

Berufliches Schulzentrum Arwed-Rossbach-Schule Leipzig

Berufliches Gymnasium

Fachrichtung Informations- und Kommunikationstechnologie

# Belegarbeit

im Fach Informatiksysteme

## Smart Gardening

-

Das Entwickeln einer Website mit Front/- und Backend zum Thema „Smart Gardening“

Verfasser: Reza Amiri

Kurs: A23 Deutsch Leistungskurs

Schuljahr: 24/25

Kurshalbjahr: 24/1

Außerschulischer Betreuer: Fabian Greger

Schulische Betreuerin: Fr. Schachoff

Ort, Datum: 10.01.2025

## Inhaltsangabe

Hinführung zum Thema .....	3
Zentrale Fragestellung/These.....	3
Aufbau der Arbeit .....	4
1.    Theoretischer Hintergrund .....	4
2.    Methodik .....	4
3.    Praktische Umsetzung .....	4
4.    Reflexion .....	4
5.    Fazit und Ausblick.....	4
Theoretischer Hintergrund .....	5
1.    Definition und Grundlagen .....	5
2.    Technische Grundlagen .....	5
Methodik.....	8
1.    Projektplanung und Vorgehensweise .....	8
2.    Ursprung der Daten und die Verarbeitung dessen .....	9
3.    Erklärung des Eigenanteils .....	10
Genutzte Werkzeuge .....	13
1.    Im Frontend genutzte Werkzeuge: .....	13
2.    Im Backend genutzte Werkzeuge:.....	13
Praktische Umsetzung mit Ergebnissen.....	13
1.    Frontend: .....	13
2.    Backend:.....	19
Reflexion.....	24
1.    Erfahrungen während der Arbeit .....	24
Bewertung der Ergebnisse .....	25
Kritische Betrachtung .....	25
Quellenverzeichnis.....	27
Selbstständigkeitserklärung .....	28

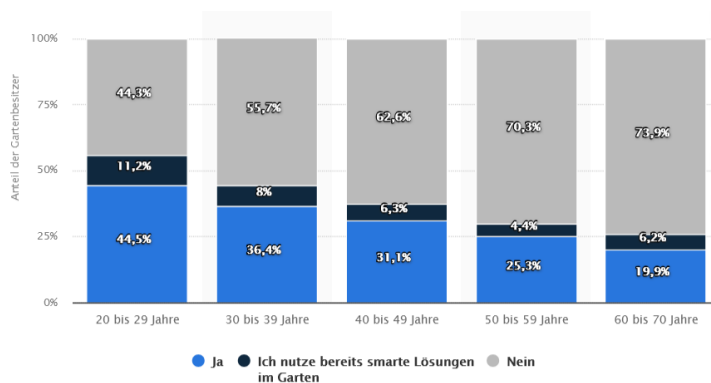
## Hinführung zum Thema

das Konzept des „smart gardening“ verbindet neue Technologien mit traditionellen Gartenarbeiten. Viele Menschen versuchen Pflanzen in ihrem Alltag zu integrieren (siehe Statistik), sei es Auf dem Balkon oder im Wohnzimmer, stoßen dabei jedoch auf einige Schwierigkeiten, wie zeitliche Einschränkungen oder einen zu hohen Pflegeaufwand. Genau in solchen Fällen, bietet das Prinzip „smartes Gärtnern“ eine Lösung. Mit der Entwicklung einer Website, die durch Echtzeit-Sensordaten und KI gestützte Analysen unterstützt wird, kann das Gärtnern flexibler, moderner und einfacher gestaltet werden.

Die Motivation dieses Projekt als mein Belegarbeits Thema zu wählen, ergibt sich aus der Interesse solcher komplexen Aufgaben, die ein großes Endziel ergeben, und zwar ein laufendes System zu haben und das Ziel digitale Werkzeuge zu nutzen, um alltägliche Probleme zu beseitigen, wie das pflegen einer oder mehrerer Pflanzen. „Smart gardening“ bietet nicht nur praktische Vorteile, sondern trägt auch dazu bei, nachhaltigere Entscheidungen zu treffen und die Pflege von Pflanzen zu optimieren. Außerdem gibt es auch viele Menschen, je jünger desto mehr, die sich eine smarte Garten-Lösung wünschen.



[2]



[1]

## Zentrale Fragestellung/These

Wie kann eine individuelle, flexible und kostengünstige Lösung für das „smart gardening“ entwickelt werden, die Echtzeit Sensordaten erfasst, mit künstlicher Intelligenz analysiert und den Nutzern auf einer Benutzeroberfläche anzeigt?

## **Aufbau der Arbeit**

### **1. Theoretischer Hintergrund**

Eine Einführung in die Grundlagen von Smart Gardening und die technischen Komponenten, die in diesem Projekt verwendet werden.

### **2. Methodik**

Beschreibung des Vorgehens bei der Planung und Umsetzung des Projekts, einschließlich der genutzten Datenquellen und Werkzeuge.

### **3. Praktische Umsetzung**

Ausführliche Beschreibung und das zeigen der Ergebnisse der entwickelten Website, einschließlich Frontend, Backend und interaktiver Funktionen wie dem laden der OpenAI Antwort und das Laden der Echtzeitdaten.

### **4. Reflexion**

Die Bewertung der erreichten Ziele und Reflexion der Herausforderungen und Vorschläge zur Weiterentwicklung.

### **5. Fazit und Ausblick**

Zusammenfassung der Ergebnisse und Ausblick für die Zukunft des Projekts für die Weiterentwicklung und die Veröffentlichung als Open-Source Projekt.

# Theoretischer Hintergrund

## 1. Definition und Grundlagen

„Smart gardening“ ist der Einsatz von neuen und modernen Technologien zur Automatisierung und Vereinfachung bzw. Optimierung der Gartenpflege. Sensoren erfassen wichtige Werte der Umgebung, wie Temperatur und Luftfeuchtigkeit, während Analysewerkzeuge und KI den Zustand der Pflanzen bewerten und Empfehlungen zur Pflege geben. Dadurch wird die Pflanzenpflege leichter, effizienter und weniger zeitaufwendig, was Menschen dazu bewegt sich Pflanzen zuzulegen und so Nachhaltiger zu leben, weil man sich zum Beispiel eigenes Gemüse anbauen kann. Hier einige Vorteile für den Selbstanbau:

- Keine Pestizide im Obst
- Günstiger als gekauftes Obst
- Erfüllende Gartenarbeit

[3]

Ein zentraler Aspekt des Smart Gardening ist die Integration von Technologien, die den Bedürfnissen der Pflanzen und den Anforderungen der Nutzer entsprechen. Hierbei spielen Flexibilität, Benutzerfreundlichkeit und Kosteneffizienz eine entscheidende Rolle, da sich Menschen, die Hobbygärtnern wollen, keine teuren Systeme kaufen wollen, um die Pflanzen zu betreuen.

## 2. Technische Grundlagen


- Entwicklung vom Frontend

Das Frontend bildet die Benutzeroberfläche der Website und wurde mit Mithril.js entwickelt. Mithril ist ein Framework. Dieses JavaScript Framework bietet eine schlanke und performante Alternative zu gängigen Frameworks wie React oder Vue, was besonders bei Projekten mit geringeren Ressourcenanforderungen von Vorteil ist.

[4]

Mithril implementierung Beispiel:

```
1 var root = document.body
2
3 * m.render(root, [
4 *   m("main", [
5     m("h1", {class: "title"}, "Mithril"),
6     m("button", "button"),
7   ])
8 ])
9
```

**Mithril**  


m.render(): Rendert in den DOM.

m(): Erzeugt virtuelle DOM-Knoten.

<main>, <h1>, <button>: UI-Elemente mit Text und Attributen.

[5]

- Entwicklung vom Backend

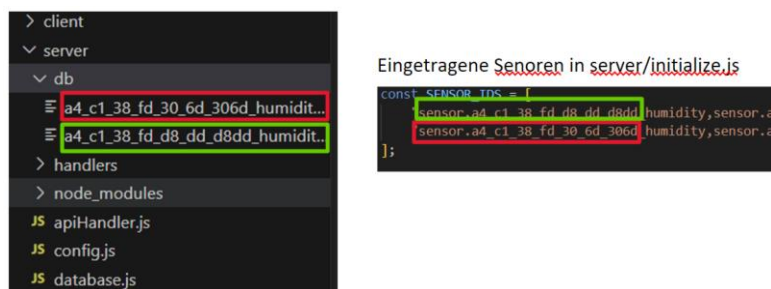
Im Backend wird die Datenverarbeitung gesteuert, wie zum Beispiel die API Aufrufe, Datenbankverwaltung und Datenbereitstellung für das Frontend. Node.js wurde aufgrund seiner Effizienz und Flexibilität für nodemailer und localhost gewählt [6] und mit einem Docker Server wird der server gehostet (auf einem Linux System) [7]. Das Backend wird von Fabian G. (Außerschulischer Betreuer) gehostet. Für das Projekt wurde eine Domain erworben: „smartgardening.rezaamiri.de“.[8]

- Datenbank (SQLite)

SQLite3 wird als lokale Datenbank verwendet auf dem Backend

→server/db/(hier werden die Datenbanken nach dem Serverstart instanziiert, jenachdem, wie viele Sensoren eingetragen sind.

2 Datenbanken hier im Beispiel:



Die Wahl fiel auf SQLite, da es eine einfache Implementierung ermöglicht und für die Datenmengen in diesem Projekt geeignet ist. Vorteile von SQLite:

- Keine zusätzliche Serverinstallation nötig.
- Ideal für unkomplizierte Datenstruktur Zeit-Wert Tabellen

Sensorname.db

Tempreture	
°id	integer
•timestamp	DateTime
°value	real

Humidity	
°id	integer
•timestamp	DateTime
°value	real

[9]

- OpenAI API

Die OpenAI API wird genutzt, um einen wöchentlichen Überblick bzw. Analyse der Sensordaten bereitzustellen. Die API liefert strukturierte JSON Ausgaben, die den Zustand der Pflanzen bewerten und Empfehlungen geben.



Hier ein Ausschnitt der Funktion „processWeeklyData“ in server/handlers/openaiHandler.js

```
const response = await openai.beta.chat.completions.parse({
  model: "gpt-4o-2024-08-06",
  messages: [
    { role: "system", content: `Du bist ein hilfreicher Assistent für die
Analyse von Sensordaten für folgende Pflanzen: ${plantSummary}. Die Pflanze/n befinden sich
in folgender Einrichtung: ${holdingConditions}. Die Antwort in die gegebene Struktur
ausgeben.` },
    { role: "user", content: prompt },
  ],
  response_format: zodResponseFormat(SummarySchema, "sensor_analysis"),
});
```

- Vergleich zu bestehenden smart gardening Systemen

Es gibt bestehende Systeme, die zu erwerben sind und sich mit den Funktionen von diesem Projekt ähneln.

Beispiel:



**Niwa Bundle (Grow Hub+ & CO2 Sensor)**

\$226.00

★★★★★ (1 customer review)

The Niwa Bundle comes with (1) Grow Hub, (1) Standard Sensor, and (1) CO2 Sensor. **For \$226**, get similar functionality to a commercial-grade controller, customize your environment, and monitor your garden from wherever you're located. You'll be able to control 4 pieces of equipment and **receive all of the Niwa readings** (VPD, temperature, relative humidity, light levels, dew point, and now CO2!)

1

ADD TO CART CHECKOUT

[10]

Dieser Anbieter bietet eine vergleichbare Lösung mit dem System von dieser Arbeit, jedoch sind die Kosten bei dem Anbieter sehr hoch im Gegensatz zu diesem Projekt. Außerdem gibt es bei „niwa“ keine KI Futures, die Analysen verfassen.

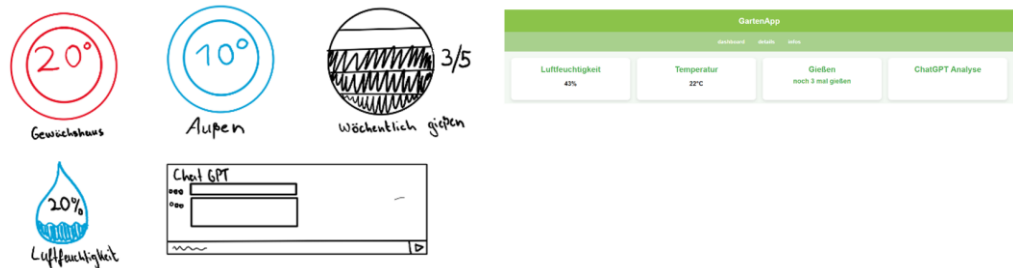
# Methodik

## 1. Projektplanung und Vorgehensweise

Die Entwicklung der Website „smartgardening.rezaamiri.de“ erfolgte in folgenden Schritten:

- Planung

Definition der Anforderungen: Integration von Sensordaten, Erstellung eines interaktiven Dashboards und Nutzung von KI Analysen in Form von Skizzen und Demos.



[11]

Auswahl der Technologien: Mithril.js für das Frontend, Node.js für das Backend und SQLite für die Datenbank. (siehe „Technische Grundlagen“).

Grund der Wahl von:

Mithril: in Mithril gab es schon Vorerfahrung wegen des Nebenjobs bei Tutorly.de

SQLite3: Bibliothek wird verwendet, weil der Außerschulische Betreuer „Fabian G.“ damit besser bekannt war und so die bessere Unterstützung geboten hat.

- Umsetzung (Implementierung)
  - Entwicklung und Testen der Backend API zur Verarbeitung und Speicherung der Sensordaten.
  - Mein Backend befindet sich in der Projektdatei im Verzeichnis: ./server
  - Erstellung des Frontends mit Fokus auf Funktionalität.
  - Das Frontend befindet sich im Client Verzeichnis: ./client
- Testphase
  - Überprüfung der korrekten Datenausgaben und Funktionalität der Website.
  - Vergleich der Sensordaten mit externen Wetterdiensten zur Validierung.



## 2. Ursprung der Daten und die Verarbeitung dessen

### Sensordaten

Die Sensordaten werden über die API von Home Assistant von Fabian G. abgerufen. Zwei Sensoren können angefragt werden:

- Außentemperatur und Luftfeuchtigkeit.
- Gewächshaustemperatur und Luftfeuchtigkeit.
- Bei der Initialisierung des Servers wird die Periode ausgelesen und für dieses Zeitraum die Daten in die Datenbank geladen.

```
async function initializeData() {
  const now = new Date();
  const periodEnd = now.toISOString();
  const periodStart = new Date(now.getTime() - 30 * 24 * 60 * 60 * 1000).toISOString(); // 30 Tage
  try {
    console.log("Datenbank leeren...");
    await clearDatabase();
    console.log("Datenbank erfolgreich geleert.");
    for (const sensorId of SENSOR_IDS) {
      console.log(`Lade Daten für Sensor: ${sensorId}`);
      const data = await fetchData(periodStart, periodEnd, sensorId);
      const formattedData = data.flat().map((entry) => ({
        entity_id: entry.entity_id,
        state: entry.state,
        last_changed: entry.last_changed,
        last_updated: entry.last_updated,
      }));
      console.log(`Speichere Daten für Sensor: ${sensorId}`);
      //console.log(formattedData);
      for (const entry of formattedData) {
        await insertSensorData(entry);
      }
    }
    console.log("Initialisierung abgeschlossen.");
  } catch (error) {
    console.error("Fehler bei der Initialisierung der Daten:", error);
  }
}
```

- Daten werden in eine Minute Intervallen geladen und an die Datenbank angehängt.  
Dabei handelt es sich um E  
eine Funktion, die sich selber jede Minute aufruft. (60 \* 1000 Millisekunden)

```
function scheduleDataFetch() {
  setInterval(async () => {
    const now = new Date();
    const periodStart = new Date(now.getTime() - 60 * 1000).toISOString(); // 1min
    const periodEnd = now.toISOString();
    //console.log("Sensor Ids on scheduleDataFetch: ", SENSOR_IDS);
    for (const sensorId of SENSOR_IDS) {
      try {
        //console.log(`Lade aktuelle Daten für Sensor: ${sensorId}`);
        const data = await fetchData(periodStart, periodEnd, sensorId);
        const formattedData = data.flat().map((entry) => ({
          entity_id: entry.entity_id,
          state: entry.state,
          last_changed: entry.last_changed,
          last_updated: entry.last_updated,
        }));
        console.log(`Speichere aktuelle Daten für Sensor: ${sensorId}`);
        console.log(transformSensorData);
        for (const entry of formattedData) {
          await insertSensorData(entry);
        }
      } catch (error) {
        console.error(`Fehler beim Abrufen aktueller Daten für Sensor ${sensorId}:`, error);
      }
    }
  }, 60 * 1000);
}
```

### Datenverkehr

- Die Sensordaten werden durch das Backend von der Home Assistant Schnittstelle geladen und gespeichert.
- Frontend greift durch API Endpunkt auf das Backend zu und bekommt als Antwort die „Rohdaten“ und verarbeitet bzw. visualisiert sie.

Eine API GET request, wo Daten vom Backend angefragt werden:

```
loadSettings: async () => {
  try {
    const data = await m.request({
      method: "GET",
      url: "http://localhost:3000/api/settings",
    });
    Dashboard.settings = data;
    Dashboard.location = data.location;
    Dashboard.remainingWatering = data.wateringFrequency || 3;
    Dashboard.alreadyWatered = data.alreadyWatered || 0;
  } catch (err) {
    console.error("Fehler beim Laden der Einstellungen:", err);
  }
},
```

In der folgenden asynchronen Funktion wird die json server/plant.json vom backend geladen und die einträge Standort (data.location), wassergießhäufigkeit (data.wateringFrequency) und schon gegossene Zahl (data.alreadyWatered) wird geladen, die dann in der Oberfläche verschiedene Nutzen haben.

die json.

Beispiel json im Backend:

```
{
  "plants": [
    {
      "name": "Sonnenblume",
      "number": 7
    },
    {
      "name": "Tomaten",
      "number": 20
    }
  ],
  "location": "Leipzig",
  "holdingConditions": "Gewächshaus",
  "wateringFrequency": 5,
  "gid": "10471",
  "alreadyWatered": 4
}
```

[12]

### 3. Erklärung des Eigenanteils

Der Eigenanteil in diesem Projekt umfasst die folgenden Bereiche im Projekt:

Backend Entwicklung:

server.js mit Routen:

- Die Implementierung der API Endpunkte für GET- und POST-Anfragen vom Frontend.
- Routen zum Abrufen und Speichern von Sensordaten sowie zur Integration der OpenAI API und auch für die plant.js (Daten, die zum Laden der Informationen für mehrere Bausteine gebraucht werden)

Integration von wetteronline.de iframe:

- Entwicklung einer Funktion zum Abrufen der gid (geografische ID, die von Wetteronline Intern benutzt wird) für die Standortanpassung.
- Nutzung der „gid“, um die Wettervorhersage dynamisch im iframe einzubinden, dass Nutzer den Standort in den Einstellungen ändern können.



- OpenAI Handler (server/handlers/openaiHandler.js):
  - Integration der OpenAI API für die Verarbeitung der Wochenstatistiken.
  - Erstellung strukturierter Prompts und Verarbeitung der API Antworten.

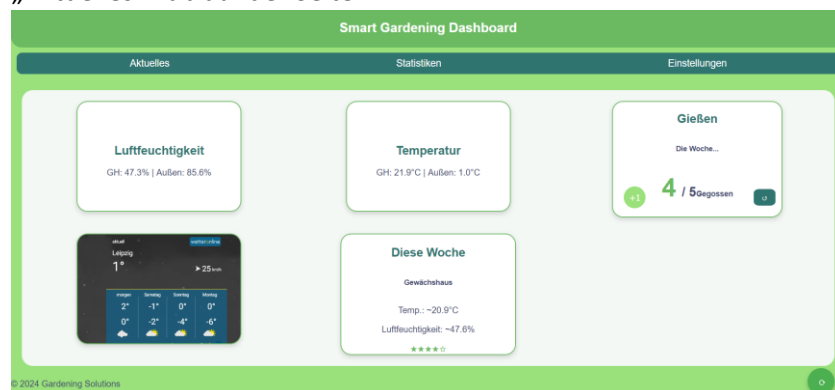
```
const response = await openai.beta.chat.completions.parse({
  model: "gpt-4o-2024-08-06", // Model für strukturierte Antworten
  messages: [
    { role: "system", content: `Du bist ein hilfreicher Assistent für die Analyse von
    Sensordaten für folgende Pflanzen: ${plantSummary}. Die Pflanze/n befinden sich in folgender Einrichtung:
    ${holdingConditions}. Die Antwort in die gegebene Struktur ausgeben.` },
    { role: "user", content: prompt },
  ],
  response_format: zodResponseFormat(SummarySchema, "sensor_analysis"),
});

const SummarySchema = z.object({
  average_temperature_außen: z.number(),
  average_humidity_außen: z.number(),
  average_temperature_gh: z.number(),
  average_humidity_gh: z.number(),
  trends: z.string(),
  evaluation: z.number(),
  recommendations: z.string(),
});
```

### Frontend Entwicklung:

- GET- und POST-Anfragen im Frontend:
  - Aufbau der Kommunikation mit dem Backend:
    - GET-Anfragen zur Anzeige von Sensordaten und Statistiken.
    - POST-Anfragen zur Aktualisierung des Gieß-Plans und für OpenAI-Analysen.
- Frontend Datenstruktur:
  - Implementierung des Frontends mit der Sortierung und Visualisierung der Daten.
  - Nutzung von Mithril.js zur effizienten Darstellung der Daten und Interaktion mit dem Nutzer.

„Aktuelles“ Tab auf der Seite



- Verarbeitung der Wochenstatistiken:
  - Entwicklung der details.js, die für die Aggregation (für Tag/Woche/Monat) und Sortierung der Rohdaten zuständig ist.
  - Übergabe der sortierten Wochenstatistiken an dashboard.js, um die OpenAI Analyse zu triggern und die Antwort anzuzeigen.

**Wochenstatistiken** → **dashboard.js** (führt POST durch und erwartet Antwort vom server) → **server.js** (ruft funktion auf, von...) → **openaiHandlers.js** (führt funktion für Api abfrage aus und gibt Antwort als Rückgabe wieder)

Nicht selbst umgesetzt:

- Komplexe Backend Funktionen:
  - Funktionen wie die Initialisierung der SQLite3 Datenbank und das Speichern der Sensordaten wurden in Zusammenarbeit mit meinem außerschulischen Begleiter Fabian G. entwickelt.
  - Diese Aufgaben waren aufgrund meiner bisherigen Erfahrung in der Softwareentwicklung schwer allein umzusetzen.
- CSS-Design und erste HTML Demos:
  - Die CSS-Elemente wurden größtenteils von GitHub Copilot generiert.
  - Erste Konzeptdemos wurden in HTML erstellt, um das Layout und die Struktur der Website zu visualisieren.
- Abgabe code kommentiert, wer was gemacht hat.

Außerschulische Betreuung „Fabian G.“:

- Fabian G. war ein wesentlicher Partner in diesem Projekt. Wir haben gemeinsam an der Planung und Implementierung gearbeitet.
- Er brachte fundiertes Wissen in Softwareentwicklung und Datenbankintegration ein, da er als Fachinformatiker ausgebildet ist, während ich den Fokus auf Frontend Entwicklung und API-Integration legte.
- Die Zusammenarbeit war geprägt von gegenseitiger Unterstützung und einem gemeinsamen Überblick über das gesamte Projekt.

## Genutzte Werkzeuge

### 1. Im Frontend genutzte Werkzeuge:

- Mithril.js: Framework, was genutzt wurde, um einfache API abfragen zu tätigen und leichter und strukturierter Elemente auf der Website hinzuzufügen, um eine dynamische Umgebung zu schaffen.
- Chart.js: ist für die Anzeige der Graphen bei den Statistiken zuständig und rendert diese.
- Vite: für Localhost während der Entwicklung, weil Änderungen auf dem Frontend direkt sichtbar sind im Browser... Es ist also kein Neustart des Localhosts nötig.

### 2. Im Backend genutzte Werkzeuge:

- Node.js: für der localhost bzw. die Server Initialisierung.
- Axios: für API abfragen im backend mit z.B Home Assistant.
- Datenbank: mit SQLite3 – in der server/initialize.js werden die Datenbanken initialisiert und von dort aus auch weitergeführt (minütlich neue Anhänge)

KI Analysen:

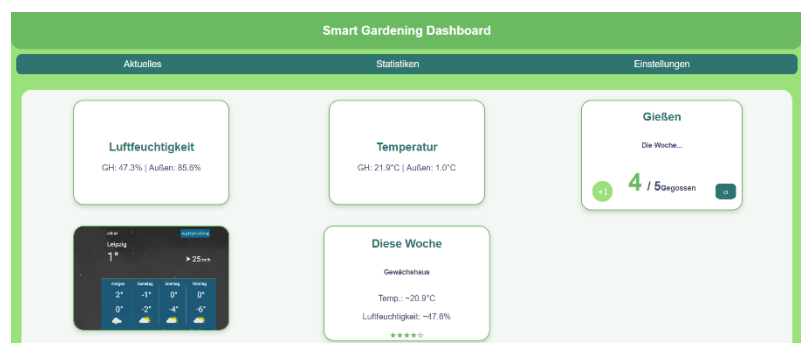
- OpenAI: OpenAI stellt eine breitgefächerte Bibliothek für Developer zur Verfügung.

## Praktische Umsetzung mit Ergebnissen

### 1. Frontend:

Datenstruktur:

```
client
├── Dashboard.js
├── Details.js
├── Settings.js
├── index.html
├── main.js
├── notification.js
├── package-lock.json
├── package.json
├── styles.css
└── vite.config.js
```



[13]

Das Dashboard (main.js) ist dreigeteilt in:

- Aktuelles (Dashboard.js)
- Statistiken (Details.js)
- Einstellungen (settings.js)

## Dashboard.js:

Oberfläche:

5 Bausteine zeigen jeweils:

1. Luftfeuchtigkeit im GH (Gewächshaus) und Außen
2. Temperatur im GH (Gewächshaus) und Außen
3. Gieß-Baustein: zeigt wie oft man schon gegossen hat und man kann mit dem Button +1 aufaddieren und mit dem button recht unten im Baustein kann man Gießvorgänge entfernen, wenn man zum Beispiel ausversehen +1 angeklickt hat.
4. Wettervorhersage: zeigt die aktuelle Wettervorhersage im gespeicherten Standort.
5. ChatGPT Analyse: zeigt eine kleine zusammenfassung von der Woche.

Umsetzung der Bausteine im einzelnen:

Luftfeuchtigkeit und Temperaturbaustein:

Dashboard.js ruft beim Aufruf folgende Funktion auf: „loadData()“

```
oninit: async () => {  
  await Dashboard.loadData(); // Allgemeine Daten laden --> echtzeitdaten  
  await Dashboard.loadSettings(); // location und gießparameter laden  
  await Dashboard.sendWeeklyDataToServer(); // Wochendaten senden und antwort erhalten  
},
```

loadData() führt eine GET request durch indem das Backend angefragt wird.

```
loadData: async () => {  
  try {  
    const result = await m.request({  
      method: "GET",  
      url: "http://localhost:3000/api/dashboard",  
    });  
    Dashboard.data = result.states.split(";").reduce((acc, entry) => {  
      const [key, value] = entry.split(":");  
      acc[key] = value;  
      return acc;  
    }, {});  
  } catch (err) {  
    console.error("Fehler beim Laden der Dashboard-Daten:", err);  
  }  
},
```

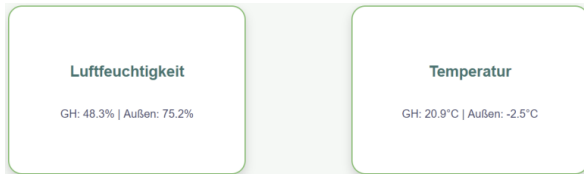
Als Antwort bekommt das Frontend einen Array der so aussieht:

[1, 85.6, 21.9, 47.3] --> [TemperaturAußen, LuftfeuchtigkeitAußen, TemperaturGH, LuftfeuchtigkeitGH]

Die Informationen werden dann mit Dashboard.data[x] ausgegeben. → schreibt dann den wert an stelle x im Array aus:

```
m("h2", "Luftfeuchtigkeit"),  
  m(  
    "p",  
    `GH: ${Dashboard.roundValue(Dashboard.data["4"])}% | Außen: ${Dashboard.roundValue(Dashboard.data["2"])}%`  
  ),  
),  
m("div.dashboard-card", [  
  m("h2", "Temperatur"),  
  m(  
    "p",  
    `GH: ${Dashboard.roundValue(Dashboard.data["3"])}°C | Außen: ${Dashboard.roundValue(Dashboard.data["1"])}°C`  
  ),  
]),
```

Auf der Website so:



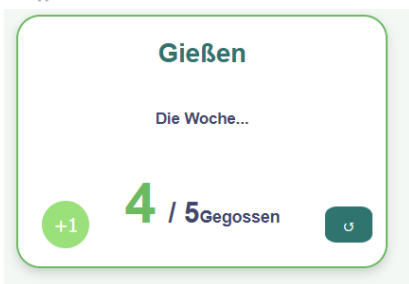
Gieß-Baustein:

Gieß-Baustein macht einen GET an dem server, jedesmal, wenn Dashboard.js aufgerufen wird, und einen POST wenn auf einem Button gedrückt wird:

```
saveSettings: async () => {
  const payload = {
    ...Dashboard.settings, // jetzige Einstellungen, sonst alles weg
    alreadyWatered: Dashboard.alreadyWatered, // neuer Wert
  };

  try {
    await m.request({
      method: "POST",
      url: "http://localhost:3000/api/settings",
      body: payload,
    });
    console.log("Einstellungen erfolgreich gespeichert:", payload);
  } catch (error) {
    console.error("Fehler beim Speichern der Einstellungen:", error);
  }
},
```

```
loadSettings: async () => {
  try {
    const data = await m.request({
      method: "GET",
      url: "http://localhost:3000/api/settings",
    });
    Dashboard.settings = data;
    Dashboard.location = data.location;
    Dashboard.remainingWatering = data.wateringFrequency || 3;
    Dashboard.alreadyWatered = data.alreadyWatered || 0;
  } catch (err) {
    console.error("Fehler beim Laden der Einstellungen:", err);
  }
},
```



Wettervorhersage:

wird ebenso eine GET ausgeführt, wo die location und gid dann im iframe eingefügt werden, um richtige Wettervorhersage zu laden:

```
m("div.dashboard-card.weather-card", [
  m("h2", "Wettervorhersage"),
  Dashboard.settings.gid
  ? m("iframe", {
    src: `https://api.wetteronline.de/wetterwidget?gid=${Dashboard.settings.gid}&modeid=FC2
&seoul=${Dashboard.location.toLowerCase()}&locationname=${Dashboard.location}&lang=de`,
  })
  : m("p", "Keine Wetterdaten verfügbar. Bitte überprüfen Sie den Standort."),
]),
```



ChatGPT Analyse:

Führt eine POST durch und sendet die Wochenstatistiken an das Backend und erwartet eine Antwort. (siehe 3. Erklärung des Eigenanteils --> Frontend Entwicklung: --> 3. Verarbeitung der Wochenstatistiken)

Die Antwort gibt es einmal in der kombinierten Version:

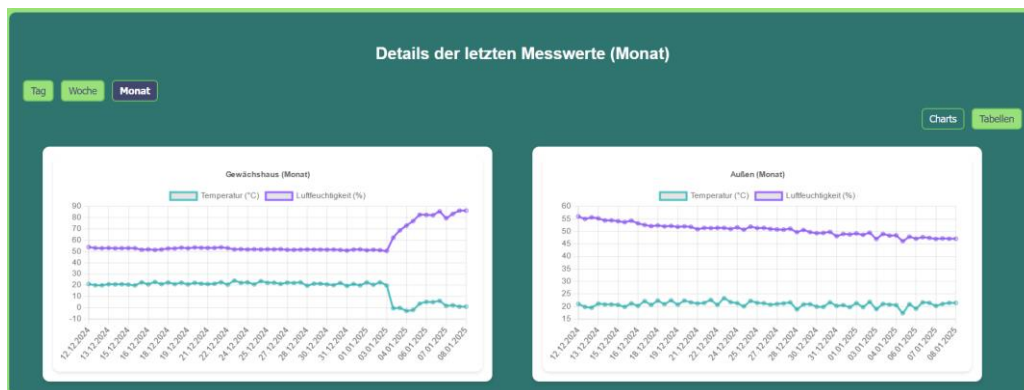


Und einmal in der erweiterten Version, wenn man mit dem Mauszeiger drüber gleitet:



## Details.js (Statistiken):

Bei den statistiken werden durch sortier funktionen die Daten strukturiert in charts angezeigt:



## Bespiel Wochenchart:

```
initialize: async () => {
  if (Details.isLoaded) return; // kann nicht mehrfach laden
  Details.isProcessing = true;
  try {
    const result = await m.request({
      method: "GET",
      url: "http://localhost:3000/api/data",
      headers: {
        "Content-Type": "application/json",
      },
    });
    Details.data = result;
    Details.aggregateAllData(); // Daten aggregieren --> dafür copilot verwendet (aggregateAllData())
    Details.isLoaded = true; // Markiere als geladen
  } catch (error) {
    console.error("Fehler beim Laden der Details-Daten:", error);
  } finally {
    Details.isProcessing = false; // Ladezustand deaktivieren
  }
},
```



Lädt die Sensordaten vom Backend mit GET Anfrage. Nach dem Laden werden die Daten mit `aggregateAllData` in tägliche, wöchentliche und monatliche Ansichten aggregiert und `isLoading` stellt sicher, dass die Daten nur einmal geladen werden.

```
// Aggregiere alle Daten vorab mit Hilfe von Copilot
aggregateAllData: () => {
  Details.agggregatedData.daily = Details.aggregateData("Tag");
  Details.agggregatedData.weekly = Details.aggregateData("Woche");
  Details.agggregatedData.monthly = Details.aggregateData("Monat");
},
// Funktion zur Aggregation von Daten
aggregateData: (view) => {
  const data = {};
  const startDate = new Date(view === "Tag" ? Details.selectedDate : new Date());
  if (view === "Woche") startDate.setDate(startDate.getDate() - 7);
  if (view === "Monat") startDate.setDate(startDate.getDate() - 30);
  Object.entries(Details.data).forEach(([dbName, entries]) => {
    const validEntries = entries.filter(entry => entry.timestamp && entry.temperature != null && entry.humidity != null);
    const groupedData = {};
    validEntries.forEach(entry => {
      const date = new Date(entry.timestamp);
      if (date >= startDate) {
        const key = view === "Tag"
          ? `${date.getFullYear()}-${date.getMonth() + 1}.toString().padStart(2, '0')}-${date.getDate().toString().padStart(2, '0')}`
          : `${date.getFullYear()}-${date.getMonth() + 1}.toString().padStart(2, '0')}-${date.getDate().toString().padStart(2, '0')}`;
        if (!groupedData[key]) {
          groupedData[key] = { temperature: 0, humidity: 0, count: 0 };
        }
        groupedData[key].temperature += entry.temperature;
        groupedData[key].humidity += entry.humidity;
        groupedData[key].count += 1;
      }
    });
    data[dbName] = Object.entries(groupedData).map(([key, values]) => ({
      timestamp: key,
      temperature: values.temperature / values.count,
      humidity: values.humidity / values.count
    }));
  });
  return data;
},
},
```

Die Funktion aggregiert die Rohdaten in Tag/Woche/Monat. Indem Fall wird sich die Wochenaggregation angeschaut:

Daten der letzten 7 Tage werden gefiltert. Durchschnittswerte für Temperatur und Luftfeuchtigkeit werden pro Tag berechnet und in Anschluss wird das Ergebnis in `Details.agggregatedData.weekly` gespeichert.

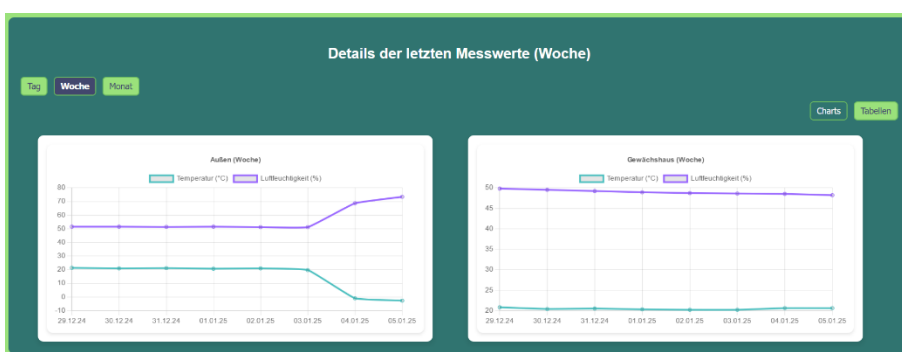
In der Ansicht wird dann erst überprüft welches der 3 Optionen (Tag/Woche/Monat) gewählt ist.

```
const currentData =
  Details.selectedView === "Tag"
    ? Details.agggregatedData.daily
    : Details.selectedView === "Woche"
      ? Details.agggregatedData.weekly
      : Details.selectedView === "Monat"
        ? Details.agggregatedData.monthly
        : {};
```

dann wird darauf basierend die dazugehörige Tabelle bzw graph angezeigt:

```
Details.selectedView === "Woche" && m("td", Details.formatWeekday(entry.timestamp)),
```

Dann werden die Graphen automatisch durch `chart.js/auto` geladen, weil man html elemente, wie `<td>` verwendet.



## Settings.js

Abschließend zum Frontend gibt es noch den Reiter Einstellungen, wo eine GET zum Zeitpunkt des aufrufs durchgeführt wird, um die gespeicherten Daten vom Backend zu laden. Mit dem button speichern kann man Änderungen speichern.

**Information**

Gießhäufigkeit (pro Woche)

- 5 +

**Pflanzen**

Sonnenblume	7	-
Tomaten	20	-

+ Neue Pflanze

**Haltungsform**

Bitte geben Sie an, wie Sie Ihre Pflanze halten und unter welchen Umständen.

Gewächshaus

**Wettervorhersage**

Standort: Hanover

Speichern Zurück zum Dashboard

Indem Fall sind 2 Funktionen für GET und POST zuständig, `saveSettings()` und `loadData()`:

```
saveSettings: async (vnode) => {
  const payload = {
    plants: vnode.state.settings.plants.map((plant) => ({
      name: plant.name.trim(),
      number: plant.number || 1,
    })),
    location: vnode.state.settings.location.trim(),
    holdingConditions:
      vnode.state.settings.holdingConditions.trim(),
    wateringFrequency:
      vnode.state.settings.wateringFrequency,
  };

  try {
    await m.request({
      method: "POST",
      url: "http://localhost:3000/api/settings",
      body: payload,
    });

    const savedPlants = payload.plants
      .map((p) => `${p.name} (${p.number} Stück)`)
      .join(", ");
    notifyAlert("GESPEICHERT!");
  } catch (error) {
    console.error("Fehler beim Speichern der Einstellungen:", error);
    notifyAlert("Fehler beim Speichern der Einstellungen. Bitte versuchen Sie es erneut.", "error");
  }
},

loadData: (vnode) => {
  m.request({
    method: "GET",
    url: "http://localhost:3000/api/settings",
  }).then((data) => {
    vnode.state.settings = {
      ...vnode.state.settings,
      ...data,
    };
  }).catch((error) => {
    console.error("Fehler beim Laden der Einstellungen:", error);
    notifyAlert("Fehler beim Laden der Einstellungen.", "error");
  });
},
```

## 2. Backend:

Das Backend erfüllt viele verschiedene Funktionalitäten. Dabei empfängt Server.js die API Anfragen vom Frontend und führt dann die jeweiligen Funktionen auf, um auf diese antworten zu können. (siehe WebApp/server/server.js)

Ordnerstruktur:

```
server
├── apiHandler.js
├── config.js
├── database.js
├── db
│   ├── Sensorname1.db
│   └── Sensorname2.db
├── handlers
│   └── openaiHandler.js
├── initialize.js
├── package-lock.json
├── package.json
├── plant.json
├── server.js
└── tables.js
```

Hier werden sehr viele komplexe Aufgaben durchgeführt, wovon ich zwei näher erläutern werde:

Wie wird eine OpenAI request an das Frontend versendet:

In der server.js gibt es die POST-Schnittstelle /api/chatgptDashboard:

```
// Verarbeiten von WeeklyData mit Fabian G.
app.post("/api/chatgptDashboard", async (req, res) => {
  try {
    const weeklyData = req.body.weeklyData;
    // schauen ob daten da sind
    if (!weeklyData || Object.keys(weeklyData).length === 0) {
      console.warn("Keine Wochendaten erhalten.");
      return res.status(400).json({ error: "Keine Wochendaten erhalten." });
    }
    console.log("WeeklyData erhalten:");
    Object.entries(weeklyData).forEach(([dbName, entries]) => {
      console.log(`Datenbank: ${dbName}`);
      entries.forEach((entry, index) => {
        console.log(
          `Eintrag ${index + 1}: Timestamp: ${entry.timestamp}, Temperatur: ${entry.temperature}, Luftfeuchtigkeit: ${entry.humidity}`
        );
      });
    });
    // zu openaiHandler.js
    const aiResponse = await processWeeklyData(weeklyData);
    console.log("OpenAI-Antwort server:", aiResponse);

    if (typeof aiResponse === "string") {
      return res.status(200).json({ parsedResponse: aiResponse });
    }
    const responsePayload = {
      average_temperature_außen: aiResponse.average_temperature_außen,
      average_humidity_außen: aiResponse.average_humidity_außen,
      average_temperature_gh: aiResponse.average_temperature_gh,
      average_humidity_gh: aiResponse.average_humidity_gh,
      trends: aiResponse.trends,
      evaluation: aiResponse.evaluation,
      recommendations: aiResponse.recommendations,
    };
    res.status(200).json(responsePayload);
  } catch (error) {
    console.error("Fehler bei der Verarbeitung der Wochendaten:", error);
    res.status(500).json({ error: "Interner Serverfehler. Bitte später erneut versuchen." });
  }
});
```

Vom Frontend wird weeklydata in einem json Format versendet.

Und es wird die Funktion processWeeklyData(weeklydata) mit den Wochenstatistiken als Übergabewert aufgerufen.

```
//mit copilot und Fabian G. --> Copilot hat nicht funktioniert aber struktur kommt davon...
async function processWeeklyData(weeklyData) {
  try {
    const { plantSummary, holdingConditions } = getPlantData();
    // dbname -> ort !muss noch dynamisch gemacht werden
    const dbToLocation = {
      "a4_c1_38_fd_30_6d_306d_humidity.db": "außen",
      "a4_c1_38_fd_d8_dd_d8dd_humidity.db": "gh",
    };

    const formattedData = Object.entries(weeklyData)
      .map(([dbName, entries]) => {
        const location = dbToLocation[dbName] || "unbekannt";
        const formattedEntries = entries.map(
          (entry) =>
            `Datum: ${entry.timestamp}, Temperatur: ${entry.temperature.toFixed(
              1
            )}°C, Luftfeuchtigkeit: ${entry.humidity.toFixed(1)}%`
        );
        return `Sensor (${location}): ${dbName}\n${formattedEntries.join("\n")}`;
      })
      .join("\n\n");
    const prompt = `
    Die folgenden Sensordaten repräsentieren eine Woche. Analysiere die Daten und liefere eine strukturierte
    JSON-Antwort mit folgenden Inhalten:
    - Durchschnittstemperatur und -luftfeuchtigkeit für 'außen' und 'gh'.
    - Bemerkenswerte Trends (1-2 Sätze, die alle Daten berücksichtigen).
    - Empfehlungen zur Verbesserung der Bedingungen, oder 'keine', falls keine Maßnahmen notwendig sind.
    - Eine Bewertung, wie es der Pflanze geht (Skala von 1-5).
    Datenbanken zu Orten:
    ${JSON.stringify(dbToLocation)}
    Daten:
    ${formattedData}
    `;
  } catch (error) {
    console.error("Fehler bei der Verarbeitung der Wochendaten:", error);
  }
}
```

Die weeklydata werden nun richtig formatiert, sodass sie im Prompt diese Struktur besitzen:

Daten:

Sensor (außen): a4\_c1\_38\_fd\_30\_6d\_306d\_humidity.db

Datum: 2024-12-26, Temperatur: 0.9°C, Luftfeuchtigkeit: 85.3%

Datum: 2024-12-27, Temperatur: 0.7°C, Luftfeuchtigkeit: 85.1%

...

Sensor (gh): a4\_c1\_38\_fd\_d8\_dd\_d8dd\_humidity.db

Datum: 2024-12-26, Temperatur: 21.8°C, Luftfeuchtigkeit: 47.2%

Datum: 2024-12-27, Temperatur: 21.3°C, Luftfeuchtigkeit: 46.8%

...

Die Funktion geht dann mit der API Abfrage an OpenAI weiter, wo beschrieben wird, wie die Antwort als eine Strukturierte json aussehen muss (siehe)

```
const response = await openai.beta.chat.completions.parse({
  model: "gpt-4o-2024-08-06", // Model für strukturierte Antworten
  messages: [
    { role: "system", content: `Du bist ein hilfreicher Assistent für die Analyse von Sensordaten für folgende Pflanzen: ${plantSummary}. Die Pflanze/n befinden sich in folgender Einrichtung: ${holdingConditions}. Die Antwort in die gegebene Struktur ausgeben.` },
    { role: "user", content: prompt },
  ],
  response_format: zodResponseFormat(SummarySchema, "sensor_analysis"),
});

const SummarySchema = z.object({
  average_temperature_außen: z.number(),
  average_humidity_außen: z.number(),
  average_temperature_gh: z.number(),
  average_humidity_gh: z.number(),
  trends: z.string(),
  evaluation: z.number(),
  recommendations: z.string(),
});
```

Die API Antwort schaut dann so aus:

```
{
  "average_temperature_außen": 0.8,
  "average_humidity_außen": 85.2,
  "average_temperature_gh": 21.55,
  "average_humidity_gh": 47,
  "trends": "Die Temperatur außen bleibt gering und relativ konstant, während die Luftfeuchtigkeit stabil hoch ist. Im Gewächshaus gibt es ebenfalls stabile Bedingungen mit angenehm warmen Temperaturen und moderater Luftfeuchtigkeit.",
  "evaluation": 4,
  "recommendations": "Keine besonderen Maßnahmen erforderlich, da die Bedingungen stabil und innerhalb der typischen Toleranzbereiche liegen."
}
```

Server.js sendet nun die json an das Frontend, wo die Json nun verarbeitet wird. (siehe loadSettings() im client/dashboard.js)

Wie werden die Echtzeitdaten an das Frontend versendet:

Im server.js gibt es folgende Schnittstelle, die vom Frontend aufgerufen werden kann:

```
// API-Anfrage für Live-Daten
app.get("/api/dashboard", async (req, res) => {
  try {
    const formattedStates = await fetchAllSensorStates(sensorIds);
    res.json({ states: formattedStates });
  } catch (error) {
    console.error("Fehler beim Abrufen der Sensorzustände:", error);
    res.status(500).json({ fehler: "Sensorzustände konnten nicht abgerufen werden." });
  }
});
```

Als Antwort ruft diese GET request eine Funktion namens fetchAllSensorStates() auf und bringt diesen in einen json-Format.

die Funktion fetchAllSensorStates befindet sich in apiHandler.js:

```
async function fetchAllSensorStates(sensorIds) {
  const results = [];
  for (let i = 0; i < sensorIds.length; i++) {
    const sensorId = sensorIds[i];
    try {
      const state = await fetchSensorState(sensorId);
      results.push(`${i + 1}:${state}`);
    } catch (error) {
      console.error(`Failed to fetch state for sensor ${sensorId}:`, error);
      results.push(`${i + 1}:null`);
    }
  }
  return results.join(';');
}
```

Diese Funktion sorgt dafür, dass jeweils jedes Sensor abgerufen wird und die state nacheinander in diesem Format: [state1 ; state2 ; state3 ; ...] zurückgibt.

fetchSensorState (nicht States) ist dafür zuständig für die übergebene sensorID eine API abfrage zu machen:

```
async function fetchSensorState(sensorId) {
  const sensorUrl = `${HA_Adress}/api/states/${sensorId}`;
  const headers = {
    Authorization: `Bearer ${AUTH_TOKEN}`,
  };
  try {
    const response = await axios.get(sensorUrl, { headers });
    const state = response.data?.state;
    if (state === undefined) {
      throw new Error('State not found in the response.');
```

dabei wird die sensorUrl die in diesem Format ist:

```
curl --location 'http://10.10.20.150:8123/api/states/sensor.a4_c1_38_fd_30_6d_306d_humidity' \
--header 'Authorization: Bearer Token'
```

Eine beispielabfrage kann man mit <https://web.postman.co/> durchführen, wo man dass eine json als Antwort bekommt:

```
{
  "entity_id": "sensor.a4_c1_38_fd_d8_dd_d8dd_humidity",
  "state": "47.35",
  "attributes": {
    "state_class": "measurement",
    "unit_of_measurement": "%",
    "device_class": "humidity",
    "friendly_name": "Amnesia1 Humidity"
  },
  "last_changed": "2024-12-25T12:26:38.033273+00:00",
  "last_reported": "2024-12-25T12:26:38.033273+00:00",
  "last_updated": "2024-12-25T12:26:38.033273+00:00",
  "context": {
    "id": "01JH70HXTFC9904405HVF45JR",
    "parent_id": null,
    "user_id": null
  }
}
```

Server.js erhält nun die Daten und sendet sie an das Frontend:

```
{"states":"1:0.15;2:84.58;3:21.59;4:47.38"}
```

(siehe Frontend Baustein 1 und 2 – Seite 14-15)

Mehr Ergebnisse bzw. die Website auf [smartgardening.rezaamiri.de](https://smartgardening.rezaamiri.de)

# Reflexion

## 1. Erfahrungen während der Arbeit

Die Umsetzung des Projekts „Smart Gardening“ brachte vieles mit sich:

- Integration von Sensoren und APIs:
  - Die Anbindung der Home Assistant API zur Sensordatenabfrage war technisch anspruchsvoll, insbesondere die Aggregation und Speicherung der Daten in einer SQLite-Datenbank, wobei Fabian G. eine sehr große Hilfe war.
  - Das Laden der Echtzeitdaten aus der Home Assistant API brachte einige Schwierigkeiten mit sich, sei es das Formatieren der Daten oder der API Aufruf an sich, trotz dessen konnte ich selbständig Lösungen finden.
- KI-Integration mit OpenAI:
  - Die Nutzung der OpenAI API für die Wochenanalyse spielte in dieser Arbeit eine zentrale Rolle. Die Erstellung strukturierter Prompts und die Verarbeitung der JSON-Antworten waren eine große Bereicherung für mein zukünftiges Arbeiten mit der OpenAI API in meinem Nebenjob bzw. im späteren Berufsleben.
  - Ein Nachteil dieser API-Schnittstelle sind die aufkommenden Kosten, die eine Einschränkung dargestellt haben, mehr Analysen von verschiedensten Statistiken zu machen oder fortlaufende Chats mit der KI zu führen.
- Zusammenarbeit mit Fabian G.:
  - Fabian G. war eine große Unterstützung, insbesondere bei komplexen Aufgaben wie der Datenbankinitialisierung und der Speicherung der Sensordaten. Die enge Zusammenarbeit hat zu einem besseren Verständnis für die Gesamtstruktur des Projekts geführt.



## **Bewertung der Ergebnisse**

### Erreichte Ziele:

Das Hauptziel wurde erreicht, eine Website zu entwickeln, die Sensordaten in Echtzeit verarbeitet und analysiert. Die OpenAI gestützte Analyse der Wochenstatistik bietet Nutzern, nützliche Einblicke und Empfehlungen zur Pflege ihrer Pflanzen.

### Benutzerfreundlichkeit:

Die übersichtliche Gestaltung des Dashboards und die interaktiven Elemente wie der Gieß Plan wurden positiv bewertet. Einige Betrachter wünschten sich jedoch visuelle Warnungen für zu hohe oder tiefe Werte und eine verbesserte Anzeige für Handys.

### Technische Stabilität:

Die Datenbank- und API-Integration funktioniert zuverlässig, und die Performance des Systems ist für den aktuellen Umfang zufriedenstellend.

## **Kritische Betrachtung**

### Stärken der Lösung:

Das System bietet eine klare und benutzerfreundliche Oberfläche mit Echtzeitdaten und Statistiken und Die OpenAI Analyse liefert strukturierte Informationen für die Nutzer.

### Schwächen und Grenzen:

Die Kosten der OpenAI API schränken die Nutzung ein, insbesondere bei häufigen Analysen. Die Lösung ist nicht vollständig skalierbar, da die Datenbankstruktur bei einer größeren Anzahl von Sensoren angepasst werden müsste.

UI ist verbesserungsfähig.

Die aktuelle Website hat einige Bugs, die auf dem Localhost nicht vorkamen. Diese führen zum falschen Laden der Daten bei den Statistiken.

Es muss auch zuerst die Statistik geöffnet werden und danach kann man erst auf Aktuelles die ChatGPT Analyse sehen, davor wird die Wochenstatistik nicht automatisch von Details.js geholt.

(auf dem Webserver von Fabian G. wird mithril.js nicht so wie auf localhost ausgeführt, was zu Problemen führt)

Der Server ist noch nicht für längere Zeit getestet worden, sodass keine Informationen vorhanden sind, wie der Server nach langer Laufzeit abschneidet.

#### zukünftige Optimierungen

- Backend Optimierungen:
  - Die Aggregation und das Sortieren der Rohdaten auf dem Backend durchführen und die nötigen bzw. fertigen Tabellen jeweils ans Frontend senden.
  - Einführung von Backups vom Server.
- Erweiterung der Funktionalität:
  - UI-Erweiterung (andere Farben und Bausteine anpassen, sowie in den anfangs Skizzen).
  - Integration weiterer Sensoren und Datentypen wie Lichtintensität oder Bodenfeuchtigkeit. (Die Sensoren sind vorhanden gewesen, jedoch war keine Zeit dafür gewesen)
- Kostenoptimierung:
  - Reduktion der OpenAI API Nutzung durch Vorgenerierung der Analysen im Backend, sodass Nutzer nicht jedesmal neue Analysen laden.

## Quellenverzeichnis

Gesamntes code sichtbar hier: <https://github.com/Komandods/Smart-Gardening>

[1]	<a href="https://de.statista.com/statistik/daten/studie/1400495/umfrage/gartenbesitzer-in-deutschland-nach-interesse-an-smart-gardening-nach-alter/">https://de.statista.com/statistik/daten/studie/1400495/umfrage/gartenbesitzer-in-deutschland-nach-interesse-an-smart-gardening-nach-alter/</a>
[2]	<a href="https://de.statista.com/infografik/21230/umfrage-zu-pflanzen-zuhause-und-im-garten/">https://de.statista.com/infografik/21230/umfrage-zu-pflanzen-zuhause-und-im-garten/</a>
[3]	<a href="https://www.proplanta.de/agrarnachrichten/verbraucher/obst--kaufen-oder-selber-anbauen-ein-vergleich_article1437419752.html">https://www.proplanta.de/agrarnachrichten/verbraucher/obst--kaufen-oder-selber-anbauen-ein-vergleich_article1437419752.html</a>
[4]	<a href="https://mithril.js.org/framework-comparison.html">https://mithril.js.org/framework-comparison.html</a>
[5]	<a href="https://mithril.js.org/index.html">https://mithril.js.org/index.html</a>
[6]	<a href="https://nodejs.org/docs/latest/api/synopsis.html">https://nodejs.org/docs/latest/api/synopsis.html</a>
[7]	<a href="https://docs.docker.com/">https://docs.docker.com/</a>
[8]	<a href="http://www.smartgardening.rezaamiri.de">www.smartgardening.rezaamiri.de</a> (durchaus verändert nach Abgabe)
[9]	Anhänge/Datenbankstruktur (die Visualisierung als Dia datei)
[10]	<a href="https://www.getniwa.com/">https://www.getniwa.com/</a>
[11]	Rechts. Mit copilot erstellte demo html
[12]	<a href="https://github.com/Komandods/Smart-Gardening">https://github.com/Komandods/Smart-Gardening</a> (server/plant.js)
[13]	Struktur mit <a href="https://marketplace.visualstudio.com/items?itemName=TinyMooshGamesInc.tree-exporter">https://marketplace.visualstudio.com/items?itemName=TinyMooshGamesInc.tree-exporter</a> ausgegeben

Zusätzliche quellen:

Auf [www.smartgardening.rezaamiri.de](http://www.smartgardening.rezaamiri.de) lädt Website (nicht im aktuellem Abgabestand – nur Veranschaulichung)

Auf der Git repository findet man das gesamte Projekt: <https://github.com/Komandods/Smart-Gardening>

Für die Codeausschnitte, um farbige Texte exportieren zu können: <https://marketplace.visualstudio.com/items?itemName=atian25.copy-syntax>

Benutzt, damit automatisch eingerückt wir: <https://marketplace.visualstudio.com/items?itemName=rvest.vs-code-prettier-eslint>

! Alle Quellen und Anhänge wurden am 05.01 zwischen 17:15 und 17:30 überprüft. !

## Selbstständigkeitserklärung

Hiermit versichere ich,

- dass ich die von mir vorgelegte Fach-/Belegarbeit **selbstständig** und ohne unzulässige fremde Hilfe verfasst habe,
- dass ich **keine anderen Hilfsmittel** als die im Vorfeld explizit erlaubten und von mir im **Hilfsmittelverzeichnis** vollständig dokumentierten verwendet habe (dazu gehören auch **KI-Werkzeuge**),
- dass ich **keine anderen Quellen** als die von mir im **Quellenverzeichnis** angegeben verwendet habe,
- dass ich alle **Stellen der Arbeit**, die ich wörtlich oder sinngemäß anderen Werken entnommen habe, als solche **unter Angabe der Quelle kenntlich gemacht habe** (dazu zählen auch Internetquellen und KI-Outputs),
- dass ich bei legitimer KI-Nutzung **KI-Outputs** auf Einhaltung **wissenschaftlicher Standards** geprüft und erforderlichenfalls überarbeitet habe.

Mir ist bewusst,

- dass die Fach-/Belegarbeit bei **Zweifeln an der Selbstständigkeit** zur Überprüfung der **Betreuerin/dem Betreuer** vorgelegt und durch ein **zusätzliches Fachgespräch** überprüft werden kann,
- dass ich im Falle eines **Täuschungsversuches** diese schriftliche Leistung nicht bestanden habe und
- dass ich bei Verwendung von KI-Werkzeugen die **Verantwortung für die KI-Outputs** trage, insbesondere hinsichtlich der inhaltlichen Richtigkeit und der Einhaltung von Datenschutz und Urheberrecht.

Ort, Datum: Torgau, 10.01.2024

Unterschrift:

