

|  |
| --- |
| Sabawoon Enayat  2022-2023 |

|  |
| --- |
| PlantPal |
| Plant Management Systeem |

# Het voorwoord

Ik ben Sabawoon Enayat en ik studeer Industriële ICT in Don Bosco Hoboken.

Mijn eindwerk is een plantmanagementsysteem. Ik heb dit onderwerp gekozen, omdat ik hou van planten en bloemen en als een programmeur wil ik altijd zo veel mogelijk automatiseren en op een makkelijke manier alles kunnen beheren.

Ik bedank Dhr. K. van Reck en Dhr. L. Sienack om me te begeleiden met het eindwerk zowel op technische vlak, maar ook met de inhoud.

Ik bedank ook de React.js, FastAPI, ESP32, Prisma.js, PostgreSQL, React-Native, Expo, en andere teams en communities om hun technologieën gratis aan te bieden.

Inhoudsopgave

[Het voorwoord 1](#_Toc135162378)

[Inleiding 3](#_Toc135162379)

[Hardware 4](#_Toc135162380)

[Schema 5](#_Toc135162381)

[Bibliotheken 6](#_Toc135162382)

[Werking 7](#_Toc135162383)

[Setup 7](#_Toc135162384)

[Database 8](#_Toc135162385)

[PostgreSQL 8](#_Toc135162386)

[ORM: Prisma 8](#_Toc135162387)

[Voorbeeld 8](#_Toc135162388)

[Modellen 9](#_Toc135162389)

[WebAPI (Application Programming Interface) 12](#_Toc135162390)

[Python 12](#_Toc135162391)

[FastAPI 12](#_Toc135162392)

[Authenticatie 12](#_Toc135162393)

[Veiligheid 13](#_Toc135162394)

[Voorbeeld 13](#_Toc135162395)

[Periodstamps 14](#_Toc135162396)

[Web Interface 16](#_Toc135162397)

[TypeScript 16](#_Toc135162398)

[React.js 16](#_Toc135162399)

[Concepten 17](#_Toc135162400)

[Voorbeeld 17](#_Toc135162401)

[Bootstrap 18](#_Toc135162402)

[Voorbeeld 18](#_Toc135162403)

[Data fetching 18](#_Toc135162404)

[Besluit 19](#_Toc135162405)

[Bibliografie 20](#_Toc135162406)

# Inleiding

Mijn eindwerk is een plantmanagementsysteem. Dit wil zeggen dat een gebruiker zich kan registreren op de centrale website. Hij kan planten linken aan zijn profiel. Zo kan hij ook instellen op welke tijd de planten bewaterd worden en of dat het automatisch moet gebeuren. Hij kan in het dashboard ook zien hoe droog dat de grond van de plant is.

Er kan ook een grafiek bekeken worden van hoeveel water een plant heeft gekregen. Zo kunnen ook planten vergeleken worden.

Voor mijn eindwerk maak ik gebruik van heel wat technologieën om alles voor de “end-user”, zo makkelijk en beveiligd mogelijk te maken. Dit gebeurt door de “userinterface” en de rest van de applicatie te scheiden.

Voor de UI (userinterface) heb ik React.js met TypeScript gebruikt. React.js heb ik gebruikt omdat het alles versimpelt voor de ontwikkelaar en omdat er heel veel documentatie errond is. TypeScript omdat het strikt is en omdat het veel problemen elimineert.

Voor de API (Application Programming Interface, wat ik bedoelde met de “rest van de applicatie”), heb ik FastAPI gebruikt. De authenticatie gebeurt door JWT-tokens.

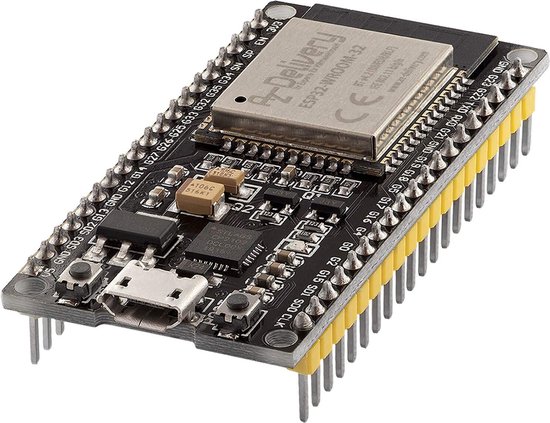
Ik heb PostgreSQL als database gebruikt, omdat ik ermee vertrouwd was en omdat er veel documentatie is.

Voor de ORM (Object Relation Mapping, de middleware tussen de API en de database), heb ik Prisma gebruikt, omdat het alles versimpelt.

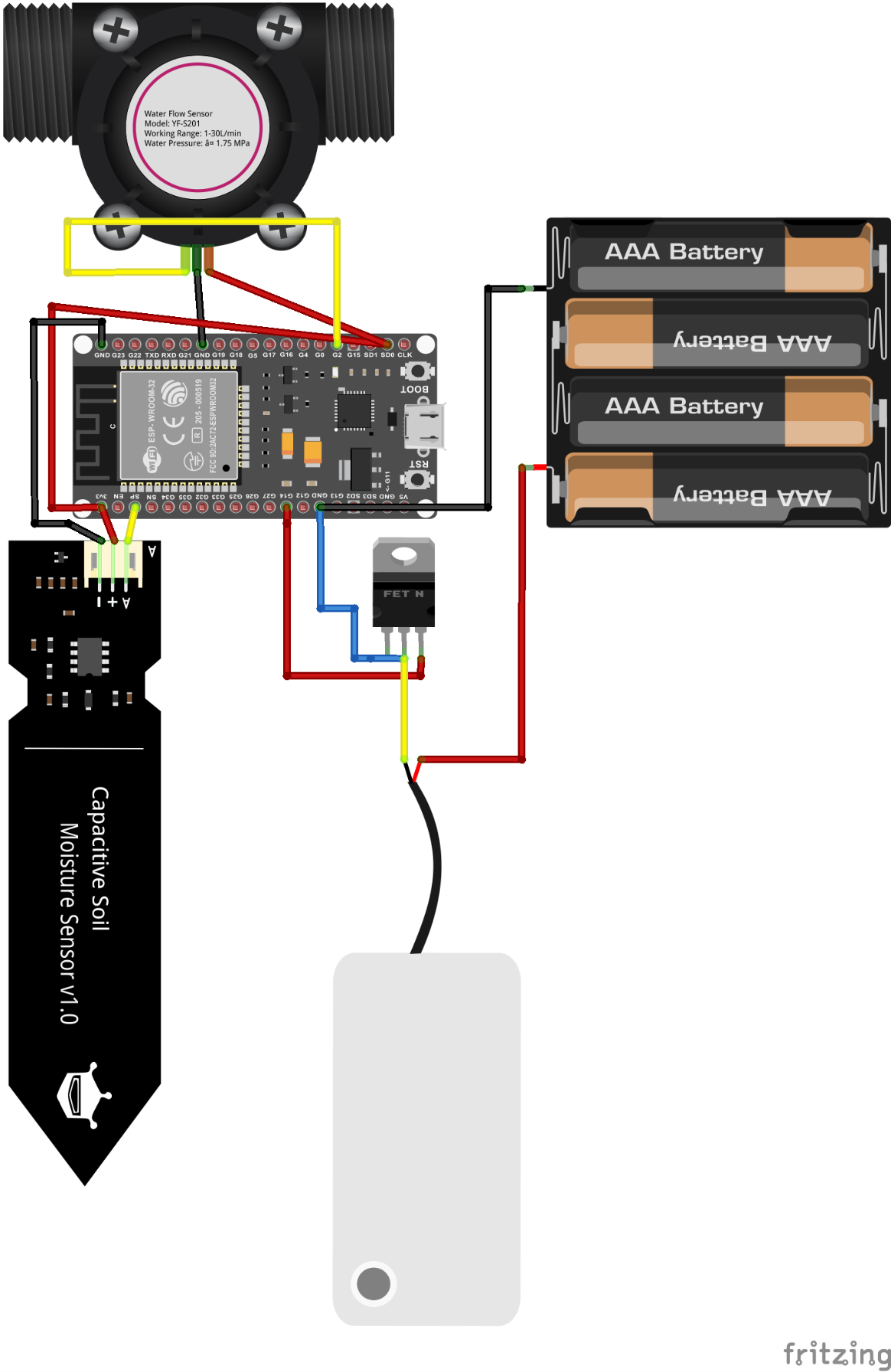
Voor de bewatering van de planten, heb ik een ESP32 gebruikt. De ESP32 heeft ook een ingebouwde Wifi module, die in AP (Access Point) mode, STA (Station) mode, of beide modes gecombineerd kan zijn.

In deze bundel maak ik gebruik van pseudo-code, welke dient om een concept te verduidelijken en waarbij irrelevante code is weggelaten voor de uitleg.

# Hardware

* ESP32
  + Dit is het belangrijkste onderdeel van het project. Het is een microcontroller die alle componenten bestuurt. Het heeft Wi-Fi en Bluetooth, en kan geprogrammeerd worden met Arduino, die gebaseerd is op C++, om de sensoren en de waterpomp aan te sturen op basis van wat de sensoren meten.
* 5V waterpomp
  + Dit is een elektrische pomp die water uit een reservoir of een waterbak naar de planten pompt. Het werkt op 5V stroom, meestal geleverd door de voedingsbron. De MOSFET wordt gebruik een MOSFET om de motor aan en uit te sturen, maar ook om pomp krachtiger te maken.
* Debietmeter
  + Dit is een sensor die meet hoeveel water er door de leiding stroomt. Ik installeer het in de leiding die water naar mijn planten levert, zodat ik kan zien hoeveel water er wordt gebruikt. Hiermee kan ik de duur en frequentie van het water geven aanpassen op basis van de metingen van de debietmeter.
* Grondvochtigheidssensor
  + Dit is een sensor die meet hoe vochtig de grond is rondom mijn planten. Ik stop het in de grond in de buurt van de wortels van de planten, zodat ik kan zien of de grond vochtig genoeg is voor de planten. Hiermee kan ik het irrigatiesysteem optimaliseren op basis van de metingen van de grondvochtigheidssensor.

## Schema



## Bibliotheken

* WiFi.h[[1]](#esp_wifi_tutorial)
  + Dit wordt gebruikt om de Wifi-functionaliteiten van de ESP32 te gebruiken. Om bijvoorbeeld een Netwerk-scan te doen, of de modus van de Wifi-module aan te passen (Station-Mode, Access-Point-Mode, STA-AP-Mode), verbinden met een netwerk, een soft-Access-Point maken enzovoort.
* WebServer.h
  + Dit is om een Webserver op te start op de ESP32. De Webserver kan dan bereikt worden door alle toestellen die in het netwerk van de ESP32 zitten. Het gebruikt de standaard HTTP-poort 80, dus een voorbeeld van een URL kan zijn `http://8.8.8.8:80/`, (de poort nummer is optioneel). De Webserver kan HTML of andere soort documenten sturen naar de Webbrowser, die formulieren bevatten. De data van het formulier kan naar een andere pagina via POST gestuurd worden, waar het gevalideerd wordt en we een “Response” kunnen krijgen van de Webserver.
* HTTPClient.h
  + Dit is gebruikt om te surfen naar een externe Webserver en zo alle HTTP-functionaliteiten te gebruiken. GET om data over de plant en bewateringen op te halen, POST om een plant aan te maken bij de database.
* Preferences.h
  + Dit is een bibliotheek die het makkelijk maakt om de NVS-flash van de ESP32 te gebruiken, om data op te slaan, zonder een zorg dat het kwijt gaat geraakt worden, na een heropstart van de ESP32. Dit maakt het mogelijk om data over de plant die we krijgen van de Webserver (de WebAPI gemaakt in Python) op te slaan voor latere gebruik.

Een belangrijk principe toegepast in dit project is een `Modulair Structuur`.

Modulaire Structuur (Single Reponsibility Principle)

* Elke functie of klasse is verantwoordelijk voor een enkele taak.
* Vergemakkelijkt onderhoud en hergebruik.
* Voorbeeld:
* Een file met de naam `plant.h`, heeft een `klasse` die uit functies bestaat. De functies kunnen worden opgeroepen, om de gegevens, zoals de ID, van de plant te lezen en schrijven met behulp van `Preferences.h`, maar ook om te checken of dat er water moet worden gegeven.

## Werking

### Setup

Bij het opstart wordt van de ESP32 wordt de WiFi-modus gezet naar `WIFI\_AP\_STA`, dit is de modus Access-Point en Station samen. Zo kunnen we een `softAP` (Soft-Access-Point) configureren, zodat een toestel kan verbinden aan de WiFi-netwerk van de ESP32. Hier wordt ook een `Webserver` geconfigureerd die kan bereikt worden door een toestel die verbonden is aan de Wifi van de ESP32. De IP-adres van de ESP32 is vastgezet en is `8.8.8.8`. Er wordt ook gecontroleerd of de ESP32 al eerder geconfigureerd is door de gebruiker (hierover later). Als dat wel het geval is, gaat de ESP32 proberen te verbinden met de WiFi-netwerk die is gegeven door de gebruiker.

Dit ziet er als volgt in pseudo-code:

void setup() {

  WiFi.mode(WIFI\_AP\_STA);

  WiFi.config(INADDR\_NONE, INADDR\_NONE, INADDR\_NONE, INADDR\_NONE);

  WiFi.setHostname(HOSTNAME);

  WiFi.softAPConfig(local\_IP, gateway, subnet);

  WiFi.softAP("PlantPalConfig", "plantpal");

  // Local IP: 8.8.8.8

  Serial.println(WiFi.softAPIP());

  // Start lokale Webserver die bereikt kan worden door een toestel in het netwerk van de ESP32

  server.start();

  if (!credentials.wifiNotWritten()) {

    // Al eerder geconfigureerd door de gebruiker

  }

}

Dan is er de `main-loop`. De code hier draait in een oneindige loop en voert verschillende taken uit, zoals het controleren of het apparaat is verbonden met wifi, het verkrijgen van gegevens over de plant en irrigatie, het meten van vochtigheidsniveaus en het regelen van de waterpomp.

De code begint met het controleren of wifi is verbonden en als dat niet het geval is, probeert het apparaat opnieuw verbinding te maken. Als er binnen 30 seconden geen verbinding kan worden gemaakt, wordt het wifi-netwerk opnieuw ingesteld en wordt het apparaat opnieuw gestart.

Vervolgens haalt de code om de 5 minuten plantconfiguratiegegevens op en om de 30 seconden gegevens over irrigatie. Als de plant water nodig heeft, begint de waterpomp te werken en als deze voldoende water heeft gekregen, stopt de pomp.

Daarna wordt de vochtigheidsniveaus van de plant gecontroleerd en als de plant water nodig heeft, begint de waterpomp te werken. De code meet ook de stroomsnelheid van het water met behulp van een flowmeter en stopt de waterpomp als de juiste hoeveelheid water is geleverd.

# Mobiele Applicatie (voor configuratie toestel)

Ik heb een mobiele applicatie gemaakt om het toestel toe te voegen aan ons account.

Dit is door gebruik te maken van `React-Native` en `Expo`. React-Native is een bibliotheek voor Node.js (JavaScript) om mobiele applicaties te ontwerpen. Expo is een omgeving die het makkelijker maakt om met React-Native te werken, door bijvoorbeeld functionaliteiten te geven en een dienst aan te bieden om applicaties voor productie te bouwen op hun servers, zowel voor `Android` als `iOS`.

## Waarom?

Voor de volgende redenen wilde ik een mobiele applicatie gebruiken om het toestel te configureren:

* Gebruikersvriendelijker
  + Bij de configuratie-website, moet de gebruiker elke keer opnieuw een netwerk selecteren en elke keer opnieuw inloggen om het toestel toe te voegen aan de account.
  + Dit is vermeden door deze gegevens te onthouden bij de mobiele applicatie, zodat ze later voor andere toestellen kunnen worden gebruikt.
* Sneller
  + Een netwerk-scan is heel traag (duurt 5 à 10 seconden) en inefficiënt op een ESP32. Dus de netwerk scan wordt gedaan door de mobiele applicatie, zodat het sneller en efficiënter is.
  + Gegevens (voor login en netwerk toevoeging) moeten niet elke keer ingevuld worden, want ze worden opgeslagen om het later te gebruiken.

## Beschrijving

Bij eerste opstart wordt er gevraagd voor een login. Deze waarden worden geëncrypteerd en veilig opgeslagen. Dit is alleen maar de eerste keer, of na het uitloggen.

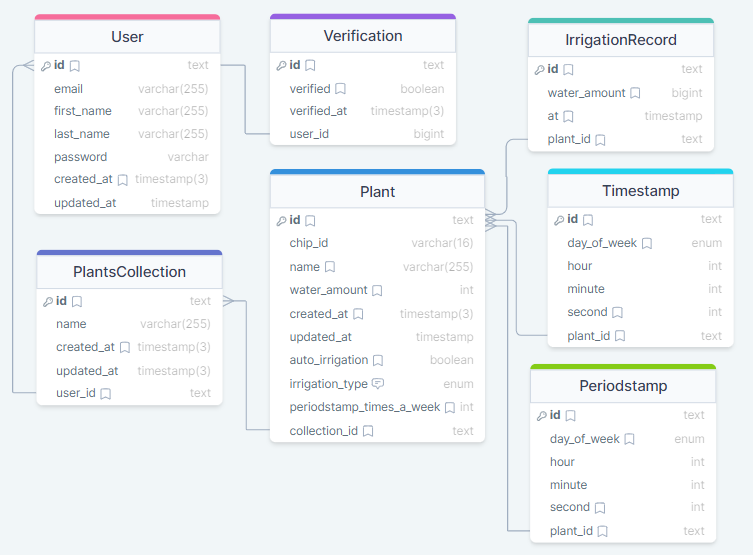
Er zijn twee tabs. De `Home` en `Settings`.

Settings heeft een rode knop om uit te loggen.

Home is waar een plant kan toegevoegd worden aan de account. Eerst is er een pagina, die controleert of dat het toestel bereikbaar is. Als het bereikbaar is er een knop om te connecteren. Na de connectie, verschijnt er een formulier om het toestel toe te voegen aan het netwerk (zodat het internet connectie heeft). Deze waarden kunnen opgeslagen worden, zodat de volgende keer de gegevens al verschijnen en je gewoon door kan gaan. Daarna verschijnt er een ander formulier, waar de naam en wachtwoord om het toestel te bereiken kunnen worden aangepast, zodat elke toestel uniek is. De standaardwaarde voor de naam is `plantpal-` gevolgd door de ID van de plant in de database.

## React-Native en Expo

# Database



## PostgreSQL

PostgreSQL is een krachtige database die ontwikkelaars kunnen gebruiken om hun applicaties te ondersteunen. Het is betrouwbaar en heeft veel handige functies, zoals ondersteuning voor complexe zoekopdrachten, ruimtelijke gegevens en tekst doorzoeken. PostgreSQL werkt op verschillende systemen en kan gebruikt worden met verschillende programmeertalen. Bovendien is het gratis en open-source, wat betekent dat iedereen het kan gebruiken en aanpassen. Dit maakt PostgreSQL een aantrekkelijke optie voor ontwikkelaars die op zoek zijn naar een krachtige, betrouwbare en flexibele database-oplossing.

## ORM: Prisma

Een ORM (Object-Relational Mapping) is een tool die ontwikkelaars kunnen gebruiken om te communiceren met hun database zonder SQL te schrijven. Prisma is een populair ORM dat werkt met Python en verschillende databases, waaronder PostgreSQL en MySQL. Prisma zorgt ervoor dat ontwikkelaars gemakkelijk objecten kunnen maken en ophalen vanuit hun database.

Je maakt eigenlijk eerst de nodige modellen in Prisma en laat Prisma de nodige Python classes en functies genereren die nodig zijn om interactie te doe met de database.

Migraties zijn ook ingebouwd in Prisma, wat ervoor zorgt dat als we de structuur aanpassen of een kolom willen toevoegen bij een tabel in de database, het heel makkelijk kan gebeuren.

### Voorbeeld

// Prisma Model

model BlogPost {

  id      String @id @default(cuid())

  title   String @unique @db.VarChar(255)

  content String @db.VarChar(255)

  createdAt DateTime @default(now())

}

# in de shell

# genereer python classes en methodes

prisma generate

# doe een migratie in de ontwikkeling-modus, met de naam “init”

prisma migrate dev –-name=“init”

# Python code om een nieuw blog-post te maken en de waarden te laten zien

import datetime

from prisma import Client

orm = Client()

new\_post = orm.blogpost.create({

    'title': 'New Post',

    'content': 'Hello there, we have a new blog post'

})

print(new\_post)

### Modellen

Dit is hoe de modellen voor dit project eruitzien in een `Prisma Schema`:

generator client {

  provider  = "prisma-client-py"

  interface = "asyncio"

}

datasource db {

  provider = "postgresql"

  url      = env("DATABASE\_URL")

}

model User {

  id           String        @id @default(cuid())

  email        String        @unique @db.VarChar(255)

  first\_name   String        @db.VarChar(255)

  last\_name    String        @db.VarChar(255)

  password     String        @db.VarChar(255)

  created\_at   DateTime      @default(now())

  updated\_at   DateTime      @updatedAt

  verification Verification?

  plants       Plant[]

}

model Verification {

  id          String   @id @default(cuid())

  verified    Boolean  @default(false)

  verified\_at DateTime @updatedAt

  user\_id String @unique

  user    User   @relation(*fields*: [user\_id], *references*: [id], *onDelete*: Cascade)

}

enum IrrigationType {

  time *// Times, e.g.: [13:15:00, 01:30:00]*

  period *// Amount per period, e.g.: 3 times per week*

}

model Plant {

  id                           String                     @id @default(cuid())

  chip\_id                      String                     @unique @db.VarChar(16)

  name                         String                     @default("New Plant") @db.VarChar(255)

*// waterAmount in milli-liter*

  water\_amount                 Int                        @default(1000)

  created\_at                   DateTime                   @default(now())

  updated\_at                   DateTime                   @updatedAt

  auto\_irrigation              Boolean                    @default(true)

  moisture\_percentage\_treshold Int                        @default(50)

  moisture\_percentage\_record   MoisturePercentageRecord[]

  irrigation\_type              IrrigationType             @default(period)

  irrigations\_record           IrrigationRecord[]

  timestamps                   Timestamp[]

  periodstamp\_times\_a\_week     Int                        @default(0)

  periodstamps                 Periodstamp[]

  user\_id String

  user    User   @relation(*fields*: [user\_id], *references*: [id], *onDelete*: Cascade)

}

model MoisturePercentageRecord {

  id         String   @id @default(cuid())

  percentage Int

  at         DateTime @default(now())

  plant\_id String

  plant    Plant  @relation(*fields*: [plant\_id], *references*: [id], *onDelete*: Cascade)

}

model IrrigationRecord {

  id           String   @id @default(cuid())

  water\_amount Int      @default(1000)

  at           DateTime @default(now())

  plant\_id String

  plant    Plant  @relation(*fields*: [plant\_id], *references*: [id], *onDelete*: Cascade)

}

enum DayOfWeek {

  monday

  tuesday

  wednesday

  thursday

  friday

  saturday

  sunday

  everyday

}

model Timestamp {

*// `Timestamps` duiden een tijd aan*

*// planten gaan geïrrigeerd worden op de geselecteerde tijden.*

  id          String    @id @default(cuid())

  day\_of\_week DayOfWeek @default(everyday)

  hour        Int

  minute      Int

  plant\_id String

  plant    Plant  @relation(*fields*: [plant\_id], *references*: [id])

}

model Periodstamp {

*// De gebruiker geeft het aantal irrigaties per week*

*// gebaseerd daarop gaat de `API`,*

*// de tijdstippen berekenen*

  id          String    @id @default(cuid())

  day\_of\_week DayOfWeek @default(everyday)

  hour        Int

  minute      Int

  plant\_id String

  plant    Plant  @relation(*fields*: [plant\_id], *references*: [id])

}

Waar nodig heb ik commentaar geschreven om het te verduidelijken.

# WebAPI (Application Programming Interface)

Dit is het gedeelte van de applicatie dat voor de authenticatie en beheer van data zorgt. Het is in essentie een interface tussen de database en de toepassing. De programmeertaal die ik gebruikt heb voor de WebAPI, is Python. De web framework die ik gebruikt heb is FastAPI.

## Python

Python heb ik voor de volgende redenen gebruikt:

* Eenvoudig gebruik
* Veel documentatie
* Heel stabiel

## FastAPI

FastAPI is een framework voor het bouwen van webapplicaties in de programmeertaal Python. Het heeft verschillende voordelen, waaronder:

1. Snelheid: FastAPI is erg snel en efficiënt, waardoor het ideaal is voor het bouwen van snelle en responsieve webapplicaties.
2. Eenvoudig te gebruiken: FastAPI heeft een eenvoudige en duidelijke documentatie, wat het gemakkelijk maakt om te leren en te gebruiken, zelfs als je geen ervaring hebt met webontwikkeling.
3. Asynchrone ondersteuning: FastAPI ondersteunt asynchrone programmering, wat betekent dat het meerdere taken tegelijk kan uitvoeren en zo de prestaties van de applicatie kan verbeteren.
4. Type hints: FastAPI maakt gebruik van type hints, waarmee ontwikkelaars het type van variabelen en functieparameters kunnen aangeven. Dit zorgt voor betere codekwaliteit en voorkomt fouten in de code.
5. Automatische documentatie: FastAPI genereert automatisch documentatie voor de API's die je bouwt, wat het gemakkelijker maakt om de API te begrijpen en te gebruiken voor anderen die ermee werken.

Kortom, FastAPI is een krachtig framework dat snel, eenvoudig te gebruiken, asynchroon en goed gedocumenteerd is. Het is daarom een goede keuze voor het bouwen van moderne webapplicaties met Python.

### Authenticatie

FastAPI JWT (JSON Web Token) authenticatie is een manier om gebruikers te authentiseren (inloggen) op een webapplicatie met behulp van JWT's. JWT's zijn stukjes informatie die worden gebruikt om de identiteit van een gebruiker te verifiëren.

De voordelen van FastAPI JWT authenticatie zijn onder andere:

1. Veiligheid: Met JWT's kun je veilig en gemakkelijk gebruikersgegevens versleutelen en overdragen tussen client en server, waardoor de veiligheid van de applicatie wordt verbeterd.
2. Eenvoudig te gebruiken: FastAPI biedt een eenvoudige manier om JWT-authenticatie in te stellen en te gebruiken, zelfs als je weinig ervaring hebt met authenticatie of beveiliging.
3. Flexibiliteit: Met JWT's kun je aangepaste claims (metadata) toevoegen aan de authenticatietokens, waardoor je extra informatie kunt opslaan over de gebruiker, zoals hun rol of toegangsrechten.
4. Efficiëntie: Met JWT's hoef je geen gebruikersgegevens op te slaan op de server, waardoor je server efficiënter wordt en minder resources nodig heeft.

Kortom, FastAPI JWT authenticatie biedt een veilige, eenvoudige en schaalbare manier om gebruikers te authentiseren en te autoriseren op een webapplicatie. Het is daarom een goede keuze voor het bouwen van moderne, veilige webapplicaties met Python.

### Veiligheid

FastAPI en Pydantic zijn frameworks die zijn ontworpen om veiligheid te waarborgen in webapplicaties. FastAPI biedt ingebouwde beveiligingsmaatregelen om bescherming te bieden tegen verschillende soorten aanvallen, terwijl Pydantic automatisch inputdata valideert om het risico op kwetsbaarheden te verminderen. Beide frameworks zijn ontworpen om te voldoen aan de nieuwste veiligheidsnormen en -praktijken, wat betekent dat het gebruik van deze frameworks samen zorgt voor een sterke beveiligingslaag in de webapplicatie.

### Voorbeeld

Dit komt uit de codebase van [PlantPal](https://github.com/nilerrors/PlantPal). Het is router die `Login`, `Signup` en `Account Verificatie` implementeert. Er is gebruik gemaakt van JWT-Tokens. Deze voorbeeld laat ook zien, hoe je een antwoord terugstuurt om te zeggen dat er iets mis is gegaan (bijvoorbeeld email en antwoord zijn fout).

from fastapi import APIRouter, Depends, HTTPException, status

from fastapi\_jwt\_auth import AuthJWT

from src.utils.send\_email import send\_email\_async

from . import crud, schemas

router = APIRouter(prefix='/auth')

@router.post('/login')

async def login(user: schemas.UserLogin, Authorize: AuthJWT = Depends()):

    auth\_user = await crud.authenticate\_user(user)

    if not auth\_user:

        raise HTTPException(status.HTTP\_404\_NOT\_FOUND, 'Email does not correspond with the password')

    db\_user = await crud.get\_user\_by\_email(user.email)

    if not db\_user or db\_user.verification is None or not db\_user.verification.verified:

        raise HTTPException(status.HTTP\_401\_UNAUTHORIZED, 'Account is not verified')

    access\_token = Authorize.create\_access\_token(subject=user.email, expires\_time=user.expires\_time)

    return {'access\_token': access\_token}

@router.post('/signup', status\_code=status.HTTP\_201\_CREATED)

async def signup(user: schemas.UserSignup):

    created\_user = await crud.create\_user(user)

    if created\_user is None:

        raise HTTPException(status.HTTP\_409\_CONFLICT, f"User with email-address '{user.email}' already exists.")

    await send\_email\_async(

        'Account Verification',

        user.email,

        {

            'title': 'Verify Account',

            'name': user.first\_name,

            'verification\_id': created\_user['verification\_id']

        },

        'verify.html'

    )

    return {'message': f'Verification email sent to {user.email}'}

@router.post('/user/verify/{verification\_id}')

async def verify\_user(verification\_id: str, Authorize: AuthJWT = Depends()):

    verification\_added = await crud.verify\_user(verification\_id)

    if verification\_added is None or verification\_added.user is None:

        raise HTTPException(status.HTTP\_404\_NOT\_FOUND, f"No account found with verification id '{verification\_id}'")

    access\_token = Authorize.create\_access\_token(subject=verification\_added.user.email, expires\_time=3600)

    return {'access\_token': access\_token}

### Periodstamps

Hieronder is de functie die de tijdstippen berekent, op basis van het aantal irrigaties per week, gegeven door de gebruiker:

async def change\_plant\_periodstamps(*user\_email*: str, *plant\_id*: str, *periodstamp*: schemas.PeriodStampsChange):

    plant = *await* get\_plant\_by\_id(user\_email, plant\_id)

*if* plant is None:

*return* None

    plant = *await* prisma.plant.update({

        'periodstamp\_times\_a\_week': periodstamp.times\_a\_week

    },

    {'id': plant.id})

*if* plant is None:

*return* None

*await* prisma.periodstamp.delete\_many(*where*={

        'plant\_id': plant.id

    })

*if* periodstamp.times\_a\_week == 0:

*return* 0

    WEEK\_IN\_MINUTES = 10\_080

    irrigation\_delay = WEEK\_IN\_MINUTES // periodstamp.times\_a\_week

    periods = [

        minutes\_to\_weektime(

            ((irrigation\_delay \* m) + random.randint(0, 10)) *if* irrigation\_delay \* m < WEEK\_IN\_MINUTES *else* WEEK\_IN\_MINUTES

        ) *for* m *in* range(periodstamp.times\_a\_week)

    ]

*return* *await* prisma.periodstamp.create\_many(*data*=[

        {

            'plant\_id': plant.id,

            'day\_of\_week': p.weekday,

            'hour': p.hour,

            'minute': p.minute

        }

*for* p *in* periods *if* p is not None

    ])

#### Uitleg

De functie gaat eerst controleren of dat het plant waarvoor de tijdstippen berekend moeten worden, bestaat. Zoniet dan gaan we niet verder (de rest wordt behandeld door de functie die deze functie oproept).

Dan worden alle bestaande tijdstippen (Periodstamps) voor deze plant verwijderd.

Dan wordt er gecontroleerd of de gebruiker meer dan 0 keer de plant wil bewateren per week. Zoniet dan wordt het aantal (dus 0) teruggestuurd.

Anders worden de tijdstippen berekend.

Een week bestaat uit 10.050 minuten. Dit wordt gedeeld door het aantal irrigaties dat de gebruiker wil. Zo krijgen we hoeveel minuten elke irrigatie van elkaar verschilt. Dit wordt voor elke tijdstip omgezet naar een structuur van (Dag, Uur, Minuut). Dit wordt dan in de database opgeslagen.

# Web Interface

Voor de web-interface heb ik, TypeScript, React.js en bootstrap gebruikt. Het platform waarop dit gemaakt wordt is Node.js.

De website heeft de volgende pagina’s:

* Login: Om zich aan te melden
* Signup: Om zich te registreren
* Verify: Om het account te verifiëren
* Plants: Om alle planten van de gebruiker te tonen
* Plant: om een specifieke plant te bekijken (er wordt verwezen naar dit pagina van Plants pagina)
* Timestamps: om de timestamps van een specifieke plant te zien en aanpassen
* Periodstamps: om de periodstamps van een specifieke plant te zien en aanpassen
* Settings: om de instellingen van een gebruiker aan te passen en account te verwijderen.
* Signout: om uit te loggen

## TypeScript

TypeScript is een programmeertaal die, net als JavaScript, gebruikt wordt voor het bouwen van websites en webapplicaties. Het grote voordeel van TypeScript ten opzichte van JavaScript is dat het een type-systeem heeft. Dit type-systeem maakt het mogelijk om fouten in de code te voorkomen voordat de code daadwerkelijk wordt uitgevoerd. Het type-systeem controleert namelijk of de verschillende delen van de code correct met elkaar zijn verbonden. Dit leidt tot meer betrouwbare code en minder problemen in het gebruik van de applicatie.

Een ander voordeel van TypeScript is dat het de programmeur helpt bij het schrijven van begrijpelijke en goed gedocumenteerde code. TypeScript maakt het namelijk mogelijk om documentatie te schrijven die de functies van verschillende delen van de code uitlegt. Dit zorgt ervoor dat andere programmeurs de code gemakkelijker kunnen begrijpen en dat de code gemakkelijker te onderhouden is.

Daarnaast biedt TypeScript nog andere handige functies, zoals bijvoorbeeld de mogelijkheid om klassen en interfaces te definiëren. Dit maakt het schrijven van complexere applicaties veel makkelijker en overzichtelijker.

Kortom, TypeScript biedt veel voordelen voor ontwikkelaars van websites en webapplicaties. Het zorgt voor meer betrouwbare en onderhoudbare code, waardoor het gemakkelijker wordt om samen te werken aan een project.

## File:React-icon.svg - Wikimedia CommonsReact.js

React.js is een populaire JavaScript-bibliotheek die wordt gebruikt om dynamische gebruikersinterfaces te bouwen voor webapplicaties. Het belangrijkste voordeel van React.js is dat het de ontwikkeling van complexe gebruikersinterfaces gemakkelijker maakt door het gebruik van componenten.

Componenten zijn kleine, herbruikbare stukjes code die specifieke functionaliteit bieden en kunnen worden gecombineerd om complexere interfaces te maken. Dit maakt het gemakkelijk om code te organiseren en te hergebruiken, wat tijd en moeite bespaart tijdens de ontwikkeling.

Een ander voordeel van React.js is dat het een efficiënte manier biedt om webpagina's dynamisch te updaten zonder de hele pagina opnieuw te laden. Dit zorgt voor een snellere en betere gebruikerservaring.

Ten slotte heeft React.js een grote en actieve community van ontwikkelaars die bijdragen aan de ontwikkeling van nieuwe functies, bugfixes en documentatie, waardoor het een betrouwbare en up-to-date bibliotheek is om te gebruiken in webontwikkelingsprojecten.

TSX (TypeScript XML) is een uitbreiding van TypeScript die wordt gebruikt in combinatie met React.js om JSX-syntax te ondersteunen. JSX staat voor JavaScript XML en is een syntaxisuitbreiding voor JavaScript die wordt gebruikt om React-componenten te maken.

Met JSX kan HTML-achtige code worden geschreven in JavaScript om gebruikersinterfaces te definiëren. Dit maakt het gemakkelijker om React-componenten te maken en de code leesbaarder te maken, omdat het lijkt op het schrijven van HTML-code.

### Concepten

1. Hooks: Hooks zijn functies in React.js die de mogelijkheid bieden om state en andere React-functies te gebruiken in functionele componenten. Hooks stellen ontwikkelaars in staat om complexe logica te schrijven in functionele componenten, wat het schrijven van code eenvoudiger maakt en de leesbaarheid verbetert.
2. Context: Context is een manier om gegevens door te geven tussen componenten in een React-applicatie zonder de gegevens expliciet te moeten doorgeven via props. Context kan worden gebruikt om thema's, gebruikersgegevens en andere gegevens te delen tussen componenten.
3. Props: Props is een afkorting voor "properties" en worden gebruikt om gegevens door te geven aan componenten. Props kunnen strings, objecten, functies en andere waarden bevatten en stellen componenten in staat om te communiceren met andere componenten in een React-applicatie.
4. State: State is een manier om gegevens op te slaan in een component. Wanneer de staat van een component verandert, wordt de component opnieuw gerenderd om de wijzigingen weer te geven. State kan worden gebruikt om gegevens op te slaan die veranderen tijdens de levensduur van een component.

### Voorbeeld

*import* React, { useState } *from* 'react';

function Example() {

  const [count, setCount] = useState(0);

*return* (

    <div>

      <p>You clicked {count} times</p>

      <button onClick={() => setCount(count + 1)}>

        Click me

      </button>

    </div>

  );

}

*export* *default* Example;

## File:Bootstrap logo.svg - Wikimedia CommonsBootstrap

Bootstrap is een styling bibliotheek, die vaak gebruikt wordt. Het zorgt voor de basis CSS-classes, zodat we de styling niet moeten schrijven, maar gewoon gebruiken. Er bestaan meer geavanceerde styling bibliotheken, zoals, mui, maar ik wou de styling simpel houden.

### Voorbeeld

Een simple knop:

Een voorbeeld van een knop in Bootstrap.

<button class="btn btn-primary">Primary</button>

De onderste CSS is voor ons al eerder geschreven. We kunnen het gewoon gebruiken in de HTML.

.btn-primary {

  background-color: #007bff;

  border-color: #007bff;

  color: #fff;

  padding: 0.5rem 1rem;

  border-radius: 0.25rem;

}

## Data fetching

Voor de data fetching gebruik ik de ingebouwde `fetch`-functie in JavaScript. Deze functie pakt een URL en een optionele “meta-data” object. Hieronder is een versimpelde toepassing die gebruikt is in de Webapplicatie:

// Een functie die alle planten van een gebruiker uit de server haalt

fetch("${API\_URL}/plants/", {

  method: "GET",

  // parameters voor authenticatie, ...

})

  .then((res) => {

    return res.json(); // converteer JSON `payload` naar een object of array

  })

  .then((data) => {

    // verwerk data

  })

  .catch((err) => {

    // behandel de error

  });

# Besluit

Ik heb zowel kunnen genieten van dit project als eruit leren. Ik zou hier verder willen aan werken om er een echt product van te maken.

Prioriteiten zijn:

* Compactere behuizing
* Mobiele applicatie uitbreiden
* Mogelijkheid voor collecties van planten
* Personalisatie:
  + Achtergrond afhankelijk van de voorkeur van gebruiker
  + Standaard data voor een plant aanpassen, zodat bij toevoegen van een nieuwe plant, deze waarden automatisch ingevuld worden.

# Bibliografie

[1] Santos, R., ESP32 Useful Wi-Fi Library Functions (Arduino IDE), internet, (<https://randomnerdtutorials.com/esp32-useful-wi-fi-functions-arduino/>), Geraadpleegd op 10 januari 2023.

[2]