33: Napaka možnega puščanja vrednosti 28](#_Toc137639227)

[Slika 34: Napaka določanja ključa elementa z vrednostjo iz funkcije map 28](#_Toc137639228)

[Slika 35: Končni rezultat analize s programom SonarQube 28](#_Toc137639229)

[Slika 36: Prvi rezultati SonarQube za klasifikacijo 29](#_Toc137639230)

[Slika 37: Predlog SonarQube za zakomentirano kodo 29](#_Toc137639231)

[Slika 38: Status klasifikacije SonarQube-a po popravkih kode 30](#_Toc137639232)

[Slika 39: Datoteka za avtomatizirano izvajanje testov 31](#_Toc137639233)

# Uvod

## Namen

Ideja je nastala zaradi pomembnosti rastlin in njihove pravilne skrbi. Rastline nas obkrožajo povsod na svetu, zaradi česar je pomembno, da vemo katere rastline nas neposredno obdajajo in ali nas lahko kakšne rastline potencialno življenjsko ogrožajo. S tem namenom smo ustvarili aplikacijo, ki omogoča prepoznavo različnih vrst rastlin glede na fotografijo. Na podlagi fotografije nato klasifikacijski model napove vrsto rastline in o tej vrsti si lahko preberemo več informacij. Hkrati omogoča shranjevanje rastlin v svoj virtualni vrt, kjer si lahko tudi izbere na koliko dni je potrebno rastlino zaliti in prvi dan zalivanja, na podlagi česar bo nato uporabnik prejemal potisna obvestila (push notifications). Aplikacija je torej namenjena vsem ljubiteljem rastlin, vsem, ki želijo izvedeti več informacij o razičnih vrstah rastlin ter vsem, ki se želijo naučiti več o pravilni skrbi za posamezno rastlino.

## Izbrane tehnologije

Za programske jezike smo izbrali python za klasifikacijo, JavaScript za zaledni sistem in TypeScript za mobilno aplikacijo. Na področju klasifikacije smo si pomagali z uporabo knjižnic, kot so Tensorflow, Numpy, Scikit-learn, matplotlib in seaborn. V sklopu zalednega sistema smo izbrali ogrodje ExpressJS, s pomočjo različnih knjižnic, kot je Mongoose za povezavo na MongoDB podatkovno bazo, dotenv za okoljske spremenljivke in Swagger za dokumentacijo API-ja. V sklopu mobilne aplikacije smo uporabili ogrodje React Native, s pomočjo različnih knjižnic, kot je Expo, zaradi česar je bilo možno aplikacijo uporabljati v ExpoGo emulatorju, sicer smo pa uporabili nekatere druge pomembne knjižnice, kot je expo-camera za uporabo kamere v aplikaciji in firebase, za avtentikacijo in uporabo firebase functions.

Za te tehnologije smo se odločili, zato ker jih vsi vsaj nekoliko poznamo, kar je olajšalo razvoj in zaradi česar je bilo reševanje težav, ki so nastopile lažje. Za NoSQL podatkovno bazo smo se odločili, saj nam omogoča lažje shranjevanje slik in podatkov, saj nismo toliko omejeni s fiksno strukturo in jo lahko dinamično spreminjamo, če bi bilo potrebno hraniti nove podatke ali specifične podatke za nekatere dokumente.

## Razdelitev dela

Ena oseba se je posvetila razvoju zalednega sistema, ena oseba se je posvetila razvoju klasifikacijskega modela, preostali dve osebi pa sta se posvetili razvoju mobilne aplikacije. Takšno razdelitev oziroma organizacijo znotraj skupine smo izbrali, saj je bilo največ dela ravno pri razvoju mobilne aplikacije. V kolikor smo svoje delo prej končali, smo pomagali tudi na drugih delih projekta, kjer je bilo največ dela.

## Organizacija dela

Pred začetkom razvoja projekta smo se v skupini dogovorili kdo bo delal na katerem delu projekta in katere tehnologije bodo izbrane. Med razvojem smo se na začetku sestali in izvedli uvoden sestanek, kjer smo šli skozi morebitne pomisleke in razložili morebitne nejasnosti. Tekom razvoje smo se vsak teden sestali na tedenskih Scrum sestankih, sicer so sestanki potekali po potrebi. V kolikor je bilo potrebnih več sestankov na teden smo jih izvedli. Za dogovarjanje smo uporabili aplikacijo Discord, kjer smo ustvarili skupinski pogovor in izvajali skupinske klice. Za organizacijo znotraj repozitorija smo pripravili korenske direktorije za vsak del projekta (zaledni sistem, mobilno aplikacijo, klasifikacijo, kasneje pa tudi za Firebase Functions in scraper-ja), ki smo jih tudi ustrezno poimenovali. Za poimenovanje commit-ov v repozitoriju smo se zgledovali po strukturi dostopni na: <http://karma-runner.github.io/6.4/dev/git-commit-msg.html> .

# Predvideni uporabniki

Predvideni uporabniki, funkcionalnosti aplikacije in integracije oziroma zunanje storitve so prikazani v naslednjem diagramu primerov uporabe, kot je bilo prvotno zamišljeno.

A picture containing text, diagram, line, drawing

Description automatically generated

Slika : Prvotni diagram primerov uporabe

Ker pa smo tekom razvoja naleteli na nekompatibilnost nekaterih knjižnic, ni bilo možno vgraditi klasifikacijskega modela, zaradi česar je bilo potrebno diagram primerov uporabe nekoliko prilagoditi.

A picture containing text, diagram, line, drawing

Description automatically generated

Slika : Prilagojen diagram primerov uporabe

# Struktura podatkovne baze

## Podatki o rastlini

Podatki o rastlinah so zbrani v kolekciji imenovani »Plant« in imajo naslednjo strukturo:

* \_id – unikatni identifikator,
* latin – latinsko ime rastline,
* common – navadno ime rastline,
* description – daljši opis rastline,
* watering – na koliko časa bi naj rastlino zalivali,
* sunlight – kolikšni količini sončne svetlobe bi naj bila izpostavljena,
* plantingTime – v katerih letnih časih rastlino sadimo,
* soil – v kakšnih prsteh rastlina uspeva,
* wateringDetail – podrobneje opisan način zalivanja,
* fertilization – na kakšen način lahko gnojimo rastlino,
* toxicity – ali je rastlina strupena in za koga,
* folder\_num – številka direktorija, v katerem se nahajajo primerki te rastline,
* image – ime datoteke slike.

A picture containing text, multimedia software, screenshot

Description automatically generated

Slika : Struktura dokumenta Plant

## Podatki o uporabniku

Podatki o uporabnikih so zbrani v kolekciji imenovani »User« in imajo naslednjo strukturo:

* \_id – unikatni identifikator,
* name – ime uporabnika,
* surname – priimek uporabnika,
* email – e-poštni naslov uporabnika,
* notifications – ali želi uporabnik prejemati potisna obvestila ali ne,
* history – zbrani rezultati klasifikacijskega modela, slika uporabljena za napoved, datum napovedi, rezultat klasifikacijskega modela ter napotki za skrb,
* personalGarden – zbrane rastline uporabniškega virtualnega vrta in podatki, ki so potrebni za skrb posamezne rastline.

A screenshot of a computer program

Description automatically generated with medium confidence

Slika : Struktura dokumenta User

A screenshot of a computer program

Description automatically generated with medium confidence

Slika : Struktura objektov v polju wateringArray

# Glavne aktivnosti uporabnikov

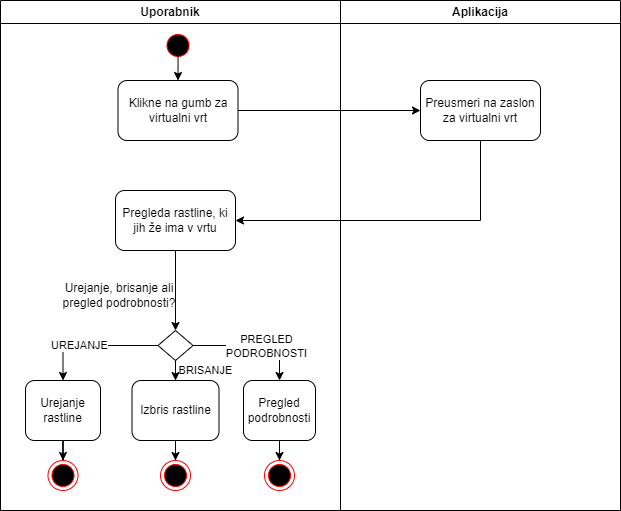
## Klasifikacija rastline

A picture containing text, diagram, screenshot, parallel

Description automatically generated

Slika : Diagram aktivnosti klasifikacija rastline

## Virtualni vrt



Slika : Diagram aktivnosti virtualni vrt

## Dodajanje rastline v virtualni vrt

A picture containing text, diagram, screenshot, plan

Description automatically generated

Slika : Diagram aktivnosti dodajanje rastline

## Urejanje rastline v virtualnem vrtu

A picture containing text, screenshot, diagram, parallel

Description automatically generated

Slika : Diagram aktivnosti urejanje rastline

## Izbris rastline iz virtualnega vrta

A picture containing text, screenshot, diagram, parallel

Description automatically generated

Slika : Diagram aktivnosti izbris rastline

## Pregled podrobnosti rastline

A picture containing text, screenshot, diagram, line

Description automatically generated

Slika : Diagram aktivnosti pregled podrobnosti rastline

## Beleženje zalivanja rastlin

A picture containing text, screenshot, diagram, parallel

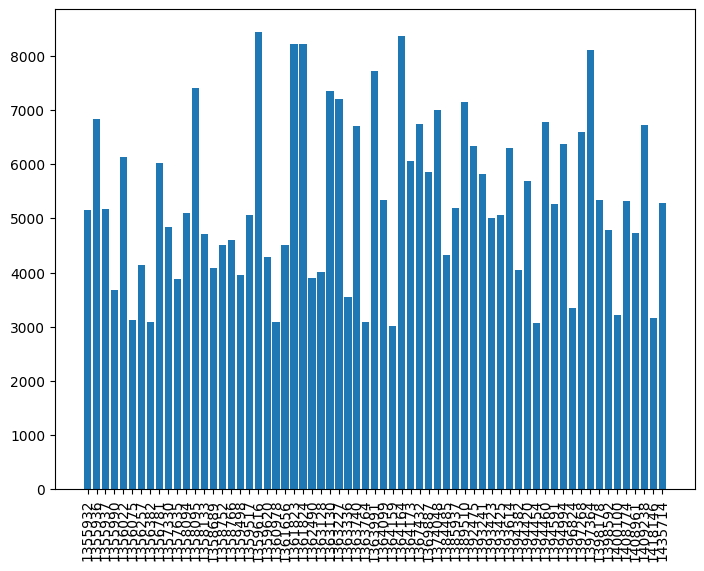
Description automatically generated

Slika : Diagram aktivnosti beleženje zalivanja rastline

# Rezultati in produkti

## Napovedni model

Končna verzija modela je bila naučena na “Augmented” podatkih, kar pomeni, da smo podatkovno množico umetno prilagodili. Tiste vrste, ki so imele manj kot 3000 primerkov (slik), smo umetno prilagodili. Vsako sliko smo vertikalno in horizontalno zrcalili in shranili kot nove primerke. Nato smo umetno prilagodili še podatke, ki so imeli manj kot 4000 podatkov, ampak še vrsta ni bila umetno prilagojena. V tem primeru smo vsako sliko zgolj vertikalno zrcalili. Na ta način smo se lahko tudi izognili underfit-tanju podatkov, kar se največkrat zgodi, ker imamo premalo primerkov za posamezno vrsto. Število vrst, ki smo jih uporabili za učenje modela, je 65.



Slika : Končno število učnih primerkov glede na razred

### Metrike napovednega modela

Končna verzija napovednega modela je dosegla 70% točnost nad testnimi primerki. Hkrati je dosegel 0,73 povprečno preciznost, 0,71 povprečen priklic in 0,71 povprečno F-mero. S testiranjem različnih parametrov, slojev, zaporedjem slojev, smo se poskusili čim bolj približati najboljšim vrednostim. Najboljše vrednosti bi bile: 100% za točnost in 1 za vse ostale metrike.

A picture containing text, screenshot, font, number

Description automatically generated

Slika : Metrike napovednega modela

Napovedi nad testnimi podatki lahko ovrednotimo tudi na podlagi matrike zmede. Vidimo lahko, da naš model ni delal preveliko napak. Kjer pa so bile prisotne napake, pa je bilo največkrat zaradi velike podobnosti različnih vrst – podobne so si bile tako v barvi kot obliki.

A picture containing screenshot, colorfulness, purple, line

Description automatically generated

Slika : Matrika zmede testnih podatkov

Primer, kjer je pravilna vrsta 1394454 (*lat.* *Pelargonium inquinans*), napovedana pa je bila vrsta 1435714 (*lat. Pelargonium zonale*).

A close-up of a pink flower

Description automatically generated with low confidence A close up of a flower

Description automatically generated with medium confidence

Slika : Pravilna vrsta Slika : Napovedana vrsta

Kot lahko vidimo iz slik, sta si vrsti precej podobni. Na podlagi imena pa lahko sklepamo, da tudi spadata v isti rod rastlin.

### Validacijska točnost in izguba

Na naslednjih grafih lahko vidimo, kako se je spreminjala točnost in izguba nad učnimi in validacijskimi podatki skozi epohe. Vidimo lahko, da se je validacijska točnost višala in izguba manjšala, kar tudi iščemo.

A picture containing line, screenshot, plot, text

Description automatically generated

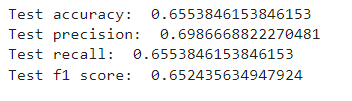
Slika : Graf točnosti in izgube

Napovedni model je bil naučen na 20 epohah in prilagojenih učnih podatkih. Dosegel je 70% točnost, kar je 5% več, kot je dosegel enak model na 30 epohah in navadnih, neprilagojenih učnih podatkih.

A picture containing line, plot, diagram, text

Description automatically generated

Slika : Graf točnosti in izgube nad neprilagojenimi učnimi podatki



Slika : Metrike kakovosti nad neprilagojenimi podatki

### Uporabljeni sloji

Uporabljeni so bili različni konvolucijski, združevalni, zgoščevalni in Dropout sloji. Konvolucijski sloji so namenjeni procesiranju (delov) slike, kjer se jedro sloja postopoma premika po sliki. Združevalni sloji so namenjeni zmanjševanju dimenzionalnosti in števila uteži. Dropout sloj se uporablja za preprečevanje prenatreniranosti modela. Temu sloju povemo kolikšen delež uteži z najmanjšim doprinosom naj izpusti oziroma pozabi. Na koncu uporabimo še Dense zgoščevalni sloj s številom razredov kot parameter.

A screenshot of a computer

Description automatically generated with medium confidence

Slika : Zaporedje slojev v napovednem modelu

Napovedni model na koncu učenja tudi izvozimo s pomočjo ukaza model.save(relativna\_pot\_shranjvanja). Ta format nato shranimo v functions/model direktorij, ki se lahko nato uporabi ob namestitvi Firebase Function-a na splet.

## Zaledni sistem

Zaledni sistem je bil ustvarjen s pomočjo ogrodja ExpressJS. Zaledni sistem je povezan z NoSQL podatkovno bazo v oblaku (MongoDB Atlas). Za povezavo na bazo je bila uporabljena javascript knjižnica MongoDB. Zaledni sistem ponuja različne metode za pridobivanje podatkov o rastlinah ali uporabnikovih podatkih. Za dostop do podatkov o določenem uporabniku je potreben veljaven žeton JWT tega uporabnika, ki se ob prijavi ustvari na strežniku ter pošlje na napravo. Na ta način vsaj do neke mere zavarujemo podatke uporabnikov, ki jih hranimo (ime, priimek, e-poštni naslov). Zaledni sistem gostimo pri ponudniku Render, ki omogoča povezavo z git repozitorijem in proženje novega grajenja zalednega sistema ob vsakem novem commit-u na izbrano vejo repozitorija.

### Glavne knjižnice

#### ExpressJS in MongoDB NodeJS gonilnik

Za ogrodje ExpressJs smo se odločili saj omogoča enostavno gradnjo spletnih aplikacij. Z njim smo ustvarili usmerjevalnike(routes) s katerimi smo obdelovali HTTP zahteve iz mobilne aplikacije. Strukturo zalednega sistema smo razdelili glede na kolekcijo iz podatkovne baze ter kasneje še na tip zahtevka (GET, POST, PUT DELETE). Strežnik izvaja poizvedbe na MongoDB podatkovno bazo v oblaku s pomočjo MongoDB NodeJS gonilnika.

#### Swagger-ui-express

Za lažje testiranje, boljše dokumentiranje ter lažje razumevanje kode smo dodali ogrodje Swagger-UI, kateri nam že sam s privzetimi nastavitvami omogoča interaktivno dokumentacijo API-jev naše aplikacije. Vsebuje opise poti, parametrov, odgovorov ter tudi sheme podatkov. Pomagal nam je hitro pridobiti vpogled v funkcionalnosti naše aplikacije ter lažje razumeti naš vmesnik med vsemi razvijalci v skupini.

#### Jsonwebtoken

Ker imamo naša aplikacija shranjuje tudi uporabnikove osebne podatke, smo se odločili da vsaj malo zavarujemo dostop do le teh z uporabo JWT žetonov. Za to smo uporabili knjižnico NodeJS »jsonwebtoken«, s katero ob vsaki avtentikaciji uporabnika generiramo nov žeton ter ob vsakem zahtevku v vmesnem sloju (angl. Middleware) preverjamo ali ima zahtevek veljaven žeton ali ne.

#### Expo-server-sdk

Naša aplikacija tudi pošilja obvestila uporabniku, kadar ima v svojem virtualnem vrtu rastline, katere mora še zaliti. Za to smo na zaledni strani uporabili NodeJS knjižnico »expo-server-sdk« s pomočjo katere od uporabnikove naprave pridobimo »ExpoPush« žeton, katerega shranimo v bazo in ga potem uporabljamo, da pošiljamo zahtevke na Expo strežnike, ki potem preko Apple-ovih ter Firebase-ovih strežnikov pošiljajo obvestila na Apple ter Android naprave.

## Mobilna aplikacija

Mobilna aplikacija je bila ustvarjena s pomočjo ogrodja React Native in Expo. Expo nam omogoča emulacijo aplikacije v ExpoGo aplikaciji, ki si jo naložimo na telefon. Uporabili smo tudi različne knjižnice, ki olajšajo implementacijo in uporabo aplikacije, kot je axios za pošiljanje http(s) zahtev, firebase za avtentikacijo in uporabo Firebase Functions in expo-camera za uporabo kamere v aplikaciji.

### Glavne knjižnice

### Firebase

Firebase knjižnica med drugim omogoča enostavno in varnejše vključevanje avtentifikacije v aplikacijo. Ponuja prijavo z različnimi načini, sami smo dodali možnost prijave z Googlovim računom in s kombinacijo elektronskega računa in gesla, ki se ga lahko obnovi. Omogoča nam tudi enostavno povezavo s Firebase Function-i, kjer gostimo naš napovedni model kot FaaS. Napovedni model lahko nato v aplikaciji preprosto kličemo kot httpsCallable funkcijo, ki nam v odgovoru vrne rezultate napovedi.

### Reat-native-calendars

React-native-calendars je knjižnica, ki omogoča prikaz koledarja z označenimi datumi in izbiro datumov. Tako lahko na pregleden način prikažemo, kdaj je bila rastlina zalita, kdaj ne in prihodnja priporočena zalivanja.

### Expo-camera in expo-image-picker

Expo-camera knjižnica nam omogoča zajem fotografij v realnem času in expo-image-picker, ki nam omogoča izbiro med že posnetimi fotografijami. Obe knjižnici smo uporabili za namene klasificiranja rastlin, ki so na sliki.

## Scraper vsebin

Tekom razvoja smo napisali tudi scraper-ja za pridobivanje informacij o vrstah rastlin. Izbrali smo spletno stran <https://www.picturethisai.com> , saj je imela vse podatke, ki smo jih želeli o rastlini beležiti in prikazovati v aplikaciji. Scraper pridobi podatke posamezne vrste rastline in nato shrani objekt te rastline s takšnimi podatki, kot jih hranimo v bazi. Na koncu izvozi polje objektov rastlin v json dokument, ki ga lahko preko mongo compass orodja enostavno uvozimo v našo MongoDB podatkovno bazo. Scraper je bil pripravljen v JavaScript s pomočjo tehnologij NodeJS, Puppeteer in ExpressJS.

## Firebase Functions

Za potrebe gostovanja klasifikacijskega modela smo uporabili Firebase Functions, ki nam omogoča enostavno definiranje funkcije, ki naj se izvede ob klicu. Funkcijo smo zapisali v programskem jeziku python, saj smo ga uporabili tudi za učenje klasifikacijskega modela. Definirali smo, kakšne parametre oziroma kakšno telo zahteve pričakujemo ob klicu in kaj naj se s telesom zahteve zgodi. Pričakujemo sliko v base64 formatu, ki ga nato dekodiramo, naložimo model iz datotek in izvedemo napoved vrste glede na novo sliko. Rezultate napovedi nato pošljemo kot telo odgovora.

# Izzivi, na katere smo naleteli tekom razvoja

## Nekompatibilnost knjižnic v pythonu

Tekom razvoja smo naleteli na nekompatibilnost nekaterih knjižnic. Nekompatibilna je bila knjižnica tensorflowjs, zaradi česar je bil potreben »downgrade« nekaterih drugih knjižnic – tensorflow, numpy, protobuf. Po spremembi verzij ostalih knjižnic je knjižnica tensorflowjs postala kompatibilna.

## Nekompatibilnost knjižnic v mobilni aplikaciji

Tekom razvoja smo naleteli na nekompatibilnost nekaterih knjižnic. Nekompatibilna je bila knjižnica @tensorflow/tfjs-react-native, ki je nujno potrebna za vgraditev že naučenega modela. Poskusili smo zmanjšati verzije preostalih knjižnic, s katerimi ni bila kompatibilna, vendar je to vodilo v izgubo funkcionalnosti knjižnic, ki smo jih nujno potrebovali. Zaradi tega že naučenega modela nismo uspeli vgraditi v aplikacijo. Sicer bi lahko model zgradili tudi v aplikaciji, ampak potem bi se model moral učiti v realnem času z viri, ki mu jih ponuja sama mobilna naprava, kar je precej slabše kot učenje modela na računalnikih, saj imajo mobilne naprave v splošnem nižje zmogljivosti kot računalniki. Hkrati sta učna in validacijska množica veliki (26,1Gb in 1,92Gb), za kar marsikdo nima prostora na mobilni napravi. Zato model gostimo kot Firebase Function oziroma Google Function, ki je dostopna preko http zahteve (FaaS - Functions as a Service).

## Augmented reality

Naša želja je bila tudi vključiti Augmented Reality v našo aplikacijo. Vendar je večina knjižnic za uporabo Augmented Reality skupaj z React Native in Expo zastarelih. Poskusili smo knjižnice:

1. @viro-community/react-viro,
2. Expo-three-ar,
3. Vuforia

Prva knjižnica je zastarela, saj podjetje, ki je ustvarilo to knjižnico, ne obstaja več. Druga knjižnica ni kompatibilna z android telefoni, zgolj z iOS. Hkrati ob implementaciji pride do napake, saj so se knjižnice precej posodobile. Expo-three-ar se navezuje na knjižnico @unimodules/react-native-adapter, ki pa je uradno zastarela. Vuforia pa zahteva ključ za njihov server API in ključ za dostop, kjer pa nikjer ni dokumentirano, kje in kako bi naj ta ključ pridobili.

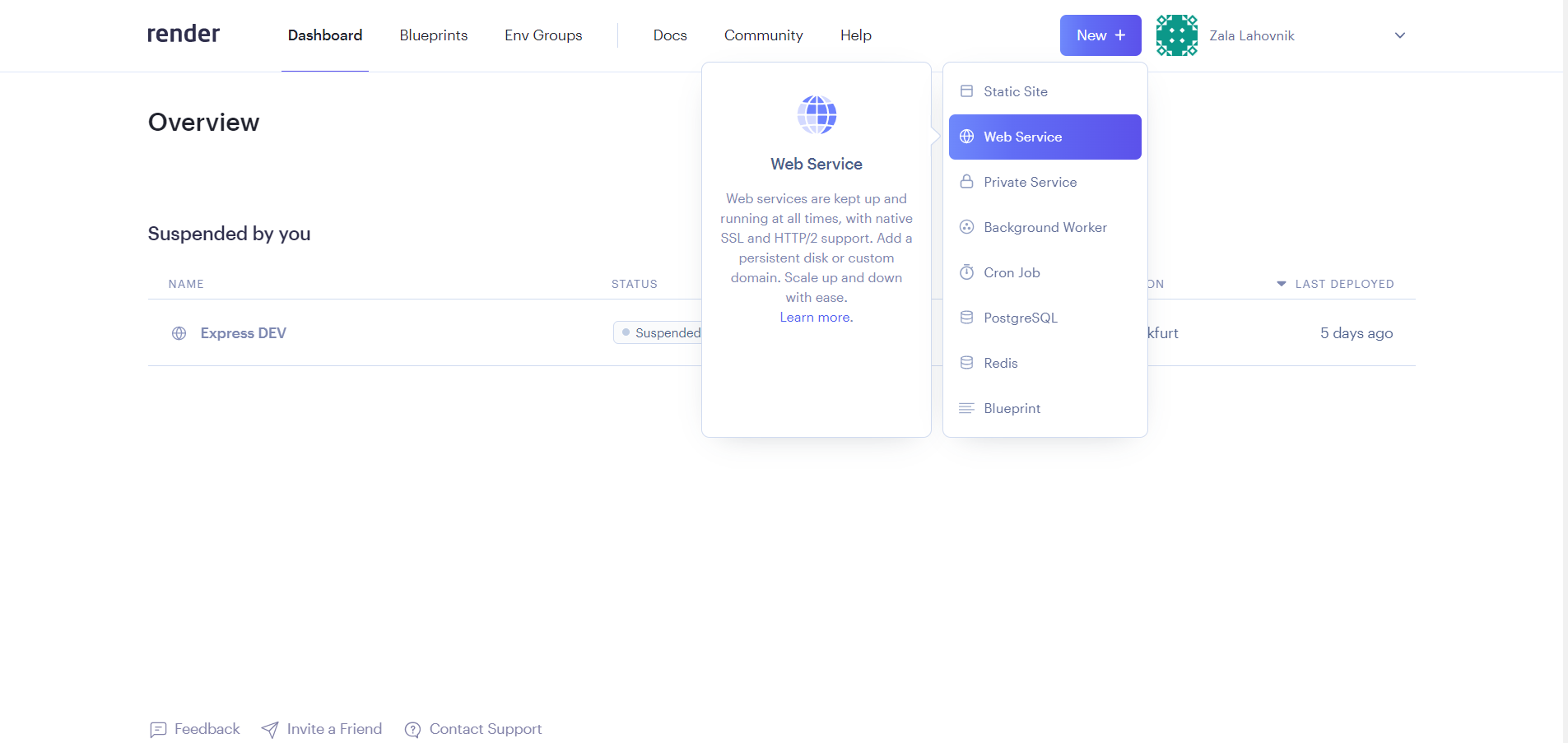
# Namestitev

## Zaledni sistem

Zaledni sistem gostimo pri ponudniku Render, saj omogoča brezplačno gostovanje Web Service-ov. Hkrati omogoča enostavno namestitev zalednega sistema s povezavo GitHub repozitorija in novo namestitvijo ob vsakem commit-u. Render omogoča tudi uporabo env datotek oziroma spremenljivk, kar smo tudi uporabili za povezavo na podatkovno bazo, da ni “hardcoded” v samo aplikacijo.

### Postopek namestitve

Na Render se prijavimo z GitHub računom, odpremo naš dashboard in nato izberemo “Web Service”.



Slika 22: Ponudbe Render-a

Nato nas Render preusmeri na naslednjo stran, kjer lahko izbiramo med našimi repozitoriji in javnimi repozitoriji. Izberemo repozitorij iz katerega želimo gostiti zaledni sistem.

A screenshot of a phone

Description automatically generated with low confidence

Slika 23: Izberemo željen repozitorij

Ko izberemo in povežemo repozitorij je potrebno izpolnit nekatere informacije, kot je ime veje iz katere želimo, da se zaledni sistem gradi, kateri direktorij je korenski direktorij našega zalednega sistema, s katero tehnologijo želimo naš zaledni sistem zgraditi. V našem primeru smo izbrali ime BotaniAI dev, regijo smo nastavili na Frankfurt (EU Central), dev branch za uporabo tekom razvoja, kasneje pa main branch, korenski direktorij Backend, izbrana Runtime tehnologija pa je bila Node.

A screenshot of a computer

Description automatically generated with medium confidence

Slika 24: Izpolnimo informacije novega Web Service-a

Za build command smo vpisali npm install, saj smo projekt ustvarili z npm in ne yarn. Naš zaledni sistem pa zaženemo z začetno komando node server.js.

A screenshot of a computer

Description automatically generated with low confidence

Slika 25: Izpolnimo še preostale informacije

Nato čisto na koncu kliknemo na gumb Advanced, kjer se nam prikažejo možnosti za dodajanje env spremenljivk. Tukaj imamo več izbir, lahko dodamo vsako spremenljivko posebej s klikom na gumb Add Environment Variable ali pa dodamo vse spremenljivke v datoteki .env s klikom na gumb Add Secret File. Ker imamo samo eno spremenljivko, smo izbrali prvo možnost.

A picture containing text, screenshot, font, line

Description automatically generated

Slika 26: Možnosti okoljskih spremenljivk

V spremenljivko vpišemo ključ, preko katerega pridobivamo vrednost, v našem primeru je ključ DATABASE\_URI, vrednost pa je povezava na podatkovno bazo.

A picture containing text, line, font, screenshot

Description automatically generated

Slika 27: Vnos okoljske spremenljivke

Nato izberemo še, ali želimo, da se zaledni sistem ponovno namesti ob vsakem novem commit-u na izbrano vejo. Ker smo izbrali dev vejo v našem repozitoriju, smo izbrali manualno namestitev, na main veji pa avtomatsko namestitev, saj ta ne bo imela tako pogostih commit-ov kot dev veja. Na koncu kliknemo na gumb Create Web Service in naš zaledni sistem se začne graditi in nameščati.

## Mobilna aplikacija

Mobilna aplikacija je implementirana s pomočjo orodja *Expo,* ki temelji na knjižnici *React Native,* *JavaScript* knjižnici za izdelavo uporabniških vmesnikov na mobilnih napravah z operacijskim sistemom *Android in IOS.*

### Lokalna uporaba

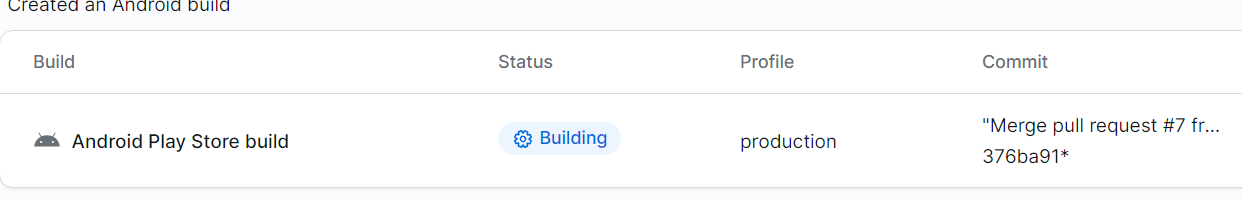
Lokalno lahko mobilno aplikacijo uporabljamo s pomočjo *ExpoGo* aplikacije, ki si jo naložimo na telefon preko *Google Play* ali *App Store*. Da lahko zaženemo in ustvarimo Expo aplikacijo je potrebno na napravo, na kateri bomo razvijali namestiti skupek orodij, ki jih predpisuje *Expo*. Za namestitev teh orodij je predpogoj nameščeno okolje *Node.js* za izvajanje *JavaScript* programske kode zunaj spletnih strani. Priporočljiva je različica vsaj 17, saj nam v nasprotnem primeru v naslednjem koraku ob uporabi katerega koli ukaza iz paketa *ExpoCli* ta javi, da uporablja zastarelo različico. Ob uspešni namestitvi *node* okolja lahko uporabimo storitev, za upravljanje s paketi *“npm”*. Ob zaključeni namestitvi lahko preverimo ali je bila le ta uspešna. To naredimo tako, da v terminalu izvedemo dva ukaza: *“npm -v” in “node -v”.* Namestitev je bila uspešna, če se nam v terminalu izpišeta obe različici, brez kakšnega opozorila ali napake. Ob predpostavki, da imamo uspešno nameščeno okolje n*ode* lahko v terminalu izvedemo ukaz “*npm install -g expo-cli",* s katerim na našo (lokalno) napravo namesti vsa potrebna orodja za uporabo *Expo* aplikacij. Nato aplikacijo zaženemo z ukazom “npx expo start” v korenskem direktoriju naše mobilne aplikacije. V našem primeru je korenski direktorij mobilne aplikacije imenovan *Frontend*. Nato preprosto v aplikaciji *ExpoGo*, v kolikor zaganjamo aplikacijo na android napravi, če pa zaganjamo aplikacijo na iOS napravi pa s kamero optično preberemo QR kodo, ki se nam prikaže v konzoli oziroma tam, kamor smo vpisali ukaz za zagon aplikacije, in aplikacija se nam bo naložila v emulator.

### Postopek namestitve

Končno aplikacijo si lahko bodisi namestimo na napravo bodisi jo naložimo na katerokoli od storitev *Google Play* ali *App Store.* Ker je aplikacija primarno narejena v JavaScript okolju jo je potrebno najprej pretvoriti v obliko, primerno za izvajanje na končni napravi, bodisi napravi z operacijskim sistemom Android, ki ga poganjata Java in pa Kotlin oziroma iOS, za katerim je programski jezik Swift. Pri tem je potrebno biti pozoren, da aplikacijo ne moremo pretvoriti, da bo kompatibilna z iOS, če naša lokalna naprava ni od proizvajalca Apple*,* oziroma ne poganjaMacOS operacijskega sistema*.*

Prvi korak v pretvarjanju *Expo* aplikacije v aplikacijo kompatibilno za mobilne naprave je izvedba ukaza  *“npx expo prebuild”,* ki je del paketa *ExpoCLI*, ki smo ga prej namestili. V tem koraku se v našem korenskem direktoriju ustvari datoteka *android,* oziroma *ios,* če smo ukaz pognali nanapravi z operacijskim sistemom *MacOS*. Obe datoteki služita kot osnova za naslednji korak, kjer bomo dejansko prevajali našo aplikacijo za ciljno platformo. Za namen prevajanja je potrebno na podoben način kot pri *ExpoCLI* namestiti storitev *EAS,* ki omogoča ustvarjanje binarnih izvršljivih datotek iz *React Native.* Namestitev opravimo z ukazom “*npm install -g eas-cli*". Ob uspešni namestitvi najprej zaženemo ukaz “*eas login*”, ki nas preusmeri na brskalnik, kjer se prijavimo z *expo* uporabniškim računom. Prijava je potrebna za namene povezave *expo.dev* lokalne platforme za spremljanje aplikacije z *eas* spletno storitvijo, ki naredi dejansko prevajanje aplikacije. Ob uspešem vpisu lahko v terminalu izvedemo ukaz “*eas build”,*  pri čemer moremo izbrati, za katero platformo želimo izvesti prevajanje. Prevajanje lahko izvedemo za platformo *Android (“eas build --platform android”), iOS (“eas build --platform ios”) ali pa za obe hkrati (“eas build --platform all”).* Korake prevajanja lahko spremljamo na platformi *expo.dev s povezavo* “*https://expo.dev/accounts/vašRačun/ projects/imeProjekta*”. Ime projekta lahko spreminjamo v datoteki *app.json*, tako da spremenimo *“name”,* pod razdelkom *expo.*

eas build -p android --profile preview



Slika : Expo.dev in stanje prevajane aplikacije

## Klasifikacijski model

Klasifikacijski model gostimo na Firebase Functions. Firebase Functions je FaaS – Functions as a Service, zaradi česar je naš model izpostavljen kot funkcija, dostopna preko http protokola.

### Postopek namestitve

Za namestitev je potrebno naš projekt inicializirati, kar smo naredili s pomočjo Nodejs in ukaza firebase init function v našem korenskem direktoriju. Pred inicializacijo ali tekom inicializacije se je potrebno prijaviti s firebase računom. V procesu inicializacije smo izbrali python programski jezik. V requirements.txt datoteko, ki se nam ustvari, zapišemo vse knjižnice, ki jih potrebujemo za model. Nato v main.py zapišemo kaj želimo, da se zgodi, ko je naša funkcija poklicana. V našem primeru želimo, da se model naloži iz datotek in nato klasificira rastlino na sliki, ki jo pridobi kot telo zahteve v base64 formatu. Kot zadnje zapišemo, kaj želimo, da se vrne kot odgovor na zahtevo. V našem primeru smo kot odgovor zapisali rezultat klasifikacije. Takšno funkcijo lahko nato uporabimo z emulatorjem ali pa jo namestimo na Google Cloud Functions. Za zagon emulatorja preprosto vpišemo ukaz firebase emulators:start, kjer se nam nato v konzoli izpiše naslov, kjer je naš emulator dostopen. Za namestitev Firebase Function-ov na Google Cloud Functions uporabimo ukaz firebase deploy --only functions Z ukazom –only povemo, da naj se namestijo samo Firebase Functions in ne vse ostale storitve, ki jih Firebase ponuja.

# Zagotavljanje kakovosti

Za zagotavljanje kakovosti smo uporabili SonarQube, saj omogoča statične preglede kode in podaja vpogled v morebitne bug-e, varnostno kritične dele kode, tehnični dolg in podvojene del kode. Za SonarQube smo se odločili tudi zaradi podpore pregledov TypeScript datotek in Python datotek.

## SonarQube

### Zaledni sistem

Ob prvem zagonu pregleda datotek zalednega sistema je bilo najdenih 10 varnostno kritičnih delov kode.

A screenshot of a computer

Description automatically generated

Slika : Prvi zagon SonarQube-a na kodi zalednega sistema

A screenshot of a computer

Description automatically generated with medium confidence

Slika : Prikaz enega izmed varnostno kritičnih delov kode

Po pregledu najdenih varnostih tveganj, smo v zalednem sistemu nastavili zgornjo mejo velikosti datotek, ki se lahko pošljejo na strežnik ter s tem odstranili vsa varnostna tveganja, ki jih je orodje v prvem zagonu prikazalo.

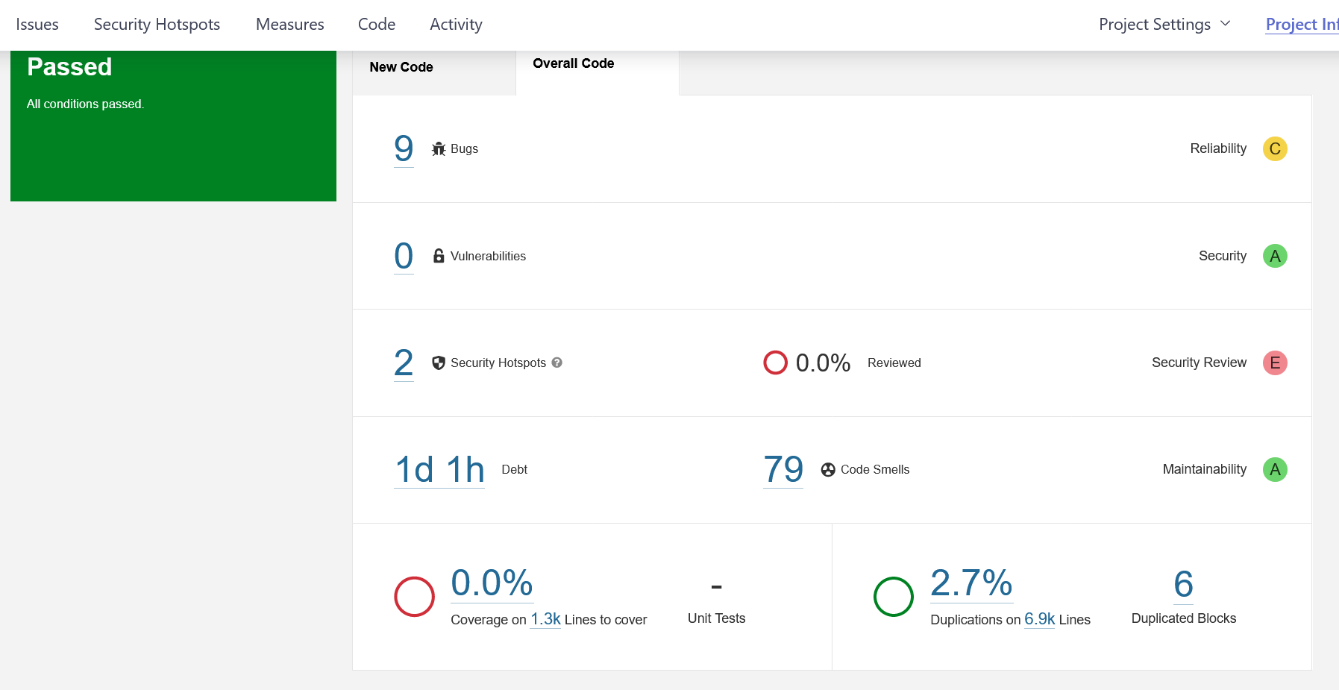
A screenshot of a computer

Description automatically generated with medium confidence

Slika : Zagon analize po spremembi zalednega sistema

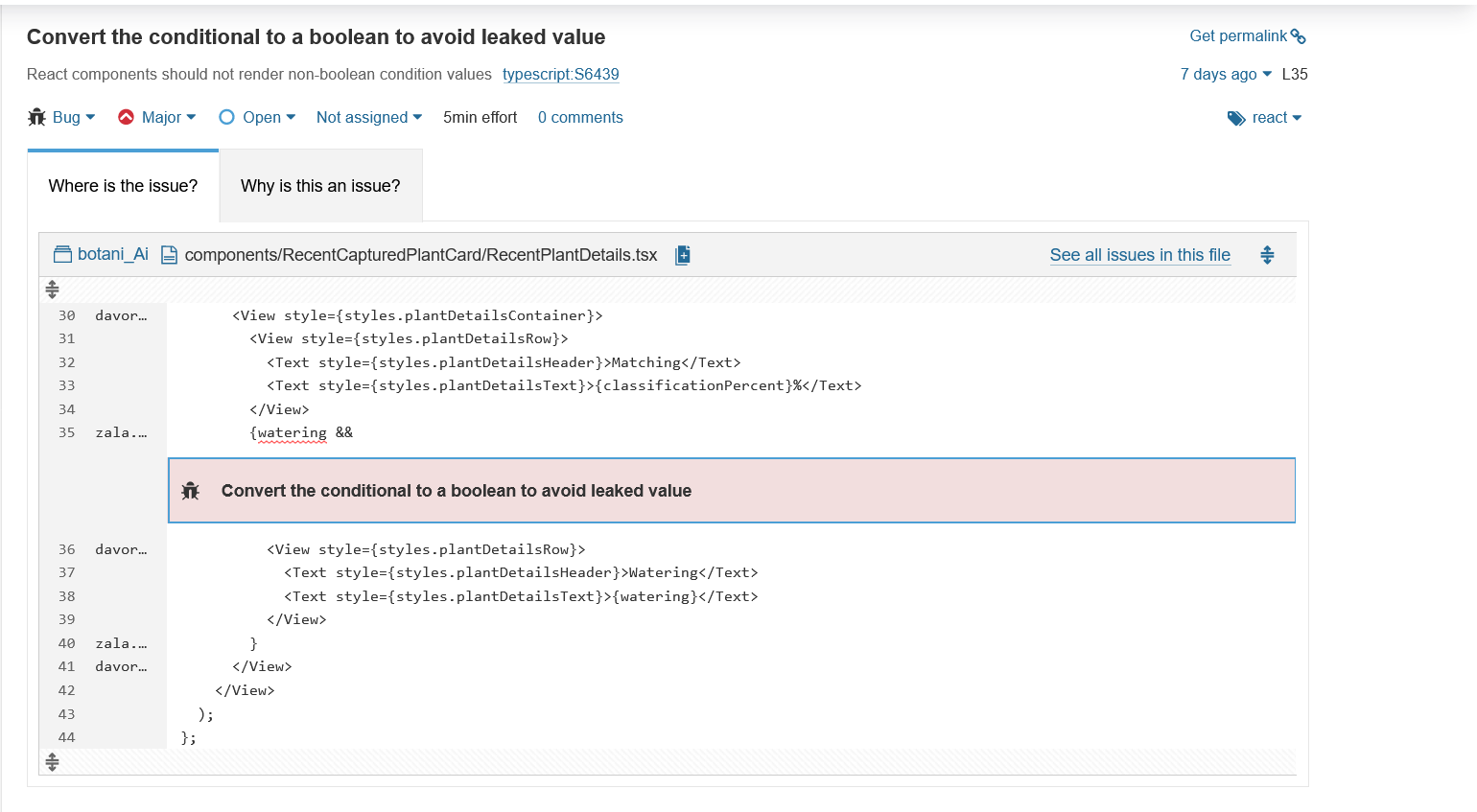
### Mobilna aplikacija

Ob prvem zagonu pregleda datotek mobilne aplikacije je bilo najdenih 79 možnih nepravilnosti v programski kodi, 2 varnostno kritična dela kode in 9 hroščev.



Slika : Prvi zagon SonarQube-a na kodi mobilne aplikacije

Po podrobni analizi rezultatov o možnih problemih v programski kodi, se je največkrat pojavil problem, kjer je program za analizo zaznal, da bi lahko na tej točki prišlo do *“puščanja vrednosti“.* V tem primeru gre za koncept imenovan *“Conditional Rendering”*, ki je del narave knjižnice React Native, ne pa kot možna napaka, saj se vrednosti tipa *string* in *array* v tem primeru uporabita kot *boolean* vrednost, *false*, če je seznam prazen oziroma je prazna vrednost *string* in *true* sicer.



Slika : Napaka možnega puščanja vrednosti

Druga izmed pogostih težav, je bila uporaba *index* vrednosti, ki je eden izmed argumentov metode *map,* ki se uporablja za enolično določanje ključa elementa.

Slika, ki vsebuje besede besedilo, posnetek zaslona, pisava

Opis je samodejno ustvarjen

Slika : Napaka določanja ključa elementa z vrednostjo iz funkcije map

Kritične napake, klasificirane s programom za analizo SonarQube smo odpravili tako, da smo se znebili vseh potencialnih nevarnosti, varnostno kritičnih točk in možnih hroščev. V kodi še ostaja 26 potencialnih opozoril, vendar so ti bolj kot problemi del konceptov React Native knjižnice.

Slika, ki vsebuje besede besedilo, posnetek zaslona, pisava, številka

Opis je samodejno ustvarjen

Slika : Končni rezultat analize s programom SonarQube

### Klasifikacija

Ob prvem zagonu pregleda python datotek je bilo najdenih 9 hroščev, 16 code smell-ov in skoraj 2h tehničnega dolga. SonarQube žal ne podpira pregledov .ipynb datotek, zato so datoteke bile pretvorjene v .py format, ki pa je nekoliko drugačen.

A screenshot of a computer

Description automatically generated

Slika : Prvi rezultati SonarQube za klasifikacijo

Ob pregledu nekaterih predlogov SonarQube-a smo ugotovili, da vsi niso smiselni. SonarQube je namreč želel, da izbrišemo zakomentirane dele kode, kar pa mi ne želimo. Nekateri deli kode se morejo zagnati zgolj enkrat na začetku in se takrat tudi odkomentirajo. Zaradi tega smo takšne predloge označili kot »Info« namesto »Major« in označili kot »False positive«, saj ne smatramo tega kot napake.

A screenshot of a computer

Description automatically generated with medium confidence

Slika : Predlog SonarQube za zakomentirano kodo

Po pregledu vseh predlogov, hroščev in code smell-ov, imamo kodo brez hroščev, code smell-ov in varnostno kritičnih delov kode.

A screenshot of a computer

Description automatically generated

Slika : Status klasifikacije SonarQube-a po popravkih kode

## Testi in CI/CD

### Testi

Za zagotavljanje kakovosti kode, oziroma, da ne pokvarimo delovanja funkcij v mobilni aplikacij ob dodajanju ali spreminjanju kode, smo za glavne (kritične) funkcije pripravili in ustvarili tudi teste. Za pisanje in poganjanje testov smo uporabili knjižnico Jest. Knjižnica Jest nam omogoča grajenje testnih suite-ov, kjer se nahajajo testi, ki testirajo enako funkcijo/funkcionalnost. Pripravljenih je bilo 9 testnih suite-ov, s 24 testi. Testirajo se predvsem funkcije kot so: pridobivanje naslednjih datumov zalivanja, pridobivanje časov zajema fotografij klasificiranih rastlin ali čez koliko dni je naslednji dan zalivanja.

### Avtomatizirano izvajanje testov

Za avtomatizirano izvajanje testov smo pripravili GitHub Action datoteko (workflow file). V njem smo zapisali ime avtomatiziranega izvajanja (*Frontend tests*), ob push-u na katere veje repozitorija naj se izvede Action (*main, dev, CI/CD*) in kaj naj se ob push-u zgodi. Zapisati je bilo potrebno izvajalno okolje, ki naj se uporabi, ukaz za nameščanje potrebnih knjižnic (*npm install* ali *npm i*) in ukaz za samo izvajanje testov (*npm run test*).

A screenshot of a computer program

Description automatically generated with medium confidence

Slika : Datoteka za avtomatizirano izvajanje testov