

Entwicklung einer smarten Bewässerungslösung mit Web-Anbindung

Studienarbeit

im Rahmen des
Bachelor of Science (B.Sc.)

des Studiengangs Informatik Cyber Security
der Dualen Hochschule Baden-Württemberg Mannheim

vorgelegt von

**Maximilian Schüller, Fynn Thierling, Justus Siegert,
Lukas Maier, Timon Kleinknecht**

15. April 2025

Erklärung der Eigenleistung

Hiermit erklären wir, dass wir die vorliegende Studienarbeit selbstständig und ohne fremde Hilfe verfasst haben. Wir haben keine anderen als die angegebenen Quellen und Hilfsmittel benutzt. Darüber hinaus erklären wir, dass im Rahmen des Schreibprozesses KI-gestützte Werkzeuge (ChatGPT) zur Umformulierung von Textstellen verwendet wurden. Wir bestätigen hiermit, dass alle verwendeten Quellenangaben korrekt sind und die inhaltliche Verantwortung für die Arbeit uneingeschränkt bei uns liegt.

Abstract

This study addresses the design and prototypical implementation of a smart irrigation solution for indoor environments. The objective of the work is to develop an automated system that accurately determines the water needs based on sensor data (e.g., soil moisture, temperature, humidity, and light intensity) and optimizes the watering process. As part of a proof-of-concept, the system integrates a microcontroller as the central control unit, a web-based backend infrastructure, and a user-friendly frontend for visualization and control. In addition to the technical implementation, the study explores modern IoT concepts, agile development methodologies, and the challenges of system integration. The proposed solution aims not only to achieve resource-efficient and demand-driven irrigation but also to contribute to sustainability and the enhancement of quality of life in urban residential settings.

Abstrakt

Diese Studienarbeit befasst sich mit der Konzeption und prototypischen Umsetzung einer smarten Bewässerungslösung für den privaten Innenbereich. Ziel der Arbeit ist es, ein automatisiertes System zu entwickeln, das anhand von Sensordaten (z.B. Bodenfeuchte, Temperatur, Luftfeuchtigkeit und Lichtintensität) den Wasserbedarf präzise ermittelt und den Gießvorgang optimiert. Im Rahmen eines Proof-of-Concept werden ein Mikrocontroller als zentrale Steuereinheit, eine webbasierte Backend-Infrastruktur sowie ein benutzerfreundliches Frontend zur Visualisierung und Steuerung integriert. Neben der technischen Realisierung werden dabei unter anderem moderne IoT-Konzepte, agile Entwicklungsmethoden und die Herausforderungen der Systemintegration beleuchtet. Die vorgestellte Lösung soll nicht nur eine ressourcenschonende und bedarfsgerechte Bewässerung ermöglichen, sondern auch einen Beitrag zur Nachhaltigkeit und zur Steigerung der Lebensqualität in urbanen Wohnumfeldern leisten.

Inhaltsverzeichnis

Abstract	ii
Abstrakt	iii
Abkürzungen	xii
Abbildungsverzeichnis	xiii
Listings	xiv
1 Einleitung	1
1.1 Motivation	1
1.2 Zielsetzung	4
2 Theoretische Grundlagen	6
2.1 Scrum und das Agile Manifest	6
2.1.1 Interpretation des Agilen Manifests in der Praxis	6
2.1.2 Zusammenhang zwischen Scrum und dem Agilen Manifest	7
2.1.3 Anwendung von Scrum im Sensorsa-Projekt	7
2.2 Single-page applications und Frameworks	8
2.2.1 Entstehung und Motivation von Single-page applications (SPAs)	9
2.2.2 Architektur, Vorteile und Herausforderungen von Single-page applications (SPAs)	9
2.2.3 Vergleich von React, Angular und Vue	10
2.3 Benutzerzentriertes Design und UI/UX im Frontend	10
2.3.1 User-Centered Design (UCD)	10
2.3.2 Usability-Heuristiken nach Nielsen	11
2.3.3 Gamification im UI/UX-Kontext	12
2.4 Theoretischer Vergleich: Webanwendungen versus native Apps	12
2.4.1 Architekturmodelle mobiler und Webanwendungen	12

2.4.2	Vergleich von Webanwendungen und nativen Android-Apps	13
2.4.3	Kombinationsstrategien	14
2.5	Softwaretestens mit Fokus auf Frontend-Frameworks	14
2.5.1	Ziele und Nutzen von Softwaretests	14
2.5.2	Testarten und deren Abgrenzung	15
2.5.3	Frontend-Testing mit Vue.js	15
2.5.4	Beispielhafte Unit-Test-Spezifikation mit Vue Test Utils	15
2.5.5	Beispielhafter Cypress-Test zur End-to-End-Verifikation	16
2.5.6	Codeabdeckung und Testmetriken	16
2.5.7	Einordnung und Fazit	17
2.6	REST APIs: Grundlagen und Best Practices	17
2.6.1	Warum REST?	17
2.6.2	Grundlagen einer REST API	18
2.6.3	Vertiefende Empfehlungen und Designansätze	19
2.6.4	Umsetzung für Sensora	19
2.7	Designprinzipien und -muster	20
2.7.1	Prinzipiengeleitetes Design	20
2.7.2	Coding-Guidlines	20
2.7.3	Event-Driven Architecture	21
2.7.4	Herausforderungen und Lösungen	21
2.7.5	Zusammenfassung	22
2.8	Solace PubSub+ Event Broker	22
2.8.1	Aufbau eines Message Brokers und Solace-Architektur	22
2.8.2	Topics und Queues in Solace PubSub+	24
2.8.3	MQTT und topic-basiertes Routing	25
2.8.4	Access Control Lists (ACLs)	26
2.8.5	Konfiguration über die SEMP-API	26
2.8.6	Vergleich mit anderen Messaging-Systemen	27
2.9	Sicherheitsrelevante Technologien im Auth-Service	28
2.9.1	HMAC: Hash-basierter Message Authentication Code	29
2.9.2	Fernet-Verschlüsselung: Authenticated Encryption für Tokens	31
2.9.3	Tokenisierung, Pre-Shared Key und Challenge-Response im Zusam- menspiel	33
2.10	Mikrocontroller und eingebettete Systeme	36
2.11	Grundlagen des Internet of Things (IoT) und Smart Home	41
2.11.1	Vernetzte Systeme und Sensorintegration	41
2.11.2	Kommunikationsprotokolle (MQTT, CoAP, HTTP)	41

2.11.3	Architekturmodelle (Edge, Cloud, Fog)	42
2.12	Regelungstechnik in Software	46
3	Anforderungen	50
3.1	Anforderungsdefinition für das Frontend	50
3.2	Anforderungen an die entwickelten Schnittstellen-Services im Projekt Sensorsa	51
3.2.1	Allgemeine Anforderungen an alle Schnittstellen-Services	52
3.2.2	Registrierungs- und Authentifizierungsservice (Auth-Service)	52
3.2.3	E-Mail-Verifikationsservice (Mail-Service)	53
3.2.4	Datenschreibdienst für Messdaten (Database Writer)	54
3.2.5	Service zur Steuerung von Zielwerten (Setpoint API)	55
3.2.6	Konfigurationsdienst für Kommunikationsinfrastruktur (Solace Init)	55
3.3	Anforderungen an das Backend	56
3.3.1	Architektur und Technologien	56
3.3.2	Authentifizierung und Autorisierung	56
3.3.3	Fehlerbehandlung und Logging	57
3.3.4	Skalierbarkeit und Performance	57
3.3.5	Sicherheit	57
3.3.6	API-Dokumentation	57
3.4	Anforderungen an die Datenbank	57
3.4.1	Modellierung und Struktur	58
3.4.2	Datensicherheit und Integrität	58
3.4.3	Zugriffskontrolle	58
3.4.4	Backups und Wiederherstellung	58
3.4.5	Performance und Skalierung	58
3.4.6	Technologischer Rahmen	59
3.5	Anforderungen an das IoT-Gerät	59
3.5.1	Allgemeine Anforderungen an das IoT-Gerät	59
3.5.2	Anforderungsanalyse für das IoT-Gerät	59
3.6	Anforderungen an die Peripheriegeräte	63
3.6.1	Allgemeine Anforderungen an externe Komponenten	63
3.6.2	Anforderungen an die Bodenfeuchtemessung	64
3.6.3	Anforderungen an Temperatur- und Luftfeuchtigkeitsmessung	64
3.6.4	Anforderungen an die Lichtstärkemessung	65
3.6.5	Anforderungen an die Schaltlogik (Relaismodul)	65
3.6.6	Anforderungen an den Pumpen-Aktor	66
3.6.7	Anforderungen an die externe Pumpen-Stromversorgung	66
3.6.8	Nicht-funktionale Anforderungen an Peripheriegeräte	67

4 Auswahl der Technologien	69
4.1 Auswahl von Vue.js für die Implementierung	69
4.1.1 Modularität und Komponentenstruktur	69
4.1.2 Reaktivität und Datenbindung	69
4.1.3 Community, Dokumentation und Lernkurve	69
4.1.4 Flexibilität, Integration und Zukunftssicherheit	70
4.1.5 Vergleich: Options API vs. Composition API in Vue.js	70
4.1.6 Einschränkungen und Gegenmaßnahmen	72
4.2 Einsatz von Tailwind CSS und Shadcn-Vue	73
4.2.1 Tailwind CSS: Utility-First-Ansatz	73
4.2.2 Shadcn-Vue: Komponentenbibliothek für Vue 3	73
4.2.3 Kombination beider Technologien im Frontend-Projekt	74
4.3 Internationalisierung der Anwendung in Vue.js	74
4.3.1 Motivation und Bedeutung	74
4.3.2 Technologiewahl: vue-i18n	75
4.3.3 Fazit	75
4.4 State-Management mit Pinia und Persistenz	75
4.4.1 Einordnung von State Management in SPAs	76
4.4.2 Persistenz mit <code>pinia-plugin-persistedstate</code>	76
4.4.3 Bewertung und Grenzen	76
4.5 Mobile Kompilierung mit Capacitor	77
4.5.1 Funktionsweise von Capacitor	77
4.5.2 Nutzung nativer APIs	77
4.5.3 Besonderheiten und Herausforderungen	77
4.5.4 Vor- und Nachteile der Verwendung von Capacitor	78
4.5.5 Bewertung für das Projekt	78
4.6 Wahl der Programmiersprache	78
4.6.1 Ein Einblick in Go	79
4.6.2 Einblicke in Rust	81
4.6.3 Syntax und Struktur	82
4.6.4 Vergleich: Rust vs. Go	84
4.7 Datenbankentscheidungen	85
4.7.1 Exkurs: PostgreSQL	86
4.7.2 Exkurs: MongoDB	90
4.7.3 Diskurs: Vergleich von PostgreSQL und MongoDB für NOMTIS	93
4.8 Auswahl der Technologien für die entwickelten Schnittstellen-Services	95

4.8.1	Technologieauswahl für den Authentifizierungs- und Registrierungsdienst (Auth-Service)	96
4.8.2	Technologieauswahl für den E-Mail-Verifikationsdienst (Mail-Service)	98
4.8.3	Technologieauswahl für den Datenpersistenzdienst (Database Writer)	100
4.8.4	Technologieauswahl für die Zielwert-Schnittstelle (Setpoint API) . .	102
4.8.5	Technologieauswahl für den Initialisierungsskript-Dienst (Solace Init)	103
4.9	Vergleichsanalyse: ESP32-WROOM-32D vs. Arduino vs. Raspberry Pi . .	106
4.9.1	Architektur und Rechenleistung	107
4.9.2	Unterstützung von Multitasking / RTOS	108
4.9.3	Schnittstellenvielfalt und Sensoranbindung	110
4.9.4	Netzwerkfähigkeit	112
4.9.5	Speicher- und Persistenzmöglichkeiten	113
4.9.6	Energieverbrauch und Batterietauglichkeit	116
4.9.7	Ökosystem, Entwicklungsumgebung und Community-Support . . .	118
4.9.8	Kosten, Verfügbarkeit und Formfaktor	121
4.9.9	Eignung der Plattformen für spezifische Projektanforderungen . .	123
4.9.10	Entscheidung: Warum ESP32-WROOM-32D?	127
4.10	Systemtechnologische Entscheidungen für das ESP32-basierte Bewässerungssystem	131
4.10.1	Entwicklungsframework	131
4.10.2	Dual-Core-Architektur und Taskmodell	131
4.10.3	Sensorik	132
4.10.4	Aktorik und Energieversorgung	134
5	Umsetzung	136
5.1	Servicearchitektur	136
5.1.1	Vorteile der Microservice-Architektur im Vergleich zum Monolithen	136
5.1.2	Hosting und Systemverbindungen	137
5.2	Frontendarchitektur und Datenflüsse im System	138
5.2.1	Komponentenbasierte Struktur und Navigationsmodell	138
5.2.2	Pinia als Vermittlungsinstanz zwischen API und Frontend	139
5.2.3	Datenfluss nach dem Flux-Prinzip	140
5.2.4	Fazit	140
5.3	Benutzerzentriertes Design und UI/UX im Frontend	141
5.3.1	Anwendung von UCD im smarten Bewässerungssystem	141
5.3.2	Verwendete Heuristiken in der Anwendung	142
5.3.3	Responsive Design	143
5.3.4	User-Stories und funktionale Umsetzung	143

5.4	Erweiterte Frontend-Techniken	144
5.4.1	Lazy Loading in der Anwendung	144
5.4.2	Frontend-Messung mit Lighthouse	144
5.5	Aufbau einer spezifischen View als Vertreter	145
5.6	KI-Komponente zur automatisierten Pflanzenklassifikation	147
5.6.1	Modellarchitektur und Trainingsstrategie	148
5.6.2	Regularisierung und Datenkonsolidierung	149
5.6.3	Leistung und Interpretation	149
5.6.4	Rolle der KI-Komponente bei der Pflanzenerstellung	151
5.7	Beschreibung des Datenbankaufbaus	151
5.7.1	Struktur und Besonderheiten	153
5.7.2	Technische Merkmale	153
5.8	Auth-Service: Geräteregistrierung und HMAC-Authentifizierung	154
5.8.1	Architektur und persistente Datenhaltung	154
5.8.2	Ablauf der Geräte-Registrierung und Authentifizierung	155
5.9	Mail-Service: E-Mail-Verifikation von Benutzerkonten	159
5.9.1	Architektur und Ablauf der E-Mail-Verifikation	159
5.10	Database Writer: MQTT-Datenpersistierung in PostgreSQL	161
5.10.1	Architektur: Dauerhafter Queue-Consumer	162
5.11	Setpoint API: Sollwert-Vorgabe via REST und MQTT	166
5.11.1	Funktionsweise und Ablauf	166
5.12	Solace Init: Automatisierte Broker-Konfiguration	168
5.12.1	Vorgehen und Konfiguration	168
5.13	Programmierung des ESP	170
5.13.1	Initialisierung des ESP	171
5.13.2	Regelbetrieb des ESP	179
6	Kritische Reflexion	191
6.1	Reflexion zur Frontend-Umsetzung	192
6.1.1	Verbesserungspotential	193
6.1.2	Fazit	193
6.2	Reflexion der Microcontroller-Programmierung	194
6.2.1	Komplexität der Hardwareanbindung	194
6.2.2	Entwicklung der Steuerungslogik	194
6.2.3	Fehleranalyse und Debugging	195
6.2.4	Codequalität und Modularität	195
6.2.5	Lernfortschritt und technologische Erkenntnisse	195

6.3	Kritische Refelexion Auth-Service	196
6.4	Kritische Reflexion Mailservice	197
6.5	Kritische Reflexion Database Writer	198
6.6	Kritische Reflexion Setpoint-API	200
6.7	Kritische Reflexion Solace-Init	201
7	Ausblick	203
7.1	Ausblick Frontend	204
7.2	Ausblick Microcontroller	205
7.2.1	Erweiterung der Steuerungslogik	205
7.2.2	Pufferung und Ausfallsicherheit	205
7.2.3	Sicherheit und Datenschutz	205
7.2.4	Erweiterung der Nutzerinteraktion	206
7.2.5	Updatefähigkeit und Wartung	206
7.2.6	Sensorik und Hardwarequalität	206
7.2.7	Systemintegration und Skalierbarkeit	207
	Literaturverzeichnis	208

Abkürzungen

ACID	Atomicity Consistency Isolation and Durability
API	Application Programming Interface
BSON	Binary JSON
CORS	Cross-Origin Resource Sharing
CRUD	Create Read Update Delete
DOM	Document Object Model
GC	Garbage Collector
GUI	grafische Benutzeroberfläche
JSON	JavaScript Object Notation
JWT	JSON Web Token
KI	Künstliche Intelligenz
NLL	Non Lexical Lifetimes
NOMTIS	Notification and Message Tracking Integration Service
POC	Proof of Concept
ResNet	Residual Network
ResNet50	50-layer Residual Network
REST	Representational State Transfer
REST	Representational State Transfer
RFC	Request for Comments
SEO	Search-Engine Optimization
SFC	Single File Component
SIT	Schwarz IT KG
SOAP	Simple Object Access Protocol
SPA	Single-page application
SQL	Structured Query Language
SSR	Server-Side-Rendering
TDD	Test-Driven Development

TLS Transport Layer Security

UCD User-Centered Design

UI User Interface

UX User Experience

WAL Write-Ahead Log

Abbildungsverzeichnis

2.1	Schematische Darstellung eines Publish/Subscribe Event Brokers mit Solace.	23
2.2	Challenge-Response-Authentifizierung mit Pre-Shared Key und Fernet-Token.	34
2.3	Bild eines ESP32 Mikrocontrollers	37
4.1	PostgreSQL Architektur. Quelle: [95] S. 33 Bild 4.1	87
4.2	Verbindungsauflaufbau von Client zum Server. Quelle: [95] S. 35 Bild 4.3 . . .	88
4.3	PostgreSQL Speicherarchitektur. Quelle: [95] S. 37 Bild 4.4	89
5.1	Kommunikation zwischen den Komponenten von Sensora	137
5.2	SinglePlantView in Aktion	146
5.3	Korrekt erkannt als Anthurium Andraeanum mit 99,72%	150
5.4	Korrekt erkannt als Fragaria X Ananassa mit 90,42%	150
5.5	Korrekt erkannt als Lavandula Stoechas mit 99,49%	150
5.6	Korrekt erkannt als Lavandula Angustifolia mit 98,05%	150
5.7	Beispiel für die KI-Erkennung	150
5.8	Sensora Datenbank Struktur	152
5.9	Ablaufplan Initialisierung des ESP	172
5.10	Ablaufplan Regelbetrieb des ESP	180

Listings

2.1	Beispielhafter Unit-Test mit Vue Test Utils	16
2.2	Beispielhafter Cypress-Test	16
4.1	Beispiel Options API	71
4.2	Beispiel für die Composition API mit <code><script setup></code>	72
5.1	Lazy Loading per Route in Vue Router	144
5.2	Senden der Sensordaten	182
5.3	Empfangen von Solace-Daten	183
5.4	Start der Pump-Task	185
5.5	Start der Sensorik-Task	186
5.6	Mittelwertbildung zur Glättung von Sensorwerten	187
5.7	Überprüfung der Sensoraktivität	187
5.8	Berechnung der Bewässerungsdauer	189

1 Einleitung

1.1 Motivation

Pflanzen gehören in Deutschland und Europa fest zum Alltag in Wohnung und Garten. Laut einer repräsentativen Umfrage aus dem Jahr 2020 besitzen rund drei Viertel der Bundesbürger:innen (74 %) Zimmerpflanzen in ihrem Zuhause; auch auf Balkonen (35 %), Terrassen (30 %) und Fensterbänken (21 %) grünt es, während nur etwa 10 % ganz ohne Pflanzen leben. [1] Dieses „grüne Zuhause“ liegt im Trend und gewann insbesondere während der COVID-19-Pandemie an Bedeutung – viele Menschen entdeckten 2020 im Home-Office ihre Liebe zu Haus- und Gartenpflanzen neu. [2] Entsprechend stieg der Absatz: Der deutsche Markt für Blumen und Zierpflanzen erreichte nach Jahren der Stagnation 2020 ein Rekordvolumen von 9,4 Mrd. €. [2]

Ähnlich hohe Werte zeigen sich europaweit, wo Pflanzen als wichtiger Teil der Wohn- und Lebensqualität gelten. Neben dekorativen Aspekten werden Zimmer- und Gartenpflanzen aufgrund positiver Effekte wie besserer Luftqualität und Stressreduktion geschätzt. [1] Die hohe Verbreitung und Wertschätzung von Pflanzen in Privathaushalten bildet den Ausgangspunkt für die Betrachtung, wie ihre Pflege im Alltag unterstützt werden kann. Allerdings stehen viele Pflanzenbesitzer:innen vor praktischen Herausforderungen bei der Pflege ihres „grünen Mitbewohners“. Im hektischen Alltag wird das Gießen leicht vergessen oder unregelmäßig vorgenommen; umgekehrt gießen unerfahrene Halter oft zu viel aus Sorge um die Pflanze. Studien bestätigen, dass Überwässerung der häufigste Grund für das Eingehen von Zimmerpflanzen ist. [3]

Generell erfordert jede Pflanzenart spezifische Kenntnisse zu Wasser- und Nährstoffbedarf, Lichtverhältnissen etc., über die im privaten Umfeld nicht immer ausreichend Wissen vorhanden ist. So gaben in einer Umfrage lediglich 37 % der befragten Frauen und 20 % der Männer an, einen „grünen Daumen“ zu haben [1] – die Mehrheit traut sich die optimale Pflanzenpflege also eher nicht zu. Hinzu kommt, dass während Urlaubs- oder Abwesenheitszeiten oft keine Betreuung für die heimischen Gewächse sichergestellt ist. Tatsächlich vermissten in einer Befragung 26 % der Pflanzenhalter:innen ihre Zimmerpflanzen im Urlaub sogar mehr als die Kolleg:innen [1], was die emotionale Bindung und zugleich das

Problem der Versorgung in dieser Zeit verdeutlicht. Diese Pflegeherausforderungen führen dazu, dass viele privat gehaltene Pflanzen Schäden nehmen oder vorzeitig absterben. Die Folgen von falscher oder unregelmäßiger Pflege sind in Zahlen beträchtlich. Hochrechnungen zufolge überlebt ein erheblicher Teil der gekauften Zierpflanzen nicht lange: Viele Pflanzen gehen bereits in der Lieferkette zugrunde, und rund 35 % der gekauften Zimmerpflanzen sterben später in den Wohnungen der Kundschaft. [4] Mit anderen Worten wird etwa ein Drittel aller gekauften Haus- und Gartenpflanzen letztlich aufgrund suboptimaler Bedingungen oder Pflegefehler nicht dauerhaft erhalten. Auch Verbraucherumfragen deuten auf dieses Problem hin. Beispielsweise gab über ein Drittel der Hobbygärtner in einer aktuellen Erhebung an, jedes Jahr ein bis zwei Zimmerpflanzen zu verlieren. [5] Solche Verluste sind nicht nur emotional enttäuschend für Pflanzenliebhaber, sondern bedeuten auch Ressourcenverschwendungen – insbesondere von Wasser, Zeit und Geld. Schätzungen aus den USA zeigen etwa, dass jüngere „Plant Parents“ im Durchschnitt schon Sieben ihrer erworbenen Pflanzen unbeabsichtigt zum Eingehen gebracht haben. [6] Diese Zahlen unterstreichen die Notwendigkeit, neue Wege zu finden, um häufige Pflegefehler zu vermeiden und die Lebensdauer der Pflanzen zu verlängern.

Technologische Lösungen im Sinne von *Smart Gardening* setzen hier an und versprechen Abhilfe. Insbesondere automatische Bewässerungssysteme für den Heimgebrauch bieten die Möglichkeit, den Gießvorgang zu optimieren und zu automatisieren. Solche Systeme kombinieren oft Sensoren (etwa für Bodenfeuchte oder Licht) mit internetfähigen Steuerungen, um den Pflanzen exakt bei Bedarf und in der richtigen Menge Wasser zuzuführen. Erste Ansätze sind bereits auf dem Markt verfügbar – von App-gesteuerten Bewässerungscomputern bis hin zu smarten Pflanzentöpfen mit Selbstbewässerungs-Funktion. Die Akzeptanz solcher *Smart-Home*-Technologien im Garten- und Pflanzenbereich steigt kontinuierlich. Laut dem STIHL-Gartenbarometer 2022 nutzen bereits rund 7 % der deutschen Gartenbesitzer smarte Garden-Lösungen, und etwa 30 % wünschen sich zukünftig solche automatisierten Helfer. [7] Dabei stehen Bewässerungsautomationen an erster Stelle der Wunschliste: 83 % der Befragten mit Smart-Gardening-Interesse nennen ein automatisches Bewässerungssystem als besonders gefragte Lösung. Diese Nachfrage spiegelt sich auch in anderen Ländern wider. Beispielsweise glauben in Österreich über 60 % der Gartenbesitzer, dass sich der Wasserverbrauch durch automatisierte Bewässerungsanlagen deutlich optimieren lässt. [8]

Moderne Systeme können Wetterdaten oder Bodensensoren einbeziehen, um nur dann zu wässern, wenn die Pflanze es wirklich benötigt – eine Technik, die den Pflanzenstress reduziert und zugleich Wasserverschwendungen vorbeugt. Aktuelle Untersuchungen zeigen denn auch, dass intelligente Bewässerungssteuerungen den Wasserverbrauch im Garten gegenüber herkömmlichen Timern erheblich senken können (um etwa 20–40 % je nach

System). [9]

Mehrere übergeordnete Trends begünstigen die Verbreitung von smarten Pflanzenpflege-Systemen. Zum einen führt die Urbanisierung dazu, dass immer mehr Menschen auf kleinem Raum in Städten leben – in Deutschland etwa 78 % der Bevölkerung [10] – und sich dennoch nach Natur im eigenen Umfeld sehnen. Insbesondere Stadtbewohner ohne Garten kultivieren vermehrt Zimmerpflanzen oder Balkongrün, sind aber beruflich oft stark eingebunden. Eine automatische Bewässerung kann hier den Pflegeaufwand mindern und sicherstellen, dass Pflanzen trotz hektischem Alltag oder Abwesenheiten ausreichend versorgt werden.

Zum anderen rückt Nachhaltigkeit in den Fokus: Wassermanagement und effiziente Resourcennutzung gewinnen an Bedeutung, da die Auswirkungen des Klimawandels – etwa häufigere Sommerdürreperioden – auch private Gärten und Balkone betreffen. In Umfragen äußern fast zwei Drittel der Befragten die Erwartung, dass digitale Technologien im Garten helfen können, den Klimawandel abzuschwächen, und nennen den schonenden Umgang mit Wasser als oberste Priorität. [8] Smart-Bewässerungssysteme erfüllen genau diesen Zweck, indem sie bedarfsgerecht gießen und Überwässerung verhindern.

Schließlich trägt auch die allgemeine Verbreitung von *Internet of Things*-Anwendungen im Haushalt dazu bei, dass vernetzte Lösungen immer selbstverständlicher werden. Der europäische Smart-Home-Markt verzeichnet hohe Wachstumsraten und wird 2024 bereits auf über 22 Mrd. US-\$ geschätzt [11]. Vernetzte, per App oder Sprache steuerbare Geräte – vom Thermostat bis zur Lichtsteuerung – gehören zunehmend zum Alltag. Diese Entwicklung macht auch vor dem Bereich der Pflanzenpflege nicht Halt: Die Nutzerakzeptanz für digitale Helfer im Haushalt schafft ein günstiges Umfeld für *Smart Gardening*-Innovationen.

Insgesamt ist die Einführung eines smarten Bewässerungssystems im heimischen Umfeld vor dem Hintergrund dieser Fakten sowohl technisch zeitgemäß als auch gesellschaftlich sinnvoll. Die weit verbreitete Haltung von Zimmer- und Gartenpflanzen einerseits und die häufig auftretenden Pflegeprobleme andererseits schaffen ein deutliches Bedürfnis nach Unterstützung. Automatisierte Bewässerungslösungen können hier einen doppelten Nutzen stiften: Sie helfen Pflanzenbesitzer:innen, ihre grünen Schützlinge zuverlässig und fachgerecht zu versorgen, und tragen zugleich zu Nachhaltigkeit und Komfort bei. Indem ein smartes Bewässerungssystem Wasser bedarfsgerecht dosiert und den Pflegeprozess vereinfacht, steigert es die Überlebensrate und Vitalität der Pflanzen und entlastet den Menschen von Routineaufgaben.

Die vorliegenden Studien, Statistiken und Trends untermauern somit die Notwendigkeit und den Nutzen eines solchen Systems, das im Folgenden technisch konzipiert und beschrieben wird.

1.2 Zielsetzung

Ziel dieser Studienarbeit ist die Konzeption, Entwicklung und prototypische Umsetzung eines automatisierten Systems zur Bewässerung von Zimmerpflanzen im privaten Wohnumfeld. Es soll eine lauffähige Gesamtlösung entstehen, die aus einem Mikrocontroller als zentrale Steuereinheit, einer Backend-Infrastruktur zur Datenverarbeitung und -persistierung sowie einem benutzerfreundlichen Frontend zur Visualisierung und Steuerung besteht. Die Realisierung erfolgt im Rahmen eines *Proof of Concept*, der die technische Machbarkeit sowie die Integration der Systemkomponenten demonstriert.

Das zu entwickelnde System erfasst über geeignete Sensorik (z. B. Bodenfeuchte, Temperatur, Luftfeuchtigkeit, Lichtintensität) kontinuierlich relevante Umgebungsdaten. Diese Messwerte dienen entweder als Entscheidungsgrundlage für den Nutzer, um über die Benutzeroberfläche manuell eine Bewässerung anzustoßen, oder sie werden vom Mikrocontroller automatisch verarbeitet. Im letzteren Fall wird anhand zuvor definierter Sollwerte eine autonome Steuerung der Bewässerungseinheit realisiert. Vorrangiges Ziel ist die technische Umsetzung des automatischen Betriebsmodus. Die Konzeption und Entwicklung des manuellen Modus sowie die Integration beider Steuerungsarten in das Gesamtsystem erfolgen nachrangig und abhängig von den im Projektverlauf verfügbaren Entwicklungskapazitäten.

Die Bewässerungslösung ist primär für den Einsatz in Innenräumen konzipiert. Dies umfasst insbesondere Haushalte mit Zimmerpflanzen, bei denen typische Pflegeprobleme wie unregelmäßiges Gießen oder Unsicherheit bezüglich des Wasserbedarfs adressiert werden sollen.

Der konkrete Funktionsumfang des Systems wird im Verlauf des Projekts iterativ entwickelt. Eine detaillierte Beschreibung der funktionalen und nicht-funktionalen Anforderungen sowie der Zielsystemeigenschaften erfolgt in Kapitel ???. Dabei wird angestrebt, etablierte *Best Practices* der Software- und Systementwicklung zu berücksichtigen und – wo sinnvoll und realisierbar – aktuelle Technologien gemäß dem Stand der Technik (*State of the Art*) zu verwenden. Gleichzeitig wird die technische Umsetzung unter Berücksichtigung der konzeptionellen Natur als *Proof of Concept* gewichtet, sodass pragmatische Abwägungen hinsichtlich Komplexität, Aufwand und Ressourcen erfolgen.

Insgesamt dient die Arbeit dem Ziel, ein funktional überzeugendes Demonstrationssystem zu realisieren, das eine fundierte Grundlage für weiterführende Entwicklungen, Evaluierungen oder mögliche Produktivsetzungen bietet.

Code Base in GitHub Repository

In diesem Paper wird wiederholt Programmcode referenziert. Dieser ist in einem GitHub Repository hinterlegt, das unter folgendem Link zu finden ist:

<https://github.com/Juqsi/Sensora>

2 Theoretische Grundlagen

2.1 Scrum und das Agile Manifest

Scrum basiert auf den Grundsätzen des Agilen Manifests. Das Agile Manifest ist eine Sammlung von Priorisierungsprinzipien, die im Jahr 2001 von 17 Experten und Vertretern unterschiedlicher agiler Vorgehensweisen in Snowbird, USA, entwickelt wurde. Diese Gruppe, oft als die „Snowbird 17“ bezeichnet, erkannte frühzeitig die Notwendigkeit einer neuen Ära der Softwareentwicklung. [12] Das Agile Manifest umfasst im Original nur 68 Wörter, aber aus diesem knappen Text wurden allgemeingültige Prinzipien abgeleitet, die bis heute die Grundlage für agile Methoden wie Scrum bilden.

Das Manifest lautet sinngemäß ins Deutsche übersetzt:

„Wir erschließen bessere Wege, Software zu entwickeln, indem wir es selbst tun und anderen dabei helfen. Durch diese Tätigkeit haben wir diese Werte zu schätzen gelernt:
Individuen und Interaktionen

mehr als Prozesse und Werkzeuge,

Funktionierende Software

mehr als umfassende Dokumentation,

Zusammenarbeit mit dem Kunden

mehr als Vertragsverhandlung,

Reagieren auf Veränderung

mehr als das Befolgen eines Plans.

Das heißt, obwohl wir die Werte auf der rechten Seite wichtig finden, schätzen wir die Werte auf der linken Seite höher ein.“ [13]

2.1.1 Interpretation des Agilen Manifests in der Praxis

Das Agile Manifest betont die Wichtigkeit von Praxisnähe in der agilen Entwicklung. Es reicht nicht aus, nur theoretische Konzepte zu entwickeln; vielmehr müssen praktische Erfahrungen gesammelt und in den Entwicklungsprozess eingebracht werden. Ein zentrales Prinzip ist, dass die Individuen und ihre Interaktionen im Vordergrund stehen.

Dies bedeutet, dass ein Vorgehen gefunden werden muss, das eine effektive Kommunikation und Interaktion aller Beteiligten ermöglicht. Prozesse und Werkzeuge sollten an die Bedürfnisse der Menschen angepasst werden, nicht umgekehrt.

Darüber hinaus wird die Bedeutung funktionierender Software hervorgehoben. Der Fortschritt eines Projekts wird anhand der tatsächlich funktionierenden Software gemessen, nicht anhand umfangreicher Dokumentationen. Es ist entscheidend, dass die entwickelte Software in regelmäßigen Abständen gezeigt und von den Anwendern beurteilt wird. Dies stellt sicher, dass das Projekt auf dem richtigen Weg bleibt und die Anforderungen der Nutzer erfüllt.

Da Softwareentwicklung ein dynamischer Prozess ist, entstehen oft neue Herausforderungen oder Anforderungen, selbst bei sorgfältiger Planung. Daher ist es unerlässlich, flexibel auf Veränderungen zu reagieren. Der Erfolg eines Projekts wird daran gemessen, wie gut es sich an neue Gegebenheiten anpasst und ob daraus ein Lerneffekt resultiert, der das Projekt voranbringt.

Auch wenn das Manifest die Bedeutung von Prozessen, Dokumentation, Verträgen und Plänen anerkennt, stellt es klar, dass diese Aspekte im Vergleich zu den übergeordneten Prinzipien von geringerer Priorität sind. Sie haben jedoch weiterhin ihre Daseinsberechtigung und müssen in einem angemessenen Maße berücksichtigt werden. [14]

2.1.2 Zusammenhang zwischen Scrum und dem Agilen Manifest

Scrum operationalisiert die Prinzipien des Agilen Manifests in einem strukturierten Rahmenwerk. Die regelmäßigen Sprints und die damit verbundenen Meetings – wie das Sprint Planning, Daily Stand-ups, Sprint Reviews und Retrospektiven – fördern die Kommunikation und die Interaktion zwischen den Teammitgliedern und den Stakeholdern. Durch die iterative Natur von Scrum wird sichergestellt, dass funktionierende Software frühzeitig und kontinuierlich geliefert wird, wodurch die Kundenzufriedenheit gesteigert wird.

Die enge Zusammenarbeit mit dem Kunden, die in Scrum durch die Rolle des Product Owners verkörpert wird, gewährleistet, dass das Entwicklungsteam ständig auf die sich ändernden Anforderungen reagieren kann. Diese Flexibilität ist ein direkter Ausdruck des Wertes „Reagieren auf Veränderung mehr als das Befolgen eines Plans“, der im Agilen Manifest verankert ist.

2.1.3 Anwendung von Scrum im Sensorsa-Projekt

Aus den Grundsätzen des Agilen Manifests ergeben sich spezifische Herangehensweisen für das Sensorsa-Projekt:

Zwischenergebnisse: Es wird eine hohe Frequenz bei der Präsentation von Zwischenergebnissen angestrebt. Diese regelmäßigen Präsentationen bieten eine hervorragende Gelegenheit, um mit den Stakeholdern in den Dialog zu treten, Feedback zu sammeln und Verbesserungen zu identifizieren. Zudem fungieren diese Präsentationen als Indikatoren für den Projektfortschritt, wodurch erkennbar wird, ob das Projekt planmäßig voranschreitet oder ob Maßnahmen zur Kurskorrektur erforderlich sind.

Kommunikation: Alle technischen Entscheidungen werden in enger Abstimmung mit allen Entwicklern getroffen. Dabei werden die Meinungen und Bedenken der beteiligten Personen berücksichtigt, um sicherzustellen, dass realistische Lösungen verfolgt werden. Durch diesen intensiven Austausch wird verhindert, dass Zeit und Ressourcen in ineffiziente oder unpraktikable Lösungen investiert werden. Gleichzeitig wird sichergestellt, dass das kollektive Wissen genutzt wird und potenzielle Probleme frühzeitig erkannt werden.

Lessons Learned: Im Verlauf der Entwicklung entstehen neue Erkenntnisse, die zu neuen Möglichkeiten führen. Diese Lernfortschritte, sowohl auf fachlicher als auch auf technischer Ebene, werden genutzt, um das Projekt kontinuierlich weiterzuentwickeln und anzupassen. Die Fähigkeit, aus Erfahrungen zu lernen und diese in den Entwicklungsprozess einzubeziehen, ist ein zentraler Bestandteil des agilen Vorgehens.

Zusammengefasst projiziert Scrum die Werte und Prinzipien des Agilen Manifests auf einen praxisnahen und strukturierten Entwicklungsprozess, der es Teams ermöglicht, effizient und flexibel auf die Herausforderungen der Softwareentwicklung zu reagieren. Durch die Integration dieser Prinzipien in das Sensorsa-Projekt wird sichergestellt, dass das Produkt den Anforderungen gerecht wird und gleichzeitig eine hohe Qualität und Benutzerfreundlichkeit erreicht.

2.2 Single-page applications und Frameworks

Moderne Webentwicklung ist zunehmend durch SPA geprägt, die im Vergleich zu klassischen Multi-Page Applications (MPAs) durch ein dynamischeres Nutzererlebnis überzeugen. SPAs laden nach dem initialen Seitenaufruf keine vollständigen HTML-Dokumente vom Server nach, sondern aktualisieren Inhalte durch JavaScript-Logik auf der Client-Seite [15].

2.2.1 Entstehung und Motivation von SPAs

Das Konzept der SPA entstand im Kontext steigender Nutzererwartungen an reaktions-schnelle Webanwendungen. Während frühere Webarchitekturen bei jeder Nutzerinteraktion eine komplette neue HTML-Seite vom Server luden, ermöglichen SPA eine unterbrechungsfreie Interaktion, indem Inhalte dynamisch aktualisiert werden. Technologisch wurde diese Entwicklung durch die Verfügbarkeit von AJAX, JavaScript-Frameworks sowie Browser APIs wie dem History-API begünstigt [15].

2.2.2 Architektur, Vorteile und Herausforderungen von SPAs

SPAs kommunizieren typischerweise über Representational State Transfer (REST)- oder GraphQL-APIs mit einem Backend und verwalten Zustände lokal im Browser. Durch client-seitiges Routing, etwa mit Bibliotheken wie React Router oder Vue Router, wird eine app-ähnliche Navigation ermöglicht.

Vorteile:

- **Verbessertes Nutzererlebnis:** Durch den Wegfall von Seiten reloads werden Ladezeiten reduziert, was zu einer flüssigeren Interaktion führt.
- **Geringere Serverlast:** Da nur Daten, nicht aber komplette Seiten geladen werden, reduziert sich die Serverauslastung.
- **Bessere Trennung von Frontend und Backend:** Die API-Zentrierung fördert modulare Systemarchitekturen und erleichtert die Wiederverwendung von Backend-Ressourcen.

Nachteile:

- **Schwächere:** Ohne Server-Side-Rendering (SSR) sind Inhalte für Suchmaschinen schlechter zugänglich.
- **Erhöhte Komplexität:** Zustandsverwaltung, Routing und Sicherheitsaspekte müssen client-seitig implementiert werden.
- **Initiale Ladezeit:** Die gesamte Anwendung (inkl. JavaScript) muss initial geladen werden, was den ersten Ladevorgang verzögern kann [16].

2.2.3 Vergleich von React, Angular und Vue

React ist eine von Facebook entwickelte Bibliothek zur Erstellung von Benutzeroberflächen. Es folgt einem funktionalen, komponentenbasierten Paradigma und nutzt die virtuelle Document Object Model (DOM) für Performanceoptimierungen. React ist minimalistisch und erfordert oft ergänzende Bibliotheken für Routing oder State-Management wie Redux oder React Router [17].

Angular ist ein umfassendes Framework von Google, das auf TypeScript basiert. Es bietet eine vollständige Lösung inklusive Dependency Injection, Routing, Formularverwaltung und mehr. Angular eignet sich besonders für große Enterprise-Anwendungen, bringt jedoch eine steilere Lernkurve mit sich [18].

Vue hingegen ist ein progressives Framework, das sich zwischen React und Angular positioniert. Es ist leichtgewichtig und modular, bietet jedoch mit seinem offiziellen Ökosystem (Vue Router, Vuex/Pinia) eine vollständige Entwicklungsumgebung [19]. Die Reaktivierung erfolgt über ein Proxy-basiertes System, das automatische DOM-Updates bei Datenänderungen ermöglicht [20].

2.3 Benutzerzentriertes Design und UI/UX im Frontend

Im Rahmen der theoretischen Grundlagen ist die Gestaltung einer benutzerfreundlichen und intuitiven Benutzeroberfläche ein zentraler Aspekt der Frontend-Entwicklung. Gerade bei SPAs ist die Nutzerinteraktion entscheidend für den Erfolg der Anwendung, da technische Funktionalität und Design eng miteinander verzahnt sind. In diesem Kapitel werden daher zentrale Konzepte wie User-Centered Design (UCD), Usability-Heuristiken, Responsive Design sowie prototypische Evaluationsmethoden behandelt und anhand des smarten Bewässerungssystems konkretisiert.

2.3.1 User-Centered Design (UCD)

User-Centered Design (UCD) bezeichnet einen iterativen Gestaltungsprozess, der die Bedürfnisse, Anforderungen und Einschränkungen der EndnutzerInnen systematisch in den Mittelpunkt stellt [21]. Ziel ist es, Systeme zu entwickeln, die für die intendierte Zielgruppe funktional, zugänglich und zufriedenstellend nutzbar sind. Die methodische Umsetzung umfasst typischerweise vier Phasen: Nutzerforschung, Anforderungsdefinition, Prototyping und Evaluation.

2.3.2 Usability-Heuristiken nach Nielsen

Die zehn Usability-Heuristiken von Jakob Nielsen gelten als grundlegende Prinzipien zur Bewertung der Gebrauchstauglichkeit grafischer Benutzungsschnittstellen [22]. Diese lauten:

1. **Sichtbarkeit des Systemstatus:** NutzerInnen sollten stets darüber informiert sein, was im System vorgeht. Dies erfolgt durch visuelles oder auditives Feedback in angemessener Zeit.
2. **Entsprechung zwischen System und realer Welt:** Das System sollte die Sprache der NutzerInnen sprechen, mit vertrauten Begriffen, Konzepten und logischen Abläufen.
3. **Benutzerkontrolle und -freiheit:** NutzerInnen sollten stets die Möglichkeit haben, unbeabsichtigte Aktionen rückgängig zu machen oder zu verlassen (z. B. durch eine „Zurück“-Funktion).
4. **Konsistenz und Standards:** Einheitliche Begriffe, Symbole und Designmuster sollten in der gesamten Anwendung verwendet werden, um Lernaufwand zu minimieren.
5. **Fehlervorbeugung:** Systeme sollten so gestaltet sein, dass Fehler bereits im Vorfeld vermieden werden (z. B. durch Eingabebeschränkungen oder Bestätigungsdialoge).
6. **Wiedererkennung statt Erinnerung:** Die Benutzeroberfläche sollte Informationen sichtbar und abrufbar machen, anstatt sich auf das Gedächtnis der NutzerInnen zu verlassen.
7. **Flexibilität und Effizienz der Nutzung:** Die Anwendung sollte sowohl AnfängerInnen als auch fortgeschrittene NutzerInnen durch Tastenkombinationen oder Shortcuts unterstützen.
8. **Ästhetisches und minimalistisches Design:** Die Darstellung sollte sich auf relevante Informationen beschränken und keine irrelevanten oder selten gebrauchten Inhalte enthalten.
9. **Hilfe beim Erkennen, Beheben und Vermeiden von Fehlern:** Fehlermeldungen sollten klar formuliert, Ursachen aufzeigen und konstruktive Lösungsvorschläge machen.
10. **Hilfe und Dokumentation:** Auch wenn ein System ohne externe Hilfe bedienbar sein sollte, ist dokumentierte Unterstützung für komplexe Aufgaben hilfreich.

Diese Prinzipien bieten ein umfassendes Rahmenwerk zur Gestaltung und Evaluation benutzerfreundlicher User Interfaces (UIs).

2.3.3 Gamification im UI/UX-Kontext

Ein zunehmend relevanter Gestaltungsansatz im Bereich benutzerzentrierter Systeme ist die Gamification, also die Anwendung spieltypischer Elemente in nicht-spielerischen Kontexten, um Motivation, Engagement und Nutzerbindung zu steigern. Im Rahmen von UI/UX verfolgt Gamification das Ziel, Interaktionen mit digitalen Produkten emotional aufzuladen und ein immersives Nutzungserlebnis zu schaffen [23].

Typische Gamification-Elemente sind unter anderem personalisierte visuelle Repräsentationen in Form von Avataren, die NutzerInnen eine Identifikation mit dem System erleichtern. Ebenso zählen Punkte, Levels und Fortschrittsanzeigen dazu, die als quantitative Rückmeldungen den NutzerInnen ein Gefühl von Zielerreichung und Fortschritt vermitteln. Darüber hinaus spielen Herausforderungen und Belohnungen eine wichtige Rolle, indem sie Aufgaben oder Ziele durch Belohnungssysteme motivierend verstärken. Ergänzt wird dies durch Feedback-Mechanismen, welche sofortige Reaktionen auf Nutzeraktionen bieten, etwa durch Animationen, akustische Signale oder farbliche Hervorhebungen[23], [24].

Gamification stellt somit eine strategische Erweiterung klassischer UX-Designprinzipien dar, die insbesondere in Anwendungen mit repetitiven Aufgaben oder hohem Interaktionspotenzial zur Steigerung der Motivation und der Verhaltensveränderung beitragen kann [23].

2.4 Theoretischer Vergleich: Webanwendungen versus native Apps

Die Auswahl einer geeigneten Plattformstrategie stellt eine zentrale Entscheidung im Rahmen der Softwarearchitektur dar. Dabei stehen insbesondere Webanwendungen, native Applikationen sowie hybride Entwicklungsansätze zur Debatte. Diese Sektion beleuchtet die theoretischen Grundlagen und vergleicht insbesondere Webanwendungen mit nativen Android-Apps hinsichtlich Architektur, Entwicklungskosten, Plattformunabhängigkeit, Performance und User Experience (UX).

2.4.1 Architekturmödelle mobiler und Webanwendungen

Grundsätzlich lassen sich vier Hauptkategorien unterscheiden:

1. **Native Apps:** Sie werden spezifisch für ein Betriebssystem wie Android (in Kotlin/Java) oder iOS (in Swift/Objective-C) entwickelt. Sie bieten vollen Zugriff auf Systemfunktionen und gelten als performanteste Option [25].
2. **Webanwendungen:** Sie laufen in einem Browser, basieren auf HTML, CSS und JavaScript und sind betriebssystemunabhängig. Sie müssen nicht installiert werden und sind über URLs zugänglich.
3. **Hybride Apps:** Diese kombinieren Webtechnologien mit nativen Container-Komponenten (z. B. Cordova oder Capacitor). Sie werden einmal entwickelt und können in mehreren App Stores publiziert werden [25].
4. **Cross-Platform-Apps:** Hierbei wird der Quellcode in einer plattformübergreifenden Sprache geschrieben (z. B. Dart bei Flutter oder JavaScript bei React Native) und nativ kompiliert. Ziel ist ein natives Look-and-Feel bei reduzierter Entwicklungszeit [26].

2.4.2 Vergleich von Webanwendungen und nativen Android-Apps

Plattformbindung und Zugänglichkeit Webanwendungen sind geräteunabhängig und benötigen lediglich einen Browser, was ihre Zugänglichkeit und Reichweite maximiert. Native Android-Apps hingegen müssen über den Play Store installiert werden und sind an die Android-Plattform gebunden [25].

Performance und Systemintegration Native Apps bieten bessere Performance, da sie direkt auf System-APIs zugreifen und hardwarenah ausgeführt werden. Dies ist vorteilhaft für grafikintensive oder hardwareabhängige Anwendungen. Webanwendungen sind dagegen durch die Browserumgebung limitiert, können aber durch moderne Web-APIs zunehmend auf Sensorik und Offline-Funktionen zugreifen (z. B. via Progressive Web Apps) [26].

Entwicklungs- und Wartungskosten Webanwendungen bieten durch einheitlichen Code für alle Plattformen eine höhere Wartungseffizienz und geringere Kosten. Native Android-Apps erfordern separate Entwicklungsprozesse für Android und ggf. weitere Plattformen, was zeit- und ressourcenintensiv ist [26].

UX und Benutzerbindung Native Apps ermöglichen eine tiefere Integration in das User Experience (UX)-Paradigma des Betriebssystems (z. B. Gesten, native Navigation, Push-Notifications). Webanwendungen sind hier limitiert, bieten aber durch Responsive Design und Progressive Enhancement eine übergreifende Benutzererfahrung [25].

2.4.3 Kombinationsstrategien

Angesichts der Vor- und Nachteile einzelner Plattformen existieren Bestrebungen, Synergien zu nutzen. Ein Ansatz ist die Entwicklung einer Webanwendung als Basis, die bei Bedarf über Frameworks wie Capacitor oder Cordova in native Apps umgewandelt wird. Dadurch lassen sich Web-Technologien mit gerätespezifischer Distribution kombinieren. Alternativ können Cross-Plattform-Frameworks wie Flutter oder React Native eingesetzt werden, um native App-Erlebnisse bei einmaliger Codebasis zu realisieren [26].

Abschließend lässt sich festhalten, dass die Entscheidung für Webanwendung oder native App stets kontextabhängig ist. Kriterien wie Funktionsumfang, Zielgruppe, Budget und langfristige Wartbarkeit sind dabei zentral.

2.5 Softwaretestens mit Fokus auf Frontend-Frameworks

Das Testen von Software stellt einen fundamentalen Bestandteil des Entwicklungsprozesses dar. Es dient der systematischen Qualitätssicherung und verfolgt das Ziel, Fehler frühzeitig zu identifizieren, Korrektheit zu überprüfen und die Wartbarkeit des Codes zu erhöhen. Insbesondere im Kontext moderner, komponentenbasierter Frontend-Frameworks wie Vue.js nimmt das Testen eine zentrale Rolle ein, um das dynamische Verhalten von Benutzeroberflächen valide zu verifizieren.

2.5.1 Ziele und Nutzen von Softwaretests

Softwaretests haben in der Praxis mehrere eng miteinander verknüpfte Funktionen. Sie dienen in erster Linie der Fehlererkennung und -vermeidung. Ein umfassend getestetes System weist eine signifikant geringere Wahrscheinlichkeit für kritische Laufzeitfehler oder nicht intendiertes Verhalten auf [27]. Gleichzeitig erfüllen Tests eine dokumentierende Funktion. Insbesondere automatisierte Tests können als maschinenlesbare Spezifikationen fungieren, da sie in kodifizierter Form definieren, wie sich Komponenten unter bestimmten Bedingungen verhalten sollen [28]. Darüber hinaus ermöglichen Tests die Durchführung von Regressionstests, bei denen sichergestellt wird, dass Änderungen im Quelltext keine unbeabsichtigten Nebeneffekte hervorrufen. Im Rahmen Test-Driven Development (TDD) werden Tests sogar vor dem eigentlichen Code geschrieben, was die Modularität und Wartbarkeit von Software verbessert.

2.5.2 Testarten und deren Abgrenzung

Die Theorie des Softwaretestens unterscheidet verschiedene Teststufen, die unterschiedliche Aspekte der Softwarequalität absichern. Unit-Tests stellen die unterste Ebene dar. Sie testen kleinste funktionale Einheiten, beispielsweise einzelne Funktionen oder Komponenten, isoliert von deren Kontext. Diese Tests sind schnell ausführbar und gut automatisierbar, bilden jedoch nicht das Zusammenwirken mehrerer Module ab.

An diese schließen sich Integrationstests an. Hier wird das Zusammenspiel mehrerer Komponenten oder Module geprüft. Ziel ist es, Schnittstellen und Interaktionen zwischen Teilsystemen zu validieren. Integrationstests sind insbesondere in komponentenbasierten Frameworks relevant, da viele logische Fehler nicht in der Einzelkomponente, sondern im Wechselspiel zwischen Komponenten auftreten.

Darüber hinaus existieren End-to-End-Tests (E2E), welche die gesamte Anwendung aus Sicht eines realen Nutzers durchlaufen. Dabei wird die gesamte Technologie-Stack inklusive Frontend, Backend und Persistenzschicht berührt. E2E-Tests sind besonders geeignet, um kritische Pfade wie Login-Prozesse, Formularinteraktionen oder komplexe Benutzerflüsse zu validieren. Sie zeichnen sich durch eine hohe Aussagekraft aus, sind jedoch in der Regel aufwändiger in Wartung und Ausführung [29].

2.5.3 Frontend-Testing mit Vue.js

Vue.js als komponentenbasiertes Framework bietet umfassende Möglichkeiten zur modularisierten Testung. Der offizielle Stack sieht insbesondere Werkzeuge wie Vue Test Utils, Jest und Cypress vor. Mit Vue Test Utils lassen sich einzelne Komponenten isoliert rendern und ihre Interaktionen mit dem DOM gezielt untersuchen [30]. Jest dient als Ausführungs-Umgebung für Unit- und Snapshot-Tests, wobei durch das Speichern von DOM-Zuständen automatisiert Regressionen erkannt werden können. Für End-to-End-Tests empfiehlt sich der Einsatz von Cypress, welches auf der Ebene realer Nutzerinteraktionen arbeitet und dabei u. a. Klicks, Navigationen und Eingaben überprüft.

Die Architektur von Vue-Komponenten, insbesondere deren Trennung in Template, Script und Style, ermöglicht eine gezielte Testbarkeit. Darüber hinaus fördert die Reaktivierung durch die Composition API eine deklarative und testfreundliche Logikstruktur [30].

2.5.4 Beispielhafte Unit-Test-Spezifikation mit Vue Test Utils

Zur strukturierten Validierung einzelner Komponenten eignet sich das Framework *Vue Test Utils*. Nachfolgend wird exemplarisch ein Unit-Test für die Komponente `PlantCard.vue`

dargestellt, der die korrekte Darstellung des Pflanzennamens überprüft:

```

1 import { mount } from '@vue/test-utils'
2 import PlantCard from '@/components/PlantCard.vue'
3
4 describe('PlantCard', () => {
5     it('zeigt den Pflanzennamen korrekt an', () => {
6         const wrapper = mount(PlantCard, {
7             props: { name: 'Ficus lyrata' }
8         })
9         expect(wrapper.text()).toContain('Ficus lyrata')
10    })
11})

```

Listing 2.1: Beispielhafter Unit-Test mit Vue Test Utils

Dieser Ansatz wurde im vorliegenden Proof of Concept (POC) nicht umgesetzt, stellt jedoch ein zentrales Element moderner Qualitätssicherung in Vue-basierten Projekten dar [30].

2.5.5 Beispielhafter Cypress-Test zur End-to-End-Verifikation

Für umfassende Systemtests eignet sich *Cypress* als Werkzeug zur End-to-End-Verifikation. Es ermöglicht die Simulation realer Benutzerinteraktionen über den gesamten Technologie-Stack hinweg. Im Folgenden ist ein Beispieltest für das Anlegen einer Pflanze dargestellt:

```

1 describe('Plant hinzufuegen', () => {
2     it('oeffnet das Formular und speichert eine neue Pflanze', () => {
3         cy.visit('/plants')
4         cy.contains('Pflanze hinzufuegen').click()
5         cy.get('input[name="name"]').type('Monstera')
6         cy.get('select[name="room"]').select('Wohnzimmer')
7         cy.get('button[type="submit"]').click()
8         cy.contains('Monstera').should('exist')
9     })
10})

```

Listing 2.2: Beispielhafter Cypress-Test

Ein vollständiger Cypress-Test wurde im Rahmen des Prototyps nicht umgesetzt, kann aber als Vorlage für spätere Implementierungen dienen [31].

2.5.6 Codeabdeckung und Testmetriken

Zur Überprüfung der Testabdeckung bietet sich der Einsatz von `@vitest/coverage` an. Es berechnet Kennzahlen wie *Statements Covered*, *Branches* und *Line Coverage*. Diese Methode wurde im Rahmen des POC nicht implementiert, wäre jedoch für ein Produktivsystem ein relevanter Bestandteil der Qualitätssicherung.

2.5.7 Einordnung und Fazit

Obwohl Softwaretests einen entscheidenden Beitrag zur Qualitätssicherung leisten, wurden im vorliegenden Projekt keine automatisierten Tests implementiert. Der Grund dafür liegt in der prototypischen Natur der Anwendung als Proof-of-Concept (PoC), wodurch der Fokus auf Funktionalität und Benutzerfluss lag. Insbesondere zeitliche und ressourcenbezogene Einschränkungen sprechen in frühen Entwicklungsphasen oftmals gegen einen vollständigen Testaufbau. Gleichwohl ist festzuhalten, dass insbesondere bei der Weiterentwicklung oder einer produktiven Nutzung automatisierte Tests ein integraler Bestandteil des Entwicklungsprozesses sein sollten.

2.6 REST APIs: Grundlagen und Best Practices

Representational State Transfer (REST) Application Programming Interface (API) ist ein Architekturstil für verteilte Systeme, insbesondere für Webanwendungen. Er wurde erstmals im Jahr 2000 von Roy Thomas Fielding in seiner Dissertation eingeführt. REST definiert eine Reihe von Prinzipien, die die Interaktion zwischen Clients und Servern in einem verteilten System standardisieren und vereinfachen sollen. Obwohl es keine offizielle Spezifikation wie einen Request for Comments (RFC) oder eine ISO-Norm für REST gibt, hat sich der Architekturansatz in der Praxis durchgesetzt und bildet die Grundlage für viele der heutigen Web-APIs.

REST basiert auf dem Prinzip, dass ein Webdienst über eine standardisierte Schnittstelle (API) ansprechbar ist, bei der die Kommunikation zwischen Client und Server zustandslos ist. Das bedeutet, dass jede Anfrage vollständig ist und keine Informationen über den vorherigen Zustand benötigt werden. Diese Eigenschaft macht REST-APIs besonders skalierbar und flexibel.

2.6.1 Warum REST?

REST hat sich gegenüber anderen Architekturansätzen wie Simple Object Access Protocol (SOAP) aus mehreren Gründen durchgesetzt. REST-APIs sind leichter zu implementieren und zu nutzen, da sie auf den bestehenden HTTP-Standards aufbauen. REST nutzt die standardmäßigen HTTP-Verben (GET, POST, PUT, DELETE), um Create Read Update Delete (CRUD)-Operationen auf Ressourcen durchzuführen. Diese Einfachheit und die Nutzung bewährter Webstandards machen REST-APIs besonders attraktiv für Webanwendungen und mobile Apps, wo schnelle Entwicklung und hohe Performance entscheidend sind.

Ein weiterer Vorteil von REST ist seine Flexibilität und die Möglichkeit zur Integration

in eine Vielzahl von Plattformen und Programmiersprachen. Da REST-APIs auf dem HTTP-Protokoll basieren, können sie in nahezu jeder Umgebung eingesetzt werden, die HTTP unterstützt, was zu einer breiten Akzeptanz und Nutzung geführt hat.

2.6.2 Grundlagen einer REST API

Obwohl es kein formales Regelwerk für REST gibt, haben sich in der Entwicklergemeinschaft einige Best Practices etabliert, die eine REST-API als gut definieren. Diese Best Practices sind weitgehend anerkannt und werden häufig in der Praxis angewendet:

JavaScript Object Notation (JSON) als Standardformat verwenden: REST-APIs sollten JSON als Standardformat für die Datenübertragung verwenden, da JSON leichtgewichtig, gut lesbar und in den meisten Programmiersprachen nativ unterstützt wird.

Verwendung von Substantiven in Endpunktpfaden: Endpunktpfade sollten Substantive anstelle von Verben verwenden, um die Ressource zu definieren, auf die die Operation angewendet wird. Beispielsweise sollte der Endpunkt `/users` anstelle von `/getUsers` verwendet werden.

Logische Verschachtelung von Endpunkten: Endpunkte sollten logisch verschachtelt sein, um die Hierarchie der Daten widerzuspiegeln. Zum Beispiel könnte ein Endpunkt für die Bestellungen eines Benutzers `/users/userId/orders` lauten.

Fehlerbehandlung und Standard-HTTP-Fehlercodes: Eine gute REST-API sollte standardisierte HTTP-Fehlercodes verwenden, um dem Client klare Rückmeldungen über den Status der Anfrage zu geben. Beispielsweise steht der Fehlercode 404 für „Nicht gefunden“ und 500 für „Interner Serverfehler“.

Filtern, Sortieren und Paginierung: REST-APIs sollten die Möglichkeit bieten, Ergebnisse zu filtern, zu sortieren und zu paginieren, um die Rückgabemenge zu steuern und die Effizienz zu erhöhen.

Sicherheitspraktiken: REST-APIs sollten sichere Authentifizierungs- und Autorisierungsmechanismen verwenden, wie OAuth2 oder JSON Web Token (JWT), um sicherzustellen, dass nur berechtigte Benutzer auf die Ressourcen zugreifen können.

Daten-Caching: Um die Leistung zu verbessern, sollten REST-APIs Daten zwischenspeichern, wo es sinnvoll ist. Dies kann die Ladezeiten reduzieren und die Last auf dem Server verringern.

API-Versionierung: Eine gute REST-API sollte versioniert werden, um Änderungen und Verbesserungen an der API zu ermöglichen, ohne bestehende Clients zu beeinträchtigen.

Diese Best Practices bilden die Grundlage für die Entwicklung robuster und skalierbarer REST-APIs. Sie wurden von der Entwicklergemeinschaft, beispielsweise auf Plattformen wie StackOverflow [32], breit akzeptiert und weiter verfeinert.

2.6.3 Vertiefende Empfehlungen und Designansätze

Neben diesen grundlegenden Prinzipien bietet das Buch „REST API Design Rulebook“ von Mark Masse [33] eine weitergehende Sammlung von Designregeln. Diese basieren auf den ursprünglichen Prinzipien von Fielding und wurden im Laufe der Zeit durch die praktische Erfahrung ergänzt und weiterentwickelt. Masse betont beispielsweise, dass eine REST-API entworfen und nicht einfach nur codiert wird. Der Entwurfsprozess sollte klar strukturierte Ressourcen und deren Beziehungen beinhalten, um eine konsistente und verständliche API zu schaffen.

Darüber hinaus wird empfohlen, eine REST-API grafisch zu visualisieren, um Entwicklern und Nutzern der API eine klare Vorstellung von den verfügbaren Endpunkten und deren Beziehungen zu geben. Diese Visualisierung erleichtert das Verständnis der API und fördert die Konsistenz in der Implementierung.

2.6.4 Umsetzung für Sensora

Da Sensora von vielen verschiedenen Softwarelösungen der Sensora-Community genutzt werden soll, ist eine gut durchdachte REST-API erforderlich, die den oben genannten Best Practices folgt. Die API wird gemäß dem OpenAPI-Standard dokumentiert, was allen Entwicklern detaillierte Informationen über die Funktionsweise und Möglichkeiten bietet. Diese Standardisierung erleichtert die Implementierung und fördert die Konsistenz in der Nutzung der API.

Zur Visualisierung und Dokumentation der API wird Swagger verwendet, ein weit verbreitetes Tool, das es ermöglicht, die API grafisch darzustellen und interaktive Dokumentationen zu erstellen. Dies stellt sicher, dass Entwickler schnell und einfach auf die notwendigen Informationen zugreifen können, um die Sensora-API effizient in ihre Anwendungen zu integrieren.

Die sorgfältige Umsetzung der REST-Prinzipien in der Sensora-API wird dazu beitragen, eine robuste, flexible und benutzerfreundliche Schnittstelle zu schaffen, die den Produktanforderungen gerecht wird und eine breite Akzeptanz unter den Entwicklern der Sensora-Community findet.

2.7 Designprinzipien und -muster

Die Entwicklung von Sensorsa basiert auf einer klar definierten und bewährten Architektur, die die spezifischen Anforderungen an eine flexible, skalierbare und sichere Benachrichtigungsplattform erfüllt. Im Rahmen dieser Entwicklung wurden verschiedene Designprinzipien und -muster berücksichtigt, die sicherstellen, dass das System nicht nur leistungsfähig, sondern auch leicht wartbar und zukunftssicher ist.

2.7.1 Prinzipiengeleitetes Design

Sensorsa wurde unter strikter Beachtung der Coding-Guidelines entwickelt, einer Reihe von spezifischen Richtlinien, auf die sich die Entwickler geeinigt haben. Diese Guidelines werden an oberster Stelle während des gesamten Entwicklungsprozesses befolgt, um eine einheitlich hohe Qualität in der Architektur und im Code zu gewährleisten. Die API wurde gemäß den Best Practices gestaltet, um eine nahtlose Integration in die Softwarelandschaft des Gesamtprodukts zu gewährleisten. Diese Integration wurde durch die strikte Einhaltung der Prinzipien zur API-Entwicklung, einschließlich der Nutzung von REST für synchrone und asynchronen Kommunikation, ermöglicht. Zudem wurden Sicherheitsprinzipien, insbesondere das Zero Trust-Modell, konsequent umgesetzt, wodurch jede Anfrage eines Clients neu authentifiziert wird.

2.7.2 Coding-Guidelines

Die Coding-Guidelines stellen ein umfassendes Regelwerk dar, das die Grundlage für die Softwareentwicklung bildet. Diese Prinzipien sind mehr als nur Richtlinien; sie sind integraler Bestandteil der Projektkultur und stellen sicher, dass alle Entwicklungsprojekte nach denselben hohen Standards durchgeführt werden. Die Coding-Guidelines decken eine Vielzahl von Aspekten ab, von der Architektur über die Server-Nutzung bis hin zur Sicherheit und Best Practices für spezifische Programmiersprachen wie Python oder Rust.

Ein zentrales Prinzip ist das der losen Kopplung. Es fordert, dass Anwendungen nur über sprachunabhängige Protokolle miteinander kommunizieren. Dies fördert die Modularität und erleichtert die Wartung sowie die Integration neuer Systeme. In Sensorsa wird dies durch die strikte Trennung der Module und die Nutzung von klar definierten Schnittstellen erreicht. Jedes Modul ist eigenständig und kommuniziert über definierte APIs, was die Austauschbarkeit und Wiederverwendbarkeit der Komponenten sicherstellt.

Das Zero Trust-Sicherheitsmodell ist ein weiteres kritisches Element der Coding-Guidelines. Es besagt, dass keine Entität – weder Benutzer noch Gerät oder Netzwerk – automatisch vertraut wird, unabhängig davon, ob sie sich innerhalb oder außerhalb des Netzwerks

befindet. Stattdessen wird jede Anfrage verifiziert. Authentifizierung und Autorisierung erfolgen kontinuierlich. Zugriffe werden auf das Minimum beschränkt, basierend auf dem Prinzip der geringsten Privilegien. Diese Sicherheitsanforderung wurde in Sensora durch die Integration von JWTs umgesetzt, sodass jede Anfrage an einen Service strengen Authentifizierung unterliegt.

Abgerundet werden die Coding-Guidelines durch eine starke Fokussierung auf Best Practices und Clean Code. Diese umfassen spezifische Regeln für die Nutzung von Rust, wie beispielsweise die Bevorzugung von etablierten Bibliotheken wie `acti_web`, die Verwendung von `match`-Ausdrücken statt verschachtelter `if`-Bedingungen und die Anwendung des `?`-Operators zur Verbesserung der Lesbarkeit und Sicherheit des Codes. Diese Praktiken tragen dazu bei, dass der Code von Sensora nicht nur funktional, sondern auch wartbar und erweiterbar ist.

Durch die strikte Beachtung der Coding-Guidelines bei der Entwicklung von Sensora konnte ein System geschaffen werden, das nicht nur den hohen technischen Anforderungen entspricht, sondern auch die langfristigen Ziele in Bezug auf Nachhaltigkeit, Sicherheit und Effizienz unterstützt. Diese Prinzipien sind somit ein wesentlicher Bestandteil der Architektur und des Designs von Sensora und bilden das Fundament für alle getroffenen Entscheidungen während der Entwicklung.

2.7.3 Event-Driven Architecture

Ein zentrales architektonisches Muster, das bei der Entwicklung von Sensora bewusst gewählt wurde, ist die Event-Driven Architecture. Dieses Muster eignet sich besonders gut für die Verarbeitung von Messungen, die asynchron generiert und verteilt werden müssen. Die Entscheidung für eine Event-Driven Architecture ermöglicht es, auf Ereignisse in Echtzeit zu reagieren und Daten effizient zu verteilen, ohne die Performance des Systems zu beeinträchtigen. Solace, das als Messaging-System innerhalb von Sensora eingesetzt wird, spielt hierbei eine entscheidende Rolle, indem es stabile und zuverlässige asynchrone Kommunikation sicherstellt.

2.7.4 Herausforderungen und Lösungen

Eine der zentralen Herausforderungen bei der Implementierung der Event-Driven Architecture war die asynchrone Benachrichtigung der Clients. Diese Herausforderung wurde erfolgreich durch die Integration von Solace gemeistert, das als stabiles und zuverlässiges Messaging-System fungiert. Ein weiteres potenzielles Problem, nämlich die Handhabung von Zugriffskollisionen bei gleichzeitigen Datenbankzugriffen, wurde durch die Verwendung von PostgreSQL adressiert. PostgreSQL bietet ein fortschrittliches Session-

Management, das automatisch Kollisionen bei gleichzeitigen Zugriffen verwaltet, sodass Sensora selbst keine zusätzlichen Mechanismen zur Kollisionsvermeidung implementieren musste.

2.7.5 Zusammenfassung

Die Entwicklung von Sensora basiert auf einer durchdachten Kombination aus bewährten Designprinzipien und modernen Architekturmustern. Durch die konsequente Anwendung der Coding-Guidlines und die Nutzung einer Event-Driven Architecture wurde ein System geschaffen, das sowohl leistungsfähig als auch flexibel ist. Die Berücksichtigung von Best Practices und die Umsetzung eines strengen Sicherheitsmodells gewährleisten, dass Sensora nicht nur den aktuellen Anforderungen gerecht wird, sondern auch zukunftssicher und erweiterbar bleibt.

2.8 Solace PubSub+ Event Broker

Solace PubSub+ ist ein Enterprise-Messaging-Broker, der als zentrales Vermittlungssystem in einer ereignisgesteuerten Architektur dient. Er ermöglicht die asynchrone Kommunikation zwischen verteilten Anwendungen über ein publish/subscribe-basiertes Paradigma. Nachfolgend werden Aufbau und Hauptmerkmale dieses Brokers erläutert – insbesondere die Nutzung von Topics und Queues, die Unterstützung des MQTT-Protokolls, Mechanismen zur Zugriffskontrolle (ACLs) sowie die Konfiguration über die SEMP-API. Abschließend erfolgt ein Vergleich mit anderen Messaging-Lösungen.

2.8.1 Aufbau eines Message Brokers und Solace-Architektur

Ein Message Broker wie Solace PubSub+ fungiert als Vermittler zwischen Sendern und Empfängern von Nachrichten. Publisher schicken Nachrichten an den Broker, welcher diese anhand von Metadaten (meist Topics) an interessierte Subscriber weiterleitet. Dadurch sind Publisher und Subscriber entkoppelt – weder muss der Publisher die Empfänger kennen, noch der Subscriber die Quelle der Nachricht [34]. Diese Entkopplung erhöht die Skalierbarkeit und Flexibilität des Systems erheblich. Der Broker übernimmt Verantwortung für das Routing, Filtering und ggf. Persistieren der Nachrichten.

Solace PubSub+ implementiert dieses Prinzip in einer *Event Broker*-Architektur, die auf hohe Durchsatzraten und viele parallele Verbindungen ausgelegt ist. Eine Besonderheit von Solace ist die Unterstützung mehrerer Protokolle innerhalb desselben Brokers. So können Clients über unterschiedliche offene Standards wie AMQP 1.0, MQTT 3.1.1/5.0,

REST (HTTP) oder JMS 1.1 mit dem Broker kommunizieren, ohne Gateway oder Übersetzer [35]. Diese Multi-Protokoll-Fähigkeit erlaubt z.B., dass IoT-Geräte via MQTT Daten publizieren, während Backend-Systeme dieselben Daten über JMS oder WebSockets abonnieren – der Broker überbrückt dabei transparent die Protokolle.

Intern organisiert Solace den Nachrichtenraum in sogenannten *Message VPNs* (Virtual Private Networks). Ein Message VPN ist ein logisch isolierter Kontext innerhalb eines Brokers, der Mandanten-Trennung ermöglicht. Alle folgenden Objekte wie Topics, Queues, Clients und aac1-Profile sind immer innerhalb eines bestimmten VPN definiert. Der Broker kann somit mehrere VPNs parallel betreiben, die wie separate virtuelle Broker fungieren.

Abbildung 2.1 zeigt schematisch die grundlegende Funktionsweise eines Pub/Sub-Brokers. Publisher senden Ereignisse an den Broker, der diese basierend auf Topic-Filtern an die verbundenen Subscriber verteilt. Solace unterscheidet hierbei zwei Qualitäten von Nachrichtenlieferung: zum einen flüchtige *Direct Messages*, die nichtpersistiert werden (QoS 0/1, „non-persistent“), zum anderen *Guaranteed Messages*, die auf dem Broker gespeichert und bei Bedarf erneut zugestellt werden (QoS 1/2, „persistent“). Erstere bieten maximale Geschwindigkeit und werden nur an aktuell verbundene Empfänger zugestellt; Letztere werden in sogenannten *Queues* vorgehalten, bis ein Empfänger sie abruft, wodurch keine Nachrichten verloren gehen [36]. Durch diese Zweiteilung kann ein Event Broker sowohl hochvolumige Echtzeitdaten (z.B. Marktdaten-Feeds) effizient verteilen, als auch kritische Nachrichten zuverlässig Zustellen, selbst wenn Empfänger vorübergehend offline sind.

Topic: sensora/v1/send/controller123

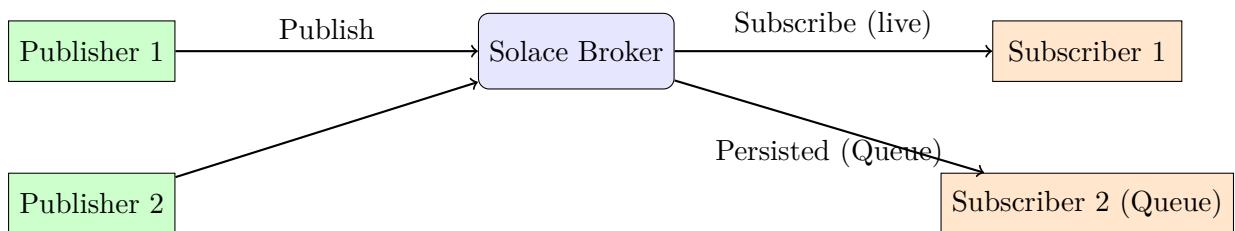


Abbildung 2.1: Schematische Darstellung eines Publish/Subscribe Event Brokers mit Solace.

2.8.2 Topics und Queues in Solace PubSub+

Solace PubSub+ verwendet ein hierarchisches Topic-System zur Adressierung von Nachrichten. Topics sind benannte Kanäle, die typischerweise durch mittels Schrägstrich / getrennte Hierarchieebenen strukturiert sind (z.B. Sensor/Temperatur/Aussen). Publisher veröffentlichen Nachrichten auf einem bestimmten Topic, und Subscriber können sich auf Topics (bzw. Muster davon) abonnieren. Eine Stärke von Solace ist, dass Topics *dynamisch* und ohne Vorab-Konfiguration genutzt werden können – der Broker akzeptiert beliebige Topic-Strings und leitet Nachrichten entsprechend weiter. Für die Filterung unterstützen Topics Platzhalter: Das Zeichen * steht für genau ein Hierarchie-Level (ein Wort zwischen den Trennzeichen), während > als Wildcard für beliebig viele nachfolgende Hierarchieebenen fungiert. Beispielsweise würde ein Subscriber auf Topic Sensor/Temperatur/* alle Temperaturwerte aller Sensoren erhalten, während Sensor/> sämtliche Nachrichten im Topic-Pfad beginnend mit Sensor matcht. Diese flexiblen Wildcards erlauben eine inhaltliche Filterung der Ereignisse bereits auf Broker-Seite.

Neben der direkten Verteilung über Topics bietet Solace die Möglichkeit, Topics mit *Queues* zu verknüpfen. Eine *Queue* ist ein benannter, persistentierender Endpunkt auf dem Broker, der Nachrichten speichern kann. Queues dienen primär der Realisierung von Lastverteilung und Persistenz (Point-to-Point-Messaging): Subscriber können eine exklusive Verbindung zu einer Queue aufbauen und erhalten die dort gespeicherten Nachrichten in der Reihenfolge ihres Eingangs. Solace verknüpft nun das Topic- und Queue-Konzept dadurch, dass man einer Queue eine oder mehrere Topic-Abonnements geben kann. Die Queue fungiert damit als langlebiger Subscriber auf die entsprechenden Topics. Publiziert ein Publisher eine Nachricht auf einem solchen Topic mit persistenter Delivery-Mode (Persistent Message), so wird die Nachricht in der zugeordneten Queue abgelegt und steht dort zur Abholung bereit. Dieses Konzept erlaubt es, die Pub/Sub-Semantik mit der Zuverlässigkeit von Queues zu kombinieren. So kann man etwa erreichen, dass eine bestimmte Ereignisart (Topic) an mehrere unabhängige Dienste zugestellt wird, indem man pro Dienst eine eigene Queue mit demselben Topic-Subscription definiert – die Nachricht wird vom Broker dupliziert und landet in allen relevanten Queues (Fan-Out auf persistente Empfänger). Im Solace-System sind Queues standardmäßig langlebig und können Nachrichten auch über Broker-Neustarts hinweg speichern; optional können sie als *exklusiv* (nur ein Verbraucher gleichzeitig) oder *non-exklusiv* (mehrere konkurrierende Verbraucher) konfiguriert werden. [37].

Im Vergleich zu anderen Messaging-Brokern vereinheitlicht Solace damit das sonst oft getrennte Modell von Topics (für Pub/Sub) und Queues (für Punkt-zu-Punkt) zu einem flexiblen Konstrukt. Während z.B. in RabbitMQ Nachrichten immer an Exchanges gesendet und dann per Binding-Key auf Queues verteilt werden müssen, kann in Solace

direkt auf einem Topic publiziert werden, das entweder von Clients abonniert oder von Queues “belauscht” wird [37]. Auch JMS unterscheidet zwischen Topic und Queue als verschiedenen Destination-Typen; Solace als JMS-Provider mappt jedoch JMS-Topics und -Queues intern auf das gleiche einheitliche Event-Model, was Verwaltung und Integration erleichtert.

2.8.3 MQTT und topic-basiertes Routing

Das *Message Queuing Telemetry Transport* (MQTT) Protokoll ist ein offener Standard für Leichtgewichts-Messaging, der vor allem im IoT-Umfeld verbreitet ist. Solace PubSub+ unterstützt MQTT in den Versionen 3.1.1 und 5.0 gemäß dem OASIS-Standard [38]. MQTT-Clients können sich mit dem Broker verbinden und Nachrichten auf MQTT-Topics publizieren bzw. abonnieren. Ein wesentliches Konzept von MQTT ist das topic-basierte Routing – analog zum oben beschriebenen Topic-System. MQTT-Themenpfade sind ebenfalls hierarchisch mit / aufgebaut; als Wildcards dienen +(ein Level) und #(mehrere Level). Solace setzt diese MQTT-Semantik um und übersetzt sie intern auf das eigene Topic-Format. Beispielsweise entspricht ein MQTT-Subscription auf `sensors/+humidity` einem Solace-Topicfilter `sensors*/humidity`. Dadurch können MQTT-Publisher und -Subscriber nahtlos mit anderen Solace-Clients interagieren. So ist es möglich, dass ein Gerät via MQTT auf `sensors/room1/humidity` sendet, während ein Java-Anwendung über JMS das Topic `sensors/>` abonniert und somit die Nachricht erhält – der Broker übernimmt die Protokollkonvertierung und das Routing [35].

MQTT sieht per Spezifikation keine expliziten Queues vor; wenn ein MQTT-Subscriber offline geht (und QoS 1 oder 2 verwendet), puffert der Broker die Nachrichten in einer Session-Queue, die beim Wiederverbinden ausgeliefert werden. Solace erweitert diese Möglichkeiten insofern, als dass man persistente Weiterleitung an benannte Queues nutzen kann, indem man MQTT-Themen mit Solace-Queues verknüpft (wie zuvor beschrieben). Rein mit MQTT-Mitteln kann ein Device also nur mittels Durable Session (Clean Session = false) Nachrichten nachliefern lassen, während Solace auch langfristige Persistenz über MQTT hinaus bietet. Allerdings gelten Protokolleinschränkungen: z.B. unterstützt MQTT v3.1.1 keine Bearbeitung von auf einer Queue gespeicherten Nachrichten, da das Konzept im Protokoll fehlt. Daher wird in Szenarien mit Bedarf an fortgeschrittenem Queueing ggf. ein anderes Protokoll (AMQP 1.0 oder Solaces eigenes API/SMF) verwendet [39]. Nichtsdestotrotz unterstreicht die MQTT-Fähigkeit von Solace die Flexibilität des Brokers, heterogene Clients in ein gemeinsames Topic-basiertes Kommunikationsgeflecht einzubinden.

2.8.4 Access Control Lists (ACLs)

In einem Multi-User-Messaging-System ist feingranulares Zugriffsmanagement essenziell, um Sicherheit und Mandantentrennung zu gewährleisten. Solace PubSub+ bietet hierzu *Access Control List (ACL) Profiles* an, mit denen reguliert wird, welche Aktionen ein Client ausführen darf. Ein aacl-Profil bestimmt erstens, ob sich ein Client überhaupt mit dem Broker/Message-VPN verbinden darf, und zweitens, welche Topics er publizieren oder abonnieren darf [40]. Konkret bestehen aacl-Profile aus Regeln, die Verbindungsversuche sowie Topic-Zugriffe kontrollieren. Für jede dieser Kategorien (Connect, Publish-Topic, Subscribe-Topic, sowie Subscribe auf Shared-Queues) lässt sich ein Standardverhalten definieren – entweder *allow* (erlauben) oder *deny* (standardmäßig verweigern) – und es können Ausnahmeregeln (*exceptions*) für spezifische Topics oder Topic-Muster hinzugefügt werden.

Jedem Client, der sich authentifiziert, wird ein aacl-Profil zugewiesen (entweder direkt am Client-Username konfiguriert oder – falls LDAP-Authentifizierung genutzt wird – über Gruppenzuordnung). Nach erfolgreicher Authentisierung prüft der Broker sämtliche Aktionen des Clients gegen die aacl des zugewiesenen Profils. Beispielsweise kann ein aacl-Profil erlauben, dass der Client nur auf Topics unterhalb von ClientA/> publizieren darf und alle anderen Topics geblockt werden. Versucht der Client eine Nachricht auf ein nicht erlaubtes Topic zu senden oder eine unberechtigte Subscription anzulegen, verweigert der Broker die Operation. Auf diese Weise können unterschiedliche Clients auf unterschiedliche Topic-Räume beschränkt werden, was im IoT-Umfeld z.B. dazu genutzt wird, dass jedes Gerät nur auf seinem eigenen Topic-Präfix senden/empfangen darf [41]. aacl-Profile sind damit ein zentrales Sicherheitsmerkmal, um unerlaubten Datenzugriff zu verhindern und eine saubere Mandantentrennung innerhalb eines Brokers (bspw. zwischen verschiedenen Anwendungen im selben Message VPN) sicherzustellen.

2.8.5 Konfiguration über die SEMP-API

Die Administration eines Solace-Brokers kann neben grafischen Tools und dem CLI auch vollständig automatisiert über eine REST-basierte Managementschnittstelle erfolgen – die *Solace Element Management Protocol API* in Version 2 (**SEMP v2**). SEMP v2 stellt eine REST/HTTP-JSON API bereit, mit der sich der Broker programmatisch konfigurieren und überwachen lässt [42]. Über definierte Endpunkte können Management-Clients z.B. Queues oder Topic-Endpunkte anlegen, aacl-Profile definieren, den Status von Clients und VPNs abfragen oder Verwaltungsaktionen (wie das Leeren einer Queue) durchführen. Die SEMP-API ist in drei Bereiche gegliedert:

- **Config:** Änderung und Abfrage der Konfigurationsobjekte

(z.B. POST /SEMP/v2/config/msgVpns/{vpn}/queues zum Anlegen einer Queue).

- **Monitor:** Lesezugriff auf betrieblichen Status und Metriken (z.B. Statistiken einer Queue, Anzahl der Clients etc.).
- **Action:** Auslösen von Aktionen, die keinen dauerhaften Konfigurationszustand ändern (z.B. *Purge* einer Queue, Failover umschalten).

SEMP v2 ist via OpenAPI-Spezifikation beschrieben und ermöglicht so die automatisierte Generierung von Client-Bibliotheken [42]. In der Praxis kann somit z.B. ein Auth-Service mittels SEMP beim Deployment sicherstellen, dass benötigte Topics, Queues und acacs im Broker eingerichtet sind, ohne manuell die Admin-Oberfläche bedienen zu müssen. Die Verwendung einer Management-API erleichtert die Integration des Brokers in Infrastructure-as-Code und DevOps-Prozesse. Insbesondere in Cloud-Umgebungen mit dynamischer Skalierung kann der Broker so automatisiert auf neue Services oder geänderte Sicherheitsrichtlinien konfiguriert werden.

2.8.6 Vergleich mit anderen Messaging-Systemen

Solace PubSub+ steht in Konkurrenz zu einer Reihe von Messaging-Systemen, die teils unterschiedliche Architekturen aufweisen. Im Folgenden werden einige charakteristische Unterschiede kurz herausgestellt: **RabbitMQ:** RabbitMQ ist ein verbreiteter Open-Source Message Broker, der das AMQP 0.9.1 Protokoll verwendet. Ähnlich wie Solace unterstützt RabbitMQ Publish/Subscribe und Queueing, jedoch meist nur über AMQP (Multi-Protokoll-Fähigkeit ist bei RabbitMQ nur über Plugins begrenzt möglich). RabbitMQ nutzt ein Exchange-Queue-Modell: Publisher senden an Exchanges, welche die Nachrichten anhand von Routing-Keys an Queues binden. Solace hingegen erlaubt das direkte Publizieren auf Topics, die intern sowohl von Abonnements als auch von Queues “gelesen” werden können [37]. In Solace ist somit das Routing-Konzept etwas einfacher, da kein separater Exchange-Schritt sichtbar ist. Bezuglich Performance skaliert Solace durch seine optimierte Architektur und optional spezialisierte Hardware-Appliances auf sehr hohe Durchsätze, während RabbitMQ für viele Anwendungsfälle ausreichend, aber in extremen Szenarien früher an Grenzen stoßen kann. RabbitMQ bietet dafür eine große Flexibilität durch Plugins (etwa für unterschiedliche Auth-Backends, Shovel/Federation für verteilte Setups usw.), wohingegen Solace viele Funktionen nativ integriert (etwa eingebaute Hochverfügbarkeit, Replay-Funktionalität, etc.). **Apache Kafka:** Kafka verfolgt einen anderen Ansatz als klassische Message Broker. Es ist eigentlich eine verteilte Commit-Log-Plattform, die primär für Streaming von Datenströmen und Event Sourcing konzipiert ist. Im Gegensatz zu Solace, das Nachrichten direkt an Subscriber weiterleitet oder zwischengeschaltet, schreibt Kafka alle Nachrichten fortlaufend in Partitionen auf Disk. Consumer

lesen diese logischen Partitionen in eigenem Tempo und behalten Zustände wie den Lese-Offset. Dadurch eignet sich Kafka hervorragend für das Replaying vergangener Events und das Verteilen großer Datenmengen auf viele Konsumenten, hat aber keinen Broker, der aktiv zu Clients zustellt – die Clients holen vielmehr die Daten. Solace dagegen ist auf Live-Zustellung optimiert: Nachrichten werden typischerweise verworfen, wenn kein Abnehmer da ist (außer es existiert eine persistente Queue). Man könnte vereinfacht sagen, Kafka speichert Streams dauerhaft (Standard ist Tage bis unbegrenzt) und garantiert dadurch Zustellung an spätere Konsumenten, während Solace klassischere Messaging-Semantiken bietet, wo persistente Nachrichten gezielt zugestellt und nach Bestätigung gelöscht werden (ähnlich JMS). Für use-cases im Auth-Service-Umfeld (kurzlebige Ereignisse, Befehle, Responses) ist ein Broker wie Solace passender; Kafka spielt seine Stärken bei Analyse- und Loggingdaten aus [43]. **JMS Broker (ActiveMQ/IBM MQ) und Andere:** Solace kann auch als JMS-Provider dienen und vergleichbare Funktionalität wie ActiveMQ, Artemis oder IBM MQ bereitstellen. Gegenüber vielen JMS-Brokern punktet Solace mit integrierter Mehrprotokollfähigkeit und höherer Performance. IBM MQ etwa ist stark auf Transaktionssicherheit und Mainframe-Integration ausgelegt, während Solace auf niedrige Latenz und flexible Verteilung optimiert ist. Moderne Cloud-Messaging-Services (AWS SNS/SQS, Azure Service Bus) trennen oft Pub/Sub und Queuing in separate Services – Solace bietet beides in einem. Insgesamt ist Solace als „Enterprise Event Broker“ positioniert, der in verteilten Architekturen (z.B. Microservices, IoT, Hybrid-Cloud) eine zentrale Kommunikationsdrehscheibe bildet und durch Features wie *Dynamic Message Routing* (für geclusterte Broker-Netze, Event Mesh) und eingebaute Hochverfügbarkeit auszeichnet. Zusammenfassend lässt sich festhalten, dass Solace PubSub+ aufgrund seines architektonischen Designs (vereinheitlichtes Topic/Queue-Model, Multi-Protokoll-Unterstützung, hohe Performance) eine besondere Rolle unter den Messaging-Systemen einnimmt. Die theoretischen Grundlagen dieses Brokers – Entkopplung durch Pub/Sub, Hierarchie-Topics, etc. – bilden die Basis für den praktischen Einsatz im Rahmen des Auth-Service, wo Solace die Nachrichtenkommunikation absichert und steuert.

2.9 Sicherheitsrelevante Technologien im Auth-Service

Im Auth-Service kommen mehrere kryptographische und sicherheitstechnische Verfahren zum Einsatz, um die Authentifizierung und Kommunikation abzusichern. In diesem Abschnitt werden die folgenden Konzepte erläutert: **HMAC** (Hash-based Message Authentication Code) – ein Verfahren zur Gewährleistung von Nachrichtenintegrität und -authentizität. **Fernet-Verschlüsselung** – ein symmetrisches Authenticated-Encryption-Schema zur sicheren Ausgabe von Tokens. **Tokenisierung mit PSK und**

Challenge-Response – das Zusammenspiel von vorab geteilten Schlüsseln (Pre-Shared Keys), Challenge-Response-Authentifizierung und der Verwendung von Tokens im Authentifizierungsprozess. Ziel ist es, die kryptographischen Grundlagen, Funktionsweisen und Sicherheitseigenschaften dieser Techniken darzustellen, um das Verständnis für ihre Umsetzung im Auth-Service zu schaffen.

2.9.1 HMAC: Hash-basierter Message Authentication Code

Ein *Message Authentication Code* (MAC) ist eine Prüfsumme, die unter Verwendung eines geheimen Schlüssels berechnet wird und es Empfängern ermöglicht, die Integrität und Authentizität einer Nachricht zu verifizieren. Der Sender berechnet einen MAC-Wert über die Nachricht und sendet ihn mit; der Empfänger kann mit dem geteilten Geheimnis denselben MAC berechnen und vergleichen. Stimmen die Werte überein, ist sichergestellt, dass die Nachricht unverändert ist und vom legitimen Sender stammt (solange der Schlüssel geheim bleibt).

HMAC (*Keyed-Hash Message Authentication Code*) ist ein spezielles, sehr weit verbreitetes MAC-Verfahren, das auf kryptographischen Hashfunktionen basiert [44]. Formal lässt sich HMAC definieren als:

$\text{HMAC}_k(m) = H((K \text{ opad}) H((K \text{ ipad}) m))$, wobei H eine kryptographische Hashfunktion (etwa SHA-256) und K ein geheimer Schlüssel ist (typischerweise auf Blockgröße von H aufgefüllt). Die Konstanten $opad$ und $ipad$ sind bitmasken (*outer/inner padding*), die den Schlüssel vor dem Hashen modifizieren. Durch diese zweistufige Konstruktion (Hash innerhalb eines Hashs) werden bestimmte Angriffe auf einfachere MAC-Konstruktionen (wie „Schlüssel+Nachricht hashen“) verhindert. Intuitiv berechnet HMAC zunächst einen Hash der Kombination aus Schlüssel und Nachricht und hasht dieses Ergebnis nochmals zusammen mit dem Schlüssel.

HMAC wurde 1996 von Bellare, Canetti und Krawczyk entwickelt und analysiert. In RFC 2104 ist HMAC als generisches Konstrukt spezifiziert, das mit jeder iterativen Hashfunktion (MD5, SHA-1, SHA-2 etc.) genutzt werden kann [44]. So verwendet HMAC-SHA256 beispielsweise die Secure Hash Algorithm 256-bit Funktion als H . Die Sicherheit von HMAC hängt von den Eigenschaften der zugrundeliegenden Hashfunktion ab, insbesondere deren Kollisionsresistenz und Unberechenbarkeit [44]. Unter typischen Annahmen lässt sich zeigen, dass HMAC annähernd ein *pseudorandom function (PRF)* ist, solange H keine gravierenden Schwächen aufweist – informell bedeutet dies, dass ohne Kenntnis des Schlüssels kein effizienter Weg bekannt ist, um den korrekten MAC für eine neue Nachricht zu erzeugen oder eine gültige Nachricht+MAC zu modifizieren. HMAC ist zudem resistent gegen sogenannte *Längenextensions-Angriffe*, die bei einfachen Hash-MACs auftreten könnten, weil durch die Verwendung von $opad/ipad$ die innere Struktur des

Hashs gesichert wird.

Aufgrund seiner starken Sicherheitseigenschaften und Einfachheit ist HMAC in vielen Protokollen und Standards fest verankert. Das US-amerikanische NIST standardisierte HMAC in FIPS 198-1 [45]. In TLS und IPsec dient HMAC (mit SHA-256 oder SHA-384) als Nachrichten-Authentifizierungsmechanismus, um die Integrität von übertragenen Daten sicherzustellen. Auch viele Anwendungsschichten nutzen HMAC: Beispielsweise beruht der HOTP-Algorithmus (HMAC-based One-Time Password) für Einmalpasswörter auf einem HMAC über einen Zähler [46], und JSON Web Tokens (JWT) können mit einem HMAC-SHA256 (HS256) Signaturteil versehen sein, um Tokenintegrität zu gewährleisten.

Im Kontext des Auth-Service wird HMAC insbesondere im Rahmen eines *Challenge-Response*-Verfahrens eingesetzt. Hierbei teilen sich der Client (z.B. ein IoT-Gerät) und der Server einen geheimen Schlüssel K (siehe „Pre-Shared Key“ unten). Zur Authentifizierung sendet der Server dem Client eine Zufallsherausforderung (Nonce) – typischerweise eine ausreichend lange zufällige Bitfolge N . Der Client berechnet daraufhin den HMAC-Wert über diese Challenge: $R = \text{HMAC}_K(N)$. Dieser Response-Wert R wird an den Server zurückgesendet. Da der Server ebenfalls K kennt, kann er seinerseits $\text{HMAC}_K(N)$ berechnen und prüfen, ob das Ergebnis mit dem empfangenen R übereinstimmt. Ist dies der Fall, beweist der Client damit, dass er den geheimen Schlüssel besitzt (denn nur mit K lässt sich der korrekte MAC erzeugen). Gleichzeitig wird die Integrität der Challenge gewährleistet – eine Manipulation von N unterwegs würde zu einem falschen MAC führen. Wichtig ist, dass für jede neue Authentifizierung eine frische, unvorhersehbare Challenge N gewählt wird, um Wiederholungsangriffe (Replay) zu verhindern. Dieses einfache Challenge-Response-Schema bietet eine zuverlässige Einweg-Authentifizierung des Clients gegenüber dem Server, ohne dass der Schlüssel selbst jemals übertragen wird. Die Verwendung von HMAC dafür ist ideal, da HMAC effizient berechenbar ist (auch auf eingeschränkter Hardware) und kryptographisch stark genug, um nicht geraten oder invertiert werden zu können.

Zusammengefasst liefert HMAC also einen grundlegenden Baustein für die Nachrichtenauthentifizierung: mathematisch basiert es auf Hashfunktionen und verknüpft einen geheimen Schlüssel mit der Nachricht zu einem Prüftoken. Seine Sicherheit wurde analytisch untermauert [47] und in Praxisstandards übernommen, was es zu einem vertrauenswürdigen Mechanismus für Integritätsschutz macht. Im Auth-Service bildet HMAC den Kern des Authentifizierungsnachweises im Challenge-Response-Protokoll.

2.9.2 Fernet-Verschlüsselung: Authenticated Encryption für Tokens

Neben der Überprüfung von Nachrichtenintegrität (via MAC) ist oft auch Vertraulichkeit und Schutz vor Manipulation bei gespeicherten oder übertragenen Tokens erforderlich. *Fernet* ist ein hochrangiges kryptographisches Schema, das symmetrische Verschlüsselung mit Integritätschutz kombiniert und dadurch sog. *Authenticated Encryption* bereitstellt. Es wurde ursprünglich als Teil der Python-Kryptographiebibliothek entwickelt, liegt aber als offene Spezifikation vor und ist in mehreren Sprachen implementiert [48]. Fernet garantiert, dass ein verschlüsselter Datenträger (Token) weder gelesen noch unbemerkt verändert werden kann, ohne den geheimen Schlüssel zu besitzen [48].

Technisch basiert Fernet auf anerkannten Standard-Primitiven:

- **Verschlüsselung:** AES im CBC-Modus mit 128-Bit-Schlüssel und PKCS#7 Padding wird verwendet, um die eigentlichen Nutzdaten zu chiffrieren.
- **Integrität/Authentizität:** Ein HMAC über die verschlüsselten Daten (inkl. Header) mit SHA-256 stellt sicher, dass Änderungen erkannt werden.
- **Zufall:** Für jede Verschlüsselung wird ein frischer 128-Bit Initialisierungsvektor (IV) per Kryptozufall erzeugt.

Diese Details sind in der Fernet-Spezifikation festgelegt [49]. Ein Fernet-Schlüssel ist 32 Byte (256 Bit) lang und wird intern in zwei Hälften aufgeteilt: 16 Byte dienen als AES-Schlüssel, 16 Byte als HMAC-Schlüssel [49]. Bei der Token-Erstellung wird folgendermaßen verfahren [49]:

1. Es wird ein 64-Bit-Zeitstempel (Sekunden seit Unix-Epoche) als *Timestamp* festgehalten, der den Erstellungszeitpunkt markiert.
2. Ein zufälliger 16-Byte IV wird generiert.
3. Die Klartextnachricht wird mit AES-128-CBC unter Verwendung des IV und des Verschlüsselungsschlüssels verschlüsselt (nachdem sie mit PKCS7 auf Blocklänge aufgefüllt wurde).
4. Über den sogenannten Token-Header (bestehend aus einer 8-Bit Versionskennung, dem Timestamp und dem IV) *und* den Ciphertext wird ein HMAC-SHA256 mit dem HMAC-Schlüssel berechnet
5. Der finale Fernet-Token besteht aus der Konkatenation von Version, Timestamp, IV, Ciphertext und HMAC. Dieser Byte-String wird zuletzt URL-sicher Base64-kodiert,

um ihn als ASCII-Token darstellbar zu machen. Das Format eines Fernet-Tokens lässt sich schematisch so darstellen:



, wobei C variable Länge (Vielfaches von 128 Bit) hat, abhängig von der Klartextlänge [49]. Die Versionsnummer ist aktuell immer 0x80 (Version 1 der Spezifikation). Der HMAC deckt den gesamten Token ohne das HMAC-Feld ab, d.h. $HMAC = \text{HMAC}(Version \parallel T_{\text{Stamp}} \parallel IV \parallel C)$ [49]. Dadurch wird sicher gestellt, dass sowohl Header als auch Ciphertext geschützt sind – eine Veränderung irgendeines Bits im Token führt zu einem HMAC-Mismatch beim Entschlüsseln.

Fernet ist somit ein klar strukturierter Fall von *Encrypt-then-MAC*: Zuerst werden Daten symmetrisch verschlüsselt, dann wird über das Chiffraut ein MAC gebildet. Diese Konstruktion gilt als sicher (im Sinne von IND-CCA2, also Wahrung von Vertraulichkeit auch gegen adaptive Angreifer), da ein Empfänger immer erst den MAC prüft, bevor er entschlüsselt. Ohne Kenntnis des Schlüssels kann ein Angreifer weder den Ciphertext sinnvoll entschlüsseln noch einen gültigen neuen Token erzeugen, da ihm sowohl der AES- als auch der MAC-Schlüssel fehlen.

Die Verwendung von Standardbausteinen (AES, HMAC) stellt sicher, dass Fernet auf bekannten Sicherheitsannahmen beruht. AES-128 ist ein symmetrischer Blockcipher, der als praktisch sicher gilt; CBC-Modus ist sicher, solange jedes IV nur einmalig verwendet wird (was durch die zufällige Wahl gegeben ist). HMAC-SHA256 ist, wie oben erläutert, ein starker MAC. Die Kombination ergibt einen symmetrischen Authenticated-Encryption-Algorithmus, der konzeptionell ähnlich ist zu anderen AEADs (Authenticated Encryption with Associated Data) wie AES-GCM – allerdings mit dem Unterschied, dass Fernet die Komponenten explizit trennt und kein zusätzliches Associated Data Feature hat. Ein Vorteil von Fernet ist die Einfachheit der Schnittstelle: Entwickler erhalten eine einzige Routine zum Verschlüsseln und Entschlüsseln von Bytestrings, wodurch Implementierungsfehler (wie Vergessen der MAC-Prüfung oder unsichere IV-Wahl) vermieden werden. Die Fernet-Implementierung in der Python-cryptography-Bibliothek z.B. stellt sicher, dass bei Aufruf von `decrypt(token)` automatisch der HMAC geprüft und bei Scheitern eine Exception geworfen wird, sodass kein Entwickler versehentlich mit unverifizierten Daten weiterarbeitet [48].

Für den Auth-Service ist besonders relevant, dass Fernet-Tokens einen eingebetteten Zeitstempel besitzen. Dies erlaubt es, die Gültigkeit von Tokens zeitlich zu begrenzen: Beim Entschlüsseln kann der Server den im Token codierten Erstellungszeitpunkt mit der aktuellen Zeit vergleichen und ein Token ggf. als abgelaufen verwerfen, wenn es älter

als eine definierte TTL (time-to-live) ist. Somit kann ein vom Auth-Service ausgestelltes Token z.B. nur für eine Stunde gültig sein, danach muss sich der Client neu authentifizieren. Fernet selbst erzwingt keine Ablaufprüfung, bietet aber die Information (Timestamp) dafür an.

Ein weiteres Merkmal ist die feste Größe des Schlüssels (256 Bit), was ausreichend Sicherheit bietet. Sollte es nötig sein, Schlüssel zu rotieren, unterstützt Fernet dies durch das Konzept von *Multi-Fernet*, bei dem mehrere Schlüssel (z.B. alter und neuer) parallel akzeptiert werden können [48]. Im Normalfall wird jedoch ein einzelner, sicher verwahrter Key vom Auth-Service verwendet, um alle Token zu signieren/verschlüsseln.

Zusammengefasst stellt Fernet also ein einfach zu handhabendes, aber robustes Verfahren dar, um aus beliebigen sensitiven Daten ein sicheres Token zu erstellen. Es gewährleistet Vertraulichkeit, Integrität und Authentizität der Daten in einem Schritt. Im Kontext von Authentication-Tokens bedeutet dies: Ein Fernet-Token kann z.B. Benutzer- oder Gerätedaten enthalten (wie eine Identifikationsnummer, Berechtigungen, Zeitstempel), und nur der Auth-Service kann diese Daten wieder entschlüsseln und verifizieren. Ein Client, der ein solches Token erhält, kann dessen Inhalt nicht ohne Weiteres auslesen oder verändern – er nutzt es lediglich als „Ausweis“, den er bei weiteren Anfragen vorzeigt.

2.9.3 Tokenisierung, Pre-Shared Key und Challenge-Response im Zusammenspiel

Nachdem nun HMAC als Authentifizierungsnachweis und Fernet als Tokenisierungsmechanismus beschrieben wurden, soll abschließend beleuchtet werden, wie diese Verfahren im Auth-Service zusammenwirken. Der Authentifizierungsablauf lässt sich vereinfacht in folgende Phasen unterteilen: Initialer Schlüsselaustausch (*Pre-Shared Key*), Challenge-Response-Authentisierung und Ausgabe eines Tokens (*Tokenisierung*).

Pre-Shared Key (PSK): Zu Beginn wird angenommen, dass Client und Server einen gemeinsamen geheimen Schlüssel besitzen, der vorgängig sicher verteilt wurde. Ein solcher vorab geteilter Schlüssel (PSK) dient als Identitätsnachweis des Clients. PSK-Verfahren sind im IoT-Bereich üblich, wenn kein komplexes PKI-System für Zertifikate verfügbar oder gewünscht ist [41]. Der PSK muss sicher auf dem Gerät hinterlegt und auf Serverseite gespeichert werden. Die Verwaltung mehrerer PSKs (für viele Geräte) erfordert Sorgfalt: Jeder PSK sollte einzigartig pro Client sein, und die Verteilung (Provisionierung) der PSKs an die Geräte muss über einen sicheren Kanal erfolgen. Im Vergleich zu Zertifikaten ist PSK-basierte Authentifizierung einfach und ressourcenschonend, skaliert aber weniger gut für sehr große Umgebungen, da der Server jeden PSK individuell managen muss und bei Kompromittierung eines einzelnen PSK kein zentrales sperren (wie bei Zertifikat-CRLs)

möglich ist [41]. Nichtsdestotrotz bildet der PSK in unserer Betrachtung die vertrauliche Grundlage, auf der sich Client und Server gegenseitig Vertrauen schenken.

Challenge-Response-Authentifizierung: Um nun einen Client gegenüber dem Server zu authentisieren, ohne den PSK im Klartext zu übertragen, wird das Challenge-Response-Verfahren eingesetzt. Der Ablauf ist in Abbildung 2.2 dargestellt. Zunächst initiiert der Client eine Anmeldung beim Auth-Service. Der Server generiert daraufhin einen kryptographisch starken Zufallswert als *Challenge N* und schickt diesen an den Client (Schritt 1 in Abb. 2.2). Der Client berechnet mit dem PSK K eine Antwort $R = \text{HMAC}_K(N)$ (Schritt 2) und sendet R zurück an den Server (Schritt 3). Der Server wiederum verifiziert R , indem er selbst $\text{HMAC}_K(N)$ ausrechnet und mit dem empfangenen Wert vergleicht (Schritt 4). Ist die Prüfung erfolgreich, gilt der Client als authentifiziert. Andernfalls wird die Authentifizierung abgelehnt. Dieses Protokoll stellt sicher, dass nur ein Client mit Kenntnis des korrekten PSK die Challenge beantworten kann. Ein Angreifer, der die Kommunikation abhört, sieht nur die zufällige Challenge und den Response-Wert – beide nützen ihm ohne K nichts. Selbst ein *Replay*-Angriff (erneutes Senden einer alten gültigen Response) würde fehlschlagen, da der Server für jede Session eine neue Challenge wählt und alte Antworten somit ungültig sind. Wichtig ist, dass der PSK niemals das Gerät verlässt; es wird stets nur ein HMAC damit gebildet. Optional könnte man dieses Schema zur *mutual authentication* erweitern, indem auch der Server sich gegenüber dem Client ausweist – etwa durch eine zweite Challenge in Gegenrichtung oder durch Ableiten gemeinsamer Session-Parameter. In vielen einfachen Anwendungen genügt jedoch die einseitige Authentifizierung des Clients.

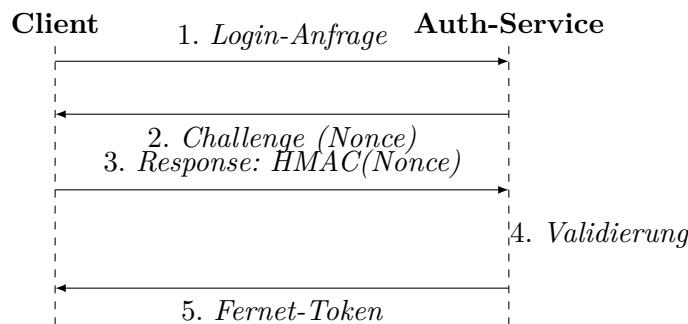


Abbildung 2.2: Challenge-Response-Authentifizierung mit Pre-Shared Key und Fernet-Token.

Tokenisierung und Verwendung des Fernet-Tokens: Nach erfolgreich bestandener Challenge-Response möchte der Server dem Client eine Art „Ticket“ ausstellen, das für nachfolgende Anfragen genutzt werden kann, damit der Client nicht bei jeder Kommunikation erneut eine Challenge-Response durchführen muss. Hier kommen Tokens ins Spiel. Ein *Token* ist im Wesentlichen ein Datenelement, das die erfolgte Authentifizierung repräsentiert.

tiert und vom Client bei weiteren Requests mitgeschickt werden kann („bearer token“ Prinzip). Im Auth-Service wird ein solcher Token nach der Authentifizierung generiert und dem Client übermittelt (Schritt 5 in Abb. 2.2).

Wichtig ist, dass dieser Token sicher gestaltet ist – andernfalls könnte ein Angreifer ihn abfangen und unberechtigt benutzen (*Token Hijacking*). Durch die Verwendung von Fernet stellt der Auth-Service sicher, dass der Token verschlüsselt und signiert ist. Konkret kann der Server z.B. eine Token-Payload erstellen, die folgende Informationen enthält: die Client-ID oder Kennung, ggf. Autorisierungsinformationen (Rollen/Rechte) und einen Zeitstempel bzw. Gültigkeitsdauer. Diese Payload wird dann mit dem Fernet-Schlüssel des Servers verschlüsselt, wodurch der Token entsteht. Da nur der Server den Fernet-Schlüssel besitzt, kann auch nur er den Token später wieder entschlüsseln und validieren. Selbst der Client kennt den Inhalt des Tokens nicht (es sei denn, man gibt absichtlich gewisse Felder in Klartext, was hier nicht der Fall ist).

Beim Erhalt eines Tokens speichert der Client diesen meist lokal (z.B. im Speicher oder in einem sicheren Element) und fügt ihn bei künftigen API-Aufrufen an den Auth-Service oder andere geschützte Dienste in der Kommunikationsarchitektur hinzu (oft als Teil eines Headers oder Nachricht, analog zu einem Session-Cookie). Der Server seinerseits kann eingehende Tokens überprüfen: Er entschlüsselt mit dem Fernet-Key die Payload und prüft die Gültigkeit. Dabei wird automatisch die Integrität via HMAC im Fernet gewährleistet – ist der Token manipuliert oder von einem Dritten erzeugt worden, schlägt die HMAC-Prüfung fehl und der Token wird verworfen. Zusätzlich kann der Server den Timestamp im Token ansehen und beurteilen, ob der Token noch innerhalb der erlaubten Lebensdauer liegt. Ist der Token abgelaufen, fordert der Server vom Client eine neue Authentifizierung (Challenge-Response) oder verweigert die Anfrage. Das Zusammenspiel dieser Komponenten schafft einen robusten Authentifizierungsmechanismus:

1. Der PSK liefert ein geteiltes Geheimnis als Vertrauensbasis.
2. Die Challenge-Response-Interaktion mit HMAC beweist die Kenntnis des PSK ohne ihn preiszugeben, schützt gegen Replay und ermöglicht einmalige Authentifizierungsvorgänge.
3. Der Fernet-verschlüsselte Token fungiert als kurzfristiger Authentifizierungsnachweis, der vom Client für wiederholte Zugriffe verwendet werden kann, ohne jedes Mal den PSK bemühen zu müssen.

Ein großer Vorteil dieser Tokenisierung ist die Entkopplung der weiteren Kommunikation vom ursprünglichen geheimen Schlüssel. Nach der Ausstellung des Tokens muss der Client den PSK für die Gültigkeitsdauer des Tokens nicht erneut verwenden oder preisgeben. Selbst wenn ein Token kompromittiert würde (etwa durch Diebstahl), würde das

zugrunde liegende PSK dadurch nicht unmittelbar bekannt. Allerdings könnte ein gestohler gültiger Token von einem Angreifer verwendet werden (*Replay Attack* mit Token). Daher ist es essenziell, die Tokens ausreichend kurzlebig zu machen und die Übertragung derselben z.B. durch Transportverschlüsselung (TLS) zu schützen. In summe ermöglicht die Kombination aus Challenge-Response und Token eine effiziente Authentifizierung: Der rechenintensivere Teil (HMAC-Berechnung) und ein Roundtrip erfolgen nur bei der Anmeldung, während für jede weitere Anfrage der Token als Authentifizierungsbeweis dient. Der Auth-Service kann damit gegenüber nachgelagerten Diensten oder bei späteren Verbindungen schnell die Identität des Clients prüfen, indem er den Token entschlüsselt, ohne erneut eine Nutzer-Eingabe oder ein komplexes Handshake-Protokoll zu durchlaufen. Dieses Muster findet sich in vielen Systemen wieder – man denke etwa an Web-Logins, bei denen nach einmaliger Passworteingabe ein Session-Cookie ausgestellt wird.

Abschließend sei erwähnt, dass Pre-Shared-Keys und symmetrische Tokens vor allem in Szenarien zum Einsatz kommen, wo eine Ende-zu-Ende-Vertrauensstellung zwischen genau zwei Parteien besteht (hier: Gerät und Auth-Service) und die Verwaltungsaufwände für PSKs tragbar sind. In großen verteilten Systemen könnte man alternativ auch Public-Key-Verfahren einsetzen (z.B. Client-Zertifikate und JWTs mit digitaler Signatur), die andere Vor- und Nachteile bieten. Für die Zwecke des betrachteten Auth-Service – vermutlich die Absicherung einer beschränkten Anzahl bekannter Devices – liefert die beschriebene Kombination jedoch einen gut geeigneten Kompromiss aus Sicherheit und Einfachheit. Alle verwendeten Bausteine (HMAC, Fernet/AES, Challenge-Response) gelten als kryptographisch solide. Somit bilden sie eine zuverlässige Grundlage, um im Auth-Service Vertraulichkeit, Integrität und Authentifizierung sicherzustellen und gegen gängige Angriffe (Abhören, Manipulation, Replay) gewappnet zu sein.

2.10 Mikrocontroller und eingebettete Systeme

Aufbau und Hauptkomponenten

Ein *Mikrocontroller* ist ein kleiner Computer auf einem einzigen integrierten Schaltkreis, der eine zentrale Recheneinheit (CPU), Speicher und Ein-/Ausgabe-Peripherie enthält. Er ist für spezifische Steuerungsaufgaben konzipiert und bildet das Herz vieler eingebetteter Systeme. *Eingebettete Systeme* sind spezialisierte Computersysteme, die als Teil eines größeren Geräts eine dedizierte Funktion erfüllen. Typischerweise integriert ein Mikrocontroller alle notwendigen Komponenten – CPU, Programmspeicher (meist Flash), Datenspeicher (SRAM), Timer, digitale/analoge I/O, serielle Schnittstellen usw. – auf einem einzigen Chip, sodass kein externes Unterstützungschip erforderlich ist. Dadurch

können Mikrocontroller autark Geräte wie Sensoren und Aktoren steuern. [50]

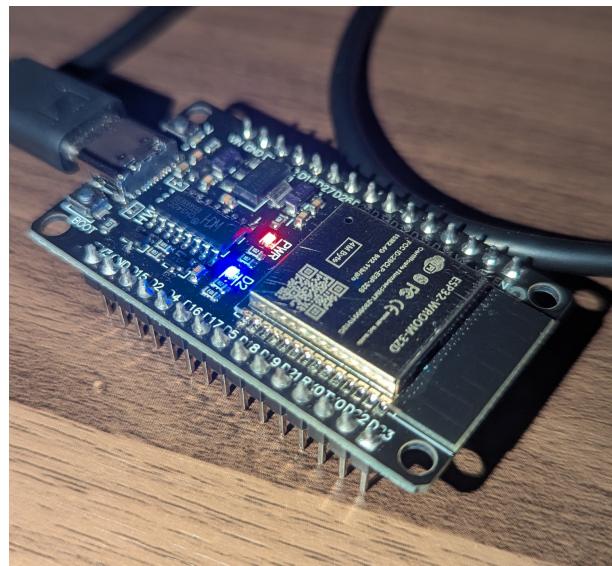


Abbildung 2.3: Bild eines ESP32 Mikrocontrollers

Energiemanagement

In Heimautomatisierung und IoT-Anwendungen ist *Energiemanagement* zentral, da viele Geräte batteriebetrieben oder dauerhaft aktiv sind. Mikrocontroller bieten verschiedene Low-Power-Modi (z. B. Schlafmodus, Tiefschlafmodus), in denen sie unbenötigte Komponenten abschalten oder deren Takt reduzieren, um Energie zu sparen. [51] Beispielsweise können Peripherien deaktiviert und durch *Clock Gating* der Takt für inaktive Module angehalten werden, um den Verbrauch zu senken. [51] Moderne Techniken wie *Dynamic Voltage and Frequency Scaling* (DVFS) passen zudem Versorgungsspannung und Taktfrequenz dynamisch an die aktuelle Rechenlast an. [51] Auf diese Weise lässt sich die Leistungsaufnahme minimieren, ohne die essenziellen Funktionen zu beeinträchtigen. Ein effizientes Energiemanagement verlängert die Batterielaufzeit und reduziert die Wärmeentwicklung – beides essenziell für sensorbasierte Haussteuerungen mit dauerhafter Betriebsbereitschaft.

Echtzeitverhalten

Eingebettete Systeme zur Steuerung von Geräten im Smart Home unterliegen oft *Echtzeit-Anforderungen*. *Echtzeitsysteme* sind Systeme, die innerhalb vorgegebener Zeitgrenzen korrekt reagieren müssen; das zeitliche Verhalten ist Bestandteil der funktionalen Korrektheit. [52] Insbesondere ist die Korrektheit eines Echtzeitsystems nicht nur vom logischen Ergebnis, sondern auch vom physikalischen Zeitpunkt der Ergebniserzeugung bestimmt. [52] Man unterscheidet:

- **Harte Echtzeit:** Eine Überschreitung der Deadline führt zu inakzeptablen Folgen (z. B. Gefährdung von Personen, Gerätebeschädigung). [52]
- **Weiche Echtzeit:** Timing-Verletzungen sind tolerierbar, führen aber zu Leistungseinbußen oder Qualitätsminderungen. [52]

In der Heimautomatisierung dominieren weiche Echtzeitanforderungen (z. B. Heizungsregelung), doch sicherheitsrelevante Funktionen wie Alarmanlagen oder Türschlosssteuerungen erfordern teilweise hartes Echtzeitverhalten. Entsprechend werden Mikrocontroller und ihre Software so ausgelegt, dass deterministische Reaktionszeiten eingehalten werden – etwa durch Einsatz von Echtzeitbetriebssystemen oder direktes Ausführen zeitkritischer Routinen ohne blockierende Aufrufe.

Interrupts

Zur Realisierung eines responsiven Echtzeitverhaltens verwenden Mikrocontroller sogenannte *Interrupts* (Unterbrechungen). Ein Interrupt ist ein asynchrones Signal, mit dem

ein Peripheriegerät oder Sensor den Prozessor veranlasst, das reguläre Programm zu unterbrechen und eine vordefinierte *Interrupt Service Routine (ISR)* auszuführen. Eine häufig zitierte Definition lautet: „Eine asynchrone Unterbrechung (IRQ) ist ein vom Prozessor-externen Umfeld generiertes Signal, das einen Zustand anzeigt und eine Behandlung durch den Prozessor anfordert. Dieses Signal ist nicht mit dem Programmlauf synchronisiert.“ [52]

Durch Interrupts können externe Ereignisse (z. B. erkannte Bewegung) sofort bearbeitet werden, ohne auf die nächste zyklische Abfrage warten zu müssen. Dies ermöglicht *ereignisgesteuerte Systeme*, die auf externe Signale in quasi Echtzeit reagieren. Alternativ existieren zeitgesteuerte (polling-basierte) Ansätze, bei denen der Mikrocontroller Sensoren in festen Intervallen abfragt. Interruptgesteuerte Designs haben den Vorteil minimaler Latenz, erfordern jedoch sorgfältiges Management (z. B. Priorisierung, Interrupt-Latenz), um Abläufe deterministisch und zuverlässig zu halten. In der Praxis werden häufig *kombinierte Ansätze* genutzt: zeitgesteuerte Überwachung für reguläre Aufgaben und Interrupts für dringende Ereignisse.

I/O und Schnittstellen

Mikrocontroller kommunizieren über diverse *Ein-/Ausgabe-Schnittstellen* mit Sensoren, Aktoren und anderen Systemkomponenten. Sie verfügen über:

- **Digitale I/O-Pins** zur Ansteuerung von Aktoren (z. B. Relais für Lampen) oder zum Erfassen binärer Signale (z. B. Fensterkontakte).
- **Analoge Eingänge** mit integrierten A/D-Wandlern zur Messung kontinuierlicher Größen (Temperatur, Helligkeit etc.).
- **PWM-Ausgänge** oder D/A-Wandler zur Ansteuerung analoger Aktoren (z. B. Dimmsteuerung, Motordrehzahl).

Gängige serielle Kommunikationsschnittstellen umfassen:

- **UART (Universal Asynchronous Receiver Transmitter)** – für einfache Punkt-zu-Punkt-Kommunikation (z. B. mit Debugger, Bluetooth-Modul),
- **SPI (Serial Peripheral Interface)** – synchroner Hochgeschwindigkeitsbus für kurze Distanzen,
- **I²C (Inter-Integrated Circuit)** – Mehrpunktbus mit Adressierung, ideal für viele Sensorkomponenten,

- **CAN, USB, Ethernet** – für leistungsstärkere Anwendungen, z. B. in Gateways oder Fahrzeugtechnik.

[50] Viele dieser Schnittstellen sind bereits im Mikrocontroller integriert. [50] Auch sogenannte *GPIOs* (General Purpose Input/Output) lassen sich flexibel als Ein- oder Ausgang programmieren. Die Vielfalt dieser Schnittstellen erlaubt es einem einzelnen Mikrocontroller, alle für eine Heimautomatisierungsaufgabe relevanten Sensoren und Aktoren direkt anzusteuern, ohne externe Steuerlogik. Die **korrekte Auswahl und Konfiguration** (z. B. Pull-up-Widerstände bei I²C, Bus-Timing, Interruptsteuerung bei UART) ist entscheidend für die Funktionssicherheit des Systems.

Abgrenzung zu System-on-Chips (SoCs)

Der Begriff *System-on-Chip* (SoC) überschneidet sich funktional mit dem des Mikrocontrollers, bezeichnet jedoch im weiteren Sinne einen integrierten Schaltkreis, der ein vollständiges elektronisches System auf einem Chip vereint. Mikrocontroller können als spezialisierte SoCs betrachtet werden, die primär einen Mikroprozessor mit Speicher und Basisperipherie für Steuerungsaufgaben enthalten.

Im Unterschied dazu integrieren klassische SoCs zusätzliche leistungsfähige Komponenten wie GPU, DSP, drahtlose Kommunikationsmodule (WLAN, LTE), externe Speicheranbindung sowie komplexe Betriebssysteme (z. B. Linux, Android). Sie finden sich typischerweise in leistungsfähigen Smart-Home-Zentralen oder Gateways, die Daten aggregieren und komplexe Anwendungen wie Spracherkennung oder Cloud-Anbindung ausführen.

Allgemein gilt: „Ein System-on-Chip ist ein vollständiges elektronisches System auf einem einzigen Chip, der eine Vielzahl von Funktionen ausführen kann.“ Mikrocontroller hingegen sind kompakter, energieeffizienter, günstiger und auf spezifische Aufgaben optimiert. Typische Flash-Größen liegen im Bereich von einigen Kilobyte bis wenigen Megabyte. [53]

Fazit: In der Heimautomatisierung finden sich beide Technologien: **Mikrocontroller** in Sensor- und Aktormodulen (z. B. ein Feuchtigkeitssensor mit Funkmodul) und **SoCs** in Smart-Home-Hubs und Gateways, die zentrale Logik und Kommunikation übernehmen.

2.11 Grundlagen des Internet of Things (IoT) und Smart Home

2.11.1 Vernetzte Systeme und Sensorintegration

Das *Internet of Things* (IoT) bezeichnet ein Netzwerk von physischen Objekten („Dingen“), die mit Sensoren, Software und Konnektivität ausgestattet sind, um Daten zu erfassen und auszutauschen. Im Smart-Home-Kontext bedeutet dies, dass Haushaltsgeräte, Sensoren und Aktoren untereinander sowie mit dem Internet vernetzt sind.

Vernetzte Systeme in der Heiamtomaticierung integrieren häufig zahlreiche Sensoren (z. B. Temperatur, Bewegung, Luftqualität, Licht, Feuchtigkeit) an verschiedenen Orten des Hauses. Diese übermitteln ihre Daten an zentrale Steuerungen oder Cloud-Dienste. Die Sensorintegration kann sowohl **lokal** – etwa über ein Heimnetzwerk (WLAN, ZigBee, Z-Wave etc.) an eine Home-Automation-Zentrale – als auch **cloud-basiert** erfolgen, wobei Sensoren direkt über Internetprotokolle kommunizieren.

Entscheidend ist, dass heterogene Geräte **interoperabel** kommunizieren können. Smart-Home-Plattformen (z. B. Apple HomeKit, Google Home) definieren dazu gemeinsame Protokolle, die eine systemübergreifende Verarbeitung und Steuerung ermöglichen. Neben der Kommunikation ist auch eine lokale **Datenvorverarbeitung** erforderlich – etwa zum Filtern von Sensorrauschen oder zur Aggregation von Messwerten – um Datenvolumen und Energieverbrauch zu reduzieren.

Die Vernetzung schafft die Grundlage für **kontextsensitive Entscheidungen** im Smart Home. So kann etwa ein Temperatur- und Feuchtigkeitssensor gemeinsam mit Wetterdaten zur effizienten Heizungsregelung genutzt werden. Insgesamt entsteht ein verteiltes System aus eingebetteten Einheiten, das auf Umweltbedingungen flexibel reagiert. [54]

2.11.2 Kommunikationsprotokolle (MQTT, CoAP, HTTP)

Für die Kommunikation zwischen IoT-Geräten und Diensten haben sich verschiedene spezialisierte Protokolle etabliert:

HTTP (HyperText Transfer Protocol) Das klassische Webprotokoll kommt auch im IoT zum Einsatz – insbesondere bei Geräten, die RESTful-Webservices anbieten. HTTP basiert auf TCP und ist textbasiert, was zu vergleichsweise hohem Overhead führt. Es eignet sich vor allem für leistungsfähige WLAN-Geräte, die direkt mit Cloud-APIs kom-

munizieren. Vorteile sind die weitverbreitete Infrastruktur und Kompatibilität. Nachteilig sind höhere Latenzen und Energiebedarf. [55]

MQTT (Message Queuing Telemetry Transport) MQTT ist ein leichtgewichtiges **Publish-Subscribe**-Protokoll, das speziell für ressourcenbeschränkte Systeme und instabile Netzwerke entwickelt wurde. Es verwendet TCP als Transportprotokoll und benötigt einen zentralen Broker, der Nachrichten zwischen *Publishern* (z. B. Sensoren) und *Subscribers* (z. B. Servern oder Aktoren) vermittelt. MQTT zeichnet sich durch minimalen Overhead und Unterstützung für verschiedene *Quality-of-Service*-Stufen aus. Es eignet sich hervorragend für bidirektionale Kommunikation in Echtzeit, z. B. zur Steuerung von Lampen oder zur Übermittlung von Messwerten an einen lokalen Broker (z. B. Raspberry Pi). [56]

CoAP (Constrained Application Protocol) CoAP ist ein binäres, auf UDP basierendes Protokoll für RESTful-Kommunikation unter extrem ressourcenarmen Bedingungen (z. B. 6LoWPAN). Es nutzt komprimierte Header, unterstützt optional zuverlässige Übertragungen über Confirmable Messages und erlaubt **Multicast**. Im Gegensatz zu MQTT ist CoAP stärker auf direkte Client-Server-Interaktionen ausgerichtet und wird häufig in sensornahen IPv6-Netzwerken oder zwischen IoT-Geräten und lokalen Gateways eingesetzt. [56]

Zusammenfassung

- HTTP: weit verbreitet, aber overheadintensiv – geeignet für leistungsfähige Geräte.
- MQTT: effizient für zentrale Nachrichtenvermittlung bei geringen Ressourcen.
- CoAP: ideal für direkte, lokale Kommunikation mit geringem Energiebedarf.

Die Protokollwahl beeinflusst maßgeblich Energieeffizienz, Reaktionszeit und Zuverlässigkeit. In modernen Smart-Home-Systemen ist häufig eine Kombination mehrerer Protokolle im Einsatz.

2.11.3 Architekturmodelle (Edge, Cloud, Fog)

Zur effizienten Verteilung von Rechenlast, Speicherbedarf und Steuerlogik im IoT haben sich drei Architekturebenen etabliert:

Edge Computing bezeichnet die Datenverarbeitung direkt am Endgerät, etwa im Mikrocontroller eines Sensors oder Aktors. Vorteilhaft sind minimale Latenz, Ausfallsicherheit bei Verbindungsproblemen und hohe Energieeffizienz. Beispiele sind das lokale Filtern von Sensorwerten oder einfache Steuerentscheidungen (z. B. Schwellenwertüberschreitung).

Fog Computing stellt eine mittlere Verarbeitungsebene dar, häufig in Form eines Gateways oder Home-Servers im lokalen Netzwerk. Hier können mehrere Datenströme aggregiert, komplexere Algorithmen ausgeführt oder lokale Regelungen umgesetzt werden. Ein Beispiel ist die Raumtemperaturregelung anhand mehrerer Sensordaten, ohne Cloud-Beteiligung.

Cloud Computing findet in entfernten Rechenzentren statt und bietet skalierbare Ressourcen für Speicherung, Analyse und KI-basierte Optimierung. Typische Aufgaben sind Nutzerverhaltenserkennung, Fernzugriff, Sprachsteuerung, Updates oder Backup-Dienste. Die Cloud erhöht die Funktionalität, ist jedoch auf stabile Internetverbindung angewiesen und bringt höhere Latenzen mit sich.

Hybride Umsetzung In der Praxis arbeiten alle drei Ebenen zusammen: Ein *Edge-Gerät* erkennt z. B. eine Türöffnung, der *Fog-Knoten* analysiert das Ereignis und entscheidet über eine Alarmmeldung, während die *Cloud* die Benachrichtigung an das Smartphone des Nutzers übernimmt. Durch diese verteilte Architektur entsteht ein robustes, skalierbares und effizient arbeitendes System. Moderne IoT-Plattformen wie *MEC* (Multi-access Edge Computing) gehen noch einen Schritt weiter, indem sie Cloud-Funktionalitäten näher an das Edge-Gerät verlagern – etwa in den Router oder das Gateway.

Fazit Ein klug verteiltes Architekturmodell erhöht Ausfallsicherheit, reduziert Latenzen und erlaubt eine flexible Skalierung je nach Anwendung und Sicherheitsanforderung. [57]

Sicherheitsaspekte (Authentifizierung, Datenschutz, Angriffsszenarien)

Mit zunehmender Vernetzung im Smart Home wachsen die Herausforderungen der IT-Sicherheit und des Datenschutzes. IoT-Geräte sind häufig Ziel von Angriffen, weil sie oft weniger geschützt sind als PCs oder Smartphones. Wichtige Sicherheitsaspekte sind:

Authentifizierung Sicherstellen, dass nur berechtigte Benutzer und Geräte Zugriff auf das Smart-Home-System haben. Viele IoT-Geräte werden ab Werk mit Standard-Kennwörtern ausgeliefert, die vom Nutzer selten geändert werden. Dieses bekannte Problem ermöglicht einfache Angriffe: „Viele IoT-Geräte werden mit Standard-Benutzernamen und -Kennwörtern ausgeliefert ... Angreifer kennen diese Standard-Anmeldedaten sehr gut.“ Starke Passwörter oder besser zertifikatsbasierte Authentifizierung sind daher essenziell. Auch Geräte untereinander sollten sich authentisieren (z. B. ein Sensor gegenüber dem Gateway), um Spoofing-Angriffe zu verhindern. In modernen Smart-Home-Plattformen kommen oft Public-Key-Verfahren zum Einsatz, bei denen jedes Gerät ein eigenes Schlüsselpaar hat. Fehlende oder schwache Authentifizierung kann dazu führen, dass Angreifer unbefugt Befehle senden (etwa Türöffnung). [58]

Datenschutz und Verschlüsselung Sensoren im Haus sammeln teils sehr persönliche Daten (Bewegungsprofile, Kameraaufnahmen, Gesundheitsdaten von Wearables etc.). Datenschutz erfordert, dass diese Daten nur zweckgebunden verwendet und angemessen geschützt werden. In der Übertragung ist Verschlüsselung obligatorisch: IoT-Geräte, die unverschlüsselt kommunizieren, ermöglichen Man-in-the-Middle-Angriffe, bei denen ein Angreifer die Daten abfängt und Einblick in das Privatleben erhält. Tatsächlich sind viele IoT-Kommunikationen anfällig, da standardmäßig oft keine Transportverschlüsselung aktiviert ist. Durchgehende Ende-zu-Ende-Verschlüsselung (etwa TLS/DTLS für IP-basierte Protokolle) sollte umgesetzt werden. Zudem müssen in Cloud-Plattformen die Daten gemäß Datenschutzrichtlinien (wie DSGVO) verarbeitet werden – etwa Minimierung der Speicherung personenbeziehbarer Daten oder Anonymisierung von Sensordaten, wo immer möglich. [58]

Angriffsszenarien IoT-Geräte können auf verschiedene Weise angegriffen werden. Neben Passwort-Angriffen (Brute-Force bekannter Standard-Passwörter) und Abhören unverschlüsselter Verbindungen sind Software-Schwachstellen ein Einfallstor. Viele Smart-

Home-Geräte haben unsichere Web-Interfaces oder veraltete Firmware mit bekannten Lücken. Wenn Hersteller keine regelmäßigen Updates/Patches bereitstellen, bleiben diese Lücken offen. Angreifer können so die Kontrolle über Geräte übernehmen und z. B. Kameras ausspähen oder das Gerät Teil eines Botnetzes werden lassen.

Ein berühmtes Beispiel ist das *Mirai-Botnet*: eine Malware durchsuchte automatisiert das Internet nach IoT-Geräten mit Standard-Login und infizierte hunderttausende Geräte (vor allem Kameras, Recorder), um daraus ein Botnetz für Distributed-Denial-of-Service (DDoS)-Attacken zu formen. [59] „Mirai zielt auf IoT-Geräte ab, bei denen das Standardpasswort noch aktiv ist ... und macht sie zu einem Teil eines Botnets, das dann für einen DDoS-Angriff verwendet wird.“ Dieses Angriffsszenario zeigte eindrücklich, dass auch vermeintlich harmlose Geräte (wie eine vernetzte Kamera) in großer Zahl massive Schäden im Internet anrichten können. [59]

Physische Sicherheit In Smart Homes stehen IoT-Geräte auch physisch zugänglich in der Wohnung oder sogar außerhalb (Außensensoren, Türklingeln). Ein Angreifer mit physischem Zugriff kann Geräte manipulieren, zurücksetzen oder austauschen. Daher sollten sicherheitskritische Komponenten manipulationsgeschützt installiert werden (z. B. Alarm-Zentrale versteckt und sabotagesicher). Zudem sollten Sicherheitschips (Secure Elements) verwendet werden, die kritische Schlüssel speichern, um Hardware-Angriffe zu erschweren.

Angriffsabwehr und Best Practices Um das Risiko zu senken, müssen in IoT-Systemen Mechanismen wie Zugriffskontrolle (z. B. rollenbasierte Rechte für Nutzer), Sicherheitsprotokolle (z. B. regelmäßig wechselnde Tokens, Timeouts für Sessions) und Monitoring (Erkennung ungewöhnlicher Geräteaktivitäten) implementiert werden. Auch im Heimnetz empfiehlt sich Netzwerksegmentierung: IoT-Geräte sollten in ein separates VLAN, um im Falle einer Kompromittierung das Ausbreiten zu verhindern. Nutzeraufklärung spielt ebenfalls eine Rolle – z. B. sollte der Bewohner wissen, wie er Geräte sicher einbindet, Updates durchführt und starke Passwörter setzt.

Sicherheit und Datenschutz sind in der Heiamtomaticierung kritische Grundlagen, da die Verletzung dieser Aspekte nicht nur abstrakte IT-Schäden bedeutet, sondern direkt die Privatsphäre und Sicherheit der Bewohner betrifft. Anerkannte Standards und Frameworks (wie *OWASP IoT Top 10* für Gerätesicherheit oder *IEC 62443* für industrielle IoT-Sicherheit) bieten Leitlinien, die zunehmend auch im Consumer-IoT Beachtung finden. Durch *Security by Design* – also Berücksichtigung von Sicherheitsmaßnahmen von der ersten Entwicklungsphase an – sollen zukünftige Smart-Home-Geräte robuste Ab-

wehrmechanismen aufweisen, um das Vertrauen der Nutzer in die Technik zu stärken.

2.12 Regelungstechnik in Software

Steuerungs- und Regelungskonzepte (Open Loop vs. Closed Loop)

In eingebetteten Systemen der Heiamtomaticierung kommen sowohl Steuerungen (Open-Loop-Control) als auch Regelungen (Closed-Loop-Control) zum Einsatz, je nach Anforderung. Bei einer Steuerung beeinflusst die Software die Stellgrößen (Aktoren) auf Basis von Eingangsgrößen nach festgelegter Logik, ohne dass eine Rückmeldung der Ausgangsgröße erfolgt. [60] Das System kennt also den Effekt seiner Eingriffe nicht direkt – ein klassisches Beispiel ist eine zeitgesteuerte Gartenbewässerung: Der Regner schaltet für 10 Minuten ein (Steuerbefehl), unabhängig davon, ob der Boden schon feucht ist oder nicht.

Eine Regelung dagegen beinhaltet eine Rückkopplung: Sensoren messen kontinuierlich die Ist-Größe und vergleichen sie mit dem Sollwert; die Software passt die Stellgröße laufend an, um Abweichungen auszugleichen. [61] Ein Thermostat ist hierfür beispielhaft: Die Heizung wird auf Basis der gemessenen Temperatur ein- oder ausgeschaltet, um den Sollwert konstant zu halten. Formal gesprochen: „Bei einer Regelung wird die Ausgangsgröße zurückgeführt und mit der Führungsgröße (Sollwert) verglichen“, was die Grundlage bildet, um Abweichungen durch Steuerbefehle zu minimieren. [60]

Der Unterschied zwischen offener und geschlossener Wirkungsweise ist wesentlich. Steuerungen (open loop) sind einfach und benötigen keine Sensor-Rückmeldung – sie können jedoch Ungenauigkeiten nicht korrigieren. Regelungen (closed loop) sind aufwendiger (erfordern Sensorik und Regleralgorithmen), bieten aber Genauigkeit und Robustheit gegenüber Störungen.

In der Praxis verwendet man in Smart Homes häufig Regelungen überall dort, wo Präzision wichtig ist (z.,B. Temperaturregelung, Motordrehzahlregelung in Rollläden), während reine Steuerungen für simpler gelagerte Aufgaben oder als Sicherheitsebene eingesetzt werden. Oft gibt es auch Kombinationen: etwa eine Zeitsteuerung mit einer nachgeschalteten Regelung. Die Beherrschung dieser Konzepte setzt Kenntnisse in der Regelungstechnik voraus, also mathematische Beschreibung von Regelkreisen, Stabilitätskriterien etc., was in die Software implementiert wird.

Schwellenwertsteuerung

Eine einfache Form der Regelung/Steuerung ist die Schwellenwertsteuerung, oft auch Zwei-Punkt-Regelung genannt. Hierbei wird eine Aktion ausgelöst, sobald eine Messgröße

einen definierten Schwellenwert über- oder unterschreitet. Das System kennt also nur zwei Zustände – z.B. Heizung **EIN** vs. **AUS** abhängig davon, ob die Temperatur unter oder über dem Sollwert liegt.

Ein klassisches Beispiel ist der einfache Thermostat: Fällt die Raumtemperatur unter 20 °C, schaltet der Regler die Heizung an; steigt sie über 20 °C, wird die Heizung abgeschaltet. [61]

Solche Schwellenwertsteuerungen arbeiten diskontinuierlich (die Stellgröße springt zwischen zwei Werten) und sind leicht implementierbar (**if-then**-Logik in Software). Vorteil ist die Einfachheit und oft hohe Zuverlässigkeit; Nachteil können Schwingungen oder ständiges Ein/Aus-Schalten sein, wenn keine Hysterese eingebaut ist. Daher fügt man praktisch meist eine Hysterese hinzu – z.B. Heizung **an** unter 19,5 °C und erst **aus** über 20,5 °C – um ein zu häufiges Schalten zu vermeiden.

In vielen Heimautomatisierungs-Szenarien genügt eine Schwellenwertsteuerung: etwa das Einschalten des Lichts bei Unterschreiten eines Helligkeitswerts, Öffnen der Jalousie bei Überschreiten eines Helligkeitsgrenzwerts oder Lüften bei Überschreiten eines CO₂-Schwellenwerts. Die Implementierung erfolgt in der Software meist durch Abfragen der Sensor-signale und Setzen der Aktorausgänge entsprechend der festgelegten Grenzwerte.

Diese Form der Steuerung zählt zu den unstetigen Reglern (binäre Ausgangszustände) und kann als Sonderfall einer Regelung angesehen werden (der Istwert wird mit dem Grenzwert verglichen = Rückführung vorhanden, aber Stellgröße nimmt nur zwei Zustände an). Insbesondere in sicherheitsgerichteten Funktionen (z.,B. Gasmelder – Gas über Schwelle ⇒ Ventil zu) sind Schwellenwertmechanismen beliebt wegen ihrer Klarheit und Nachvollziehbarkeit.

Einfache Regler (P, PI, PID)

Für feinere Kontrolle kontinuierlicher Größen werden in der Regelungstechnik häufig lineare Regler eingesetzt, vor allem die Familien der **P**-, **PI**-, **PD**- und **PID**-Regler. Diese sind durch die drei Grundaktionen *Proportional (P)*, *Integral (I)* und *Differential (D)* gekennzeichnet, die auch kombiniert auftreten können.

In eingebetteten Steuerungen eines Smart Homes findet man z.,B. in Heizungs- oder Klimaregelungen oft **PI-Regler**, um eine konstante Temperatur ohne bleibenden Fehler einzuregeln, oder **PID-Regler** in Motorsteuerungen (z.,B. für Lüfter oder Pumpen), um schnelle und stabile Stellgrößenregelungen zu erreichen.

P-Regler (Proportionalregler): Die Stellgröße wird proportional zur aktuellen Regelabweichung

$$e(t) = \text{Soll} - \text{Ist}$$

verändert. Der P-Anteil reagiert damit sofort auf Abweichungen – je größer der Fehler, desto stärker das Stellsignal. Dadurch wird ein direktes Korrekturverhalten erzielt: Ist die Raumtemperatur 2 °C unter dem Sollwert, erhöht ein P-Regler z.,B. das Ventil überproportional.

Allerdings führt ein reiner P-Regler meist zu einer bleibenden Regeldifferenz (Offset), da er bei Erreichen des Sollwerts das Stellsignal auf null reduziert, noch bevor der Istwert exakt den Sollwert erreicht.

I-Regler (Integralregler): Der I-Anteil integriert die vergangene Regelabweichung – Fehler werden über die Zeit aufaddiert. Dies ermöglicht es, einen bleibenden Fehler auszugleichen. [62] Der I-Anteil reagiert langsam, ist aber in PI/PID-Reglern essenziell, um den stationären Fehler zu beseitigen.

D-Regler (Differenzialregler): Der D-Anteil berücksichtigt die Änderungsgeschwindigkeit der Regelabweichung, also $de(t)/dt$. Er wirkt dämpfend auf schnelle Änderungen und reduziert Überschwingen. [62] In Software wird er meist durch Differenzen zwischen zwei Abtastzeitpunkten genähert.

PI- und PID-Regler: Kombinationen dieser Typen (PI, PID) vereinen schnelles Reagieren (P), Fehlerausgleich (I) und Dämpfung (D). Ein PID-Regler bietet geringe Einschwingzeit, keine bleibende Regelabweichung und gutes Störverhalten. In der Software wird ein PID-Regler meist diskret realisiert, z.,B. mit:

$$u(t) = K_P \left(e(t) + \frac{1}{T_I} \int e(t) dt + T_D \frac{de(t)}{dt} \right) \quad (2.1)$$

[63]

Datenverarbeitung zur Entscheidungsfindung

Vor jeder Stellentscheidung muss das System die Rohdaten aufbereiten. Dazu zählen:

- **Signalaufbereitung:** Rauschfilterung, Mittelwertbildung, Kalibrierung.
- **Sensorfusion:** Zusammenführen mehrerer Quellen zur robusteren Entscheidungsbasis.

- **Zustandsautomaten:** Logiksteuerung je nach Betriebsmodus.
- **KI-gestützte Algorithmen:** Zunehmender Einsatz von Embedded AI zur Prognose und Mustererkennung.
- **Fehlererkennung:** Erkennen fehlerhafter oder ausgefallener Sensoren und Wechsel in Failsafe-Zustände.

Nach dieser Verarbeitung wird auf Basis der Sensordaten entschieden, ob eine Steuerung oder Regelung aktiviert wird. Die Datenverarbeitung bildet somit die Brücke zwischen physikalischer Messung und softwarebasierter Aktorik. Sie ist damit essentiell für eine sichere, präzise und energieeffiziente Heiamtomaticierung.

3 Anforderungen

3.1 Anforderungsdefinition für das Frontend

Die im Rahmen dieses Projekts entwickelte Anwendung zielt auf eine benutzerfreundliche, responsive und funktionale Weboberfläche zur Verwaltung von Pflanzen, Sensoren und Benutzergruppen in einem smarten Bewässerungssystem ab. Die folgenden Anforderungen leiten sich aus der konzeptionellen Planung sowie grundlegenden Prinzipien des nutzerzentrierten Designs ab.

Zentrales Ziel der Frontend-Architektur ist die Bereitstellung eines klar strukturierten Interfaces, das eine intuitive Navigation und konsistente Interaktion ermöglicht. Die geplante Startansicht bietet einen Überblick über Räume und zugeordnete Pflanzen. Eine hierarchische Strukturierung in Wohnung, Raum und Pflanze unterstützt das mentale Modell der NutzerInnen und erlaubt eine modulare Skalierung der Anwendung. Die einzelnen Pflanzenkarten sollen aggregierte Umweltdaten anzeigen, welche über Sensoren erfasst werden. Eine Detailansicht ermöglicht zukünftig die Auswertung dieser Daten in Form von Diagrammen.

Eine essentielle Anforderung ist die datengetriebene Darstellung der Messwerte auf Basis einer API-Kommunikation. Das Interface muss dabei sowohl aktuelle Sensorwerte anzeigen als auch historische Veränderungen visuell aufbereiten. Die Komponenten sollen als modulare Einheiten konzipiert werden und einen reaktiven Datenfluss mit dem zentralen State-Management der Applikation unterstützen.

Das Design ist für eine vollständig responsive Darstellung ausgelegt, die sich an unterschiedliche Displaygrößen und Endgeräte anpasst. Für die Oberfläche wird eine Bottom-Navigation vorgesehen, die den Zugriff auf zentrale Bereiche wie Dashboard, Gruppenverwaltung und Profil auch bei begrenztem Platzangebot ermöglicht. Die Anwendung unterstützt sowohl einen Dark Mode als auch einen Light Mode, die dynamisch aktiviert werden können. Diese Darstellung soll auf Systemebene automatisch adaptiert werden. Die Applikation ist darüber hinaus vollständig mehrsprachig gestaltet. Alle Oberflächentexte müssen zur Laufzeit lokalisierbar sein, um eine internationale Nutzbarkeit zu ermöglichen.

Ein zentrales Funktionsmodul ist die Verwaltung der Pflanzen. Hierzu zählen das Hin-

zufügen neuer Pflanzen über ein Formular, die Auswahl eines Sensors, die Angabe von Zielwerten sowie eine textuelle Beschreibung der Pflanze. Eine komponentenbasierte Ansicht zur Pflege von Gruppen und Wohnungen soll es mehreren BenutzerInnen ermöglichen, gemeinsam auf bestimmte Räume zuzugreifen. Die Rollen- und Rechtevergabe erfolgt im Backend, die Anzeige jedoch im Frontend.

Zur Authentifizierung und Autorisierung wird ein modulares Login-System mit passwortgeschütztem Zugang benötigt. Die Registrierung erfolgt über einen mehrstufigen Account-Creation-Stepper, der BenutzerInnen schrittweise durch den Registrierungsprozess führt. Dabei werden u. a. Benutzername, E-Mail und Initialkonfigurationen für eine Gruppe abgefragt.

Darüber hinaus soll die Anwendung ein Einstellungsmodul enthalten, das NutzerInnen erlaubt, etwa Sprache oder Accountinformationen zu bearbeiten. Datenschutzoptionen oder das Löschen des App-Caches.

Zusätzlich wurde die Idee einer automatischen Pflanzenerkennung über eine bildbasierte Künstliche Intelligenz (KI)-Komponente konzipiert und eine Android-App erstellt. Diese ist jedoch nicht Bestandteil der Kernanforderungen, sondern stellt eine potenzielle Erweiterung dar, die zukünftig in den Entwicklungsprozess aufgenommen werden kann.

Für das Projekt Sensorsa gelten hohe Anforderungen an Zuverlässigkeit, Sicherheit, Erweiterbarkeit und Datenintegrität. In diesem Kapitel werden die formellen Anforderungen an das Backend sowie an die zugrunde liegende Datenbankarchitektur getrennt voneinander dargestellt. Ziel ist es, eine tragfähige Grundlage für die technische Umsetzung zu schaffen.

3.2 Anforderungen an die entwickelten Schnittstellen-Services im Projekt Sensorsa

Die Hauptaufgabe der Schnittstellen-Services lag in der Konzeption und Realisierung mehrerer Dienste, die als Vermittlungskomponenten zwischen Sensor-Controllern, Benutzerschnittstellen, dem zentralen Datenspeicher sowie einem Nachrichtenübertragungsmechanismus fungieren. Diese Services sind nicht Bestandteil der Steuerlogik auf Hardwareebene, sondern unterstützen den bidirektionalen Informationsaustausch und die sichere Verwaltung verteilter Geräteinstanzen.

Ziel dieses Kapitels ist es, die funktionalen und nicht-funktionalen Anforderungen an diese Schnittstellenkomponenten technologieoffen zu definieren. Die konkrete Wahl der eingesetzten Technologien sowie deren theoretische Fundierung erfolgt erst in den nachfolgenden Kapiteln.

3.2.1 Allgemeine Anforderungen an alle Schnittstellen-Services

Unabhängig von ihrer konkreten Aufgabe müssen sämtliche entwickelten Services folgende generelle Anforderungen erfüllen, die sich aus dem Aufbau des Gesamtsystems sowie den Entwicklungsprinzipien eines modernen verteilten Softwaresystems ergeben:

- **Modularität:** Die Komponenten sollen entkoppelt und unabhängig voneinander betreibbar sein, sodass sie einzeln aktualisiert, getestet und ersetzt werden können.
- **Plattformunabhängigkeit:** Die Services müssen innerhalb einer containerisierten Umgebung lauffähig sein und dürfen keine plattformspezifischen Abhängigkeiten voraussetzen.
- **Fehlertoleranz:** Die Komponenten müssen so gestaltet sein, dass bei Ausfall abhängiger Systeme (z. B. Netzwerk, Datenbank) keine kritischen Fehler entstehen. Entsprechende Retry-Mechanismen und Fehlerprotokollierung sind vorzusehen.
- **Datensicherheit:** Sensible Informationen dürfen zu keinem Zeitpunkt im Klartext übertragen oder ungeschützt gespeichert werden. Eine sichere Authentifizierung und Zugriffsbeschränkung ist auf allen öffentlich erreichbaren Schnittstellen erforderlich.
- **Skalierbarkeit und Erweiterbarkeit:** Die Architektur der Services soll so beschaffen sein, dass zusätzliche Sensoren, Controller oder Benutzer ohne grundlegende Systemänderungen hinzugefügt werden können.
- **Zuverlässigkeit bei der Kommunikation:** Die Kommunikation zwischen Services sowie mit externen Geräten soll gegen Nachrichtenverlust abgesichert sein, insbesondere bei systemkritischen Operationen wie Steuerbefehlen oder Datenspeicherung.

3.2.2 Registrierungs- und Authentifizierungsservice (Auth-Service)

Der Auth-Service bildet die sicherheitsrelevante Schnittstelle zur Einbindung verteilter Steuergeräte (Controller) in das System. Er ist verantwortlich für die Validierung, Registrierung und individuelle Konfiguration dieser Geräte.

Funktionale Anforderungen

- Der Service muss es ermöglichen, neue Geräteinstanzen kontrolliert durch autorisierte Verwaltungsprozesse zu registrieren. Eine unbeaufsichtigte Selbstregistrierung ist auszuschließen.

- Für jede registrierte Geräteinstanz ist ein eindeutiger Identifikator sowie ein zugehöriges Zugriffsprofil zu erzeugen. Dieses Profil muss zur differenzierten Rechtevergabe geeignet sein.
- Die Authentifizierung eines Geräts gegenüber dem System muss über ein sicheres Verfahren erfolgen, das keine langfristige Speicherung von Klartextgeheimnissen erfordert und gleichzeitig eine manipulationssichere Verifikation ermöglicht.
- Nach erfolgreicher Authentifizierung soll eine Kommunikationsfähigkeit zwischen Gerät und System ermöglicht werden, die sich explizit auf festgelegte Kommunikationskanäle beschränkt.
- Der Service muss relevante Metadaten persistieren, sodass die zugehörige Geräteklogik (z. B. Sensor- oder Aktorzuordnung) systemweit nachvollzogen werden kann.

Nicht-funktionale Anforderungen

- Die Registrierungslogik darf nur durch explizit autorisierte Systeme oder Nutzer aufrufbar sein.
- Die Kommunikation mit dem Service muss verschlüsselt erfolgen.
- Eine versehentliche Mehrfachregistrierung desselben Geräts ist zu erkennen und abzuweisen.

3.2.3 E-Mail-Verifikationsservice (Mail-Service)

Der Mail-Service übernimmt die Kommunikation mit Nutzenden zur Verifizierung neu angelegter Benutzerkonten. Seine primäre Aufgabe ist es, die Zustellung von zeitlich begrenzten Bestätigungslinks zu ermöglichen.

Funktionale Anforderungen

- Der Service muss über eine Schnittstelle ansprechbar sein, über die Verifizierungsanfragen gestellt werden können.
- Nach Validierung der Anfrage ist ein einmalig nutzbarer Link zu generieren und über einen gängigen E-Mail-Dienst an den Empfänger zu übermitteln.
- Bei Aufruf des Verifikationslinks muss der Status des entsprechenden Benutzerkontos im System aktualisiert werden.

Nicht-funktionale Anforderungen

- Nur autorisierte Systeme dürfen Anfragen zur Mailverifikation stellen.
- Die Verifizierung darf nur erfolgen, wenn die Kombination aus Benutzername und E-Mail-Adresse im System bekannt ist.
- Ein nicht eingelöster Verifizierungslink muss nach Ablauf einer definierten Zeitspanne seine Gültigkeit verlieren.

3.2.4 Datenschreibdienst für Messdaten (Database Writer)

Dieser Dienst verarbeitet eingehende Datenströme von Sensoren und persistiert die extrahierten Informationen strukturiert in einem relationalen Datensystem.

Funktionale Anforderungen

- Der Dienst muss kontinuierlich eingehende Daten von Messgeräten empfangen, analysieren und in geeigneter Form speichern.
- Die Identität der messenden Einheit muss eindeutig ermittelbar sein. Ist die Einheit im System nicht bekannt, so muss diese bei Bedarf dynamisch registriert werden können.
- Die Speicherung darf nur erfolgen, wenn eine vollständige logische Zuordnung der Sensordaten zu einer Pflanze bzw. zum zugehörigen Anwendungsfall gegeben ist.
- Es müssen regelmäßig alle bekannten Sensoren auf Aktivität geprüft werden. Wird über einen definierten Zeitraum keine neue Messung empfangen, ist dies systemintern als Fehlverhalten zu markieren.

Nicht-funktionale Anforderungen

- Der Dienst muss bei Netzwerkausfällen oder vorübergehenden Störungen in der Verbindung zur Datenbank stabil bleiben.
- Wiederholte Nachrichtenübertragungen dürfen nicht zu doppelten Datensätzen führen.
- Die Datenverarbeitung muss nachvollziehbar protokolliert werden, um Fehler oder Auffälligkeiten zu diagnostizieren.

3.2.5 Service zur Steuerung von Zielwerten (Setpoint API)

Über diesen Dienst können Zielwerte (Sollwerte) für bestimmte Sensoren oder Aktoren von außen gesetzt werden. Ziel ist es, die Regelgrößen im System dynamisch anpassen zu können.

Funktionale Anforderungen

- Der Dienst muss eine externe Schnittstelle bereitstellen, über die Zielwerte für bestimmte Komponenten adressiert werden können.
- Der Dienst muss in der Lage sein, die übermittelten Steuerinformationen an die richtige Geräteinstanz weiterzuleiten.
- Die übertragenen Datenpakete müssen die Zuordnung zur Zielkomponente sowie den anvisierten Wert enthalten.
- Eine Dokumentation der Schnittstelle muss vorliegen, um die Integration in andere Komponenten zu ermöglichen.

Nicht-funktionale Anforderungen

- Der Dienst darf keine gespeicherten Zustände über Zielwerte vorhalten.
- Die Kommunikation muss so gestaltet sein, dass sie eine zuverlässige Zustellung ermöglicht.
- Fehlkonfigurationen oder fehlerhafte Eingaben müssen zu validierbaren Fehlermeldungen führen.

3.2.6 Konfigurationsdienst für Kommunikationsinfrastruktur (So-lace Init)

Dieser Service ist verantwortlich für die initiale Einrichtung der Kommunikationsinfrastruktur im System, insbesondere im Hinblick auf Messaging-Komponenten wie Queues oder themenbasierte Weiterleitungspfade.

Funktionale Anforderungen

- Der Dienst muss eine maschinenlesbare Konfiguration interpretieren können, in der Kommunikationskanäle, Routingregeln und Zugriffspfade definiert sind.

- Auf Basis dieser Konfiguration muss die zugrunde liegende Infrastruktur um definierte Ressourcen ergänzt werden.
- Bereits existierende Konfigurationselemente dürfen dabei nicht überschrieben oder dupliziert werden.

Nicht-funktionale Anforderungen

- Der Dienst muss so gestaltet sein, dass er mehrfach ausgeführt werden kann, ohne die Konsistenz der Kommunikationsstruktur zu gefährden.
- Fehlerhafte Konfigurationseinträge sind zu erkennen, protokollieren und dürfen die Verarbeitung nicht abbrechen.

3.3 Anforderungen an das Backend

Das Backend bildet die zentrale Kommunikationsschnittstelle zwischen Clients, internen Services und der Datenbank. Entsprechend hoch sind die Anforderungen an Struktur, Sicherheit und Stabilität.

3.3.1 Architektur und Technologien

- Das Backend ist als REST-Service zu implementieren.
- Es muss eine modulare, wartbare Architektur aufweisen, die dem Prinzip der Trennung von Zuständigkeiten (Separation of Concerns) folgt.
- Als Technologie-Stack wird eine moderne, performante Sprache wie Rust mit einem Webframework wie actix_web empfohlen.

3.3.2 Authentifizierung und Autorisierung

- Alle Zugriffe auf geschützte Ressourcen müssen durch ein Authentifizierungsverfahren abgesichert werden (z.B. JWT oder OAuth2).
- Autorisierungen auf Zeit wie z.B. Sessions sind nicht erlaubt. Alternativen zu Passwörtern (z.B. Tokens) müssen begrenzt gültig sein.

3.3.3 Fehlerbehandlung und Logging

- Fehlerzustände müssen konsistent behandelt und in einem strukturierten Format an den Client kommuniziert werden.
- Es ist ein mehrstufiges Logging-System zu implementieren, das zwischen Info, Warnung und Fehler unterscheidet.
- Sensible Informationen dürfen in Logs nicht gespeichert werden. Logs müssen zentral gesammelt und gegen Manipulation abgesichert werden.

3.3.4 Skalierbarkeit und Performance

- Das Backend ist zustandslos zu gestalten, um horizontale Skalierung über Load-Balancing zu ermöglichen.
- Die Antwortzeiten für einfache CRUD Operationen sollen im Normalbetrieb unter 100ms liegen.
- Die Architektur soll auf Lastspitzen vorbereitet sein (z.B. durch Queues oder Caching).

3.3.5 Sicherheit

- Gängige Sicherheitslücken (z.B. SQL-Injection, XSS, CSRF) sind durch geeignete Mechanismen zu verhindern.
- Eingaben vom Client sind streng zu validieren und zu sanitieren.

3.3.6 API-Dokumentation

- Die Schnittstellen müssen vollständig dokumentiert werden.
- Eine maschinenlesbare API-Spezifikation (z.B. OpenAPI 3.0) ist zu pflegen.
- Optional kann eine interaktive API-Oberfläche für Entwickler bereitgestellt werden (z.B. Swagger UI).

3.4 Anforderungen an die Datenbank

Die Datenbank dient als persistente Grundlage für alle im System gespeicherten Informationen. Sie muss sowohl performant als auch sicher und konsistent arbeiten.

3.4.1 Modellierung und Struktur

- Das Datenbankschema ist klar zu dokumentieren und mindestens in der 3. Normalform zu entwerfen, sofern nicht durch Performance-Aspekte begründet anders.
- Entitäten und ihre Beziehungen müssen nachvollziehbar und versionierbar abgebildet werden.

3.4.2 Datensicherheit und Integrität

- Sensible Daten (z.B. Passwörter, Tokens) müssen verschlüsselt gespeichert werden.
- Datenbankeigene Mechanismen wie Constraints, Foreign Keys und ggf. Trigger sind zur Sicherstellung der Datenintegrität zu verwenden.
- Referentielle Integrität ist in allen relevanten Tabellen durchzusetzen.

3.4.3 Zugriffskontrolle

- Der Datenbankzugriff erfolgt ausschließlich über definierte Rollen mit minimalen Rechten.
- Es muss zwischen Administrations-, Lese- und Schreibzugriff differenziert werden.
- Externe Dienste erhalten nur selektiven Zugriff auf erforderliche Tabellen.

3.4.4 Backups und Wiederherstellung

- Es ist ein automatisiertes Backup-Konzept zu implementieren, welches tägliche Snapshots sowie inkrementelle Sicherungen vorsieht.
- Wiederherstellungs routinen müssen dokumentiert und regelmäßig geprüft werden.

3.4.5 Performance und Skalierung

- Für häufig genutzte Felder sind geeignete Indizes zu definieren.
- Bei wachsendem Datenvolumen sollen Mechanismen wie Read-Replicas, Sharding oder Partitionierung zum Einsatz kommen.
- Abfragen müssen gezielt optimiert und auf lange Laufzeiten geprüft werden.

3.4.6 Technologischer Rahmen

- Die Datenbanklösung muss Open Source, stabil, transaktionssicher und für hohe Datenmengen geeignet sein.
- Es wird der Einsatz von PostgreSQL empfohlen.

3.5 Anforderungen an das IoT-Gerät

3.5.1 Allgemeine Anforderungen an das IoT-Gerät

Das IoT-Gerät fungiert als zentrale Steuerkomponente innerhalb des automatisierten Bewässerungssystems. Es ist zuständig für die zyklische Erfassung und Verarbeitung von Umweltdaten, die Aktorsteuerung basierend auf festgelegten Zielparametern sowie die gesicherte Kommunikation mit externen Systemen. Neben diesen Kernfunktionen umfasst seine Verantwortung auch die Konfigurierbarkeit des Systems, die Benutzerinteraktion bei der Erstinbetriebnahme und die Ausgabe von Statusinformationen.

Zur Erfüllung dieser Aufgaben muss das IoT-Gerät eine Vielzahl unterschiedlicher Anforderungen erfüllen, die sowohl funktionale als auch nicht-funktionale Aspekte abdecken. Diese Anforderungen werden im Folgenden detailliert analysiert.

3.5.2 Anforderungsanalyse für das IoT-Gerät

Integration und zyklische Abfrage von Sensorik

Das System muss in der Lage sein, Umweltdaten über mehrere physikalisch unterschiedliche Sensoren zu erfassen. Dabei kommen sowohl analoge Signale (z. B. Bodenfeuchtesensor) als auch digitale Schnittstellen (z. B. I²C für Lichtsensor und GPIO für Temperatur- und Feuchtesensoren) zum Einsatz. Die Erfassung erfolgt in regelmäßigen Intervallen durch zyklische Tasks.

Eine besondere Anforderung besteht darin, dass jeder Sensor vor seiner Nutzung initialisiert und zyklisch auf Funktionsfähigkeit überprüft wird. Das System muss erkennen, ob ein Sensor inaktiv ist – etwa durch konstanten Maximalwert oder fehlende Signale – und diesen entsprechend markieren. Dadurch wird verhindert, dass fehlerhafte Sensordaten zu Fehlentscheidungen im Steuerungssystem führen.

Messdatenverarbeitung, Validierung und Zeitstempelung

Die erfassten Rohdaten müssen einer mehrstufigen Verarbeitung unterzogen werden, um zuverlässige, regelbare Kenngrößen zu erhalten. Dabei sind insbesondere folgende Prozesse

umzusetzen:

- **Mittelwertbildung:** Zur Reduktion von Ausreißern wird über mehrere Einzelmessungen ein gleitender Mittelwert berechnet. Dies erhöht die Datenstabilität und verringert den Einfluss kurzfristiger Störungen.
- **Normierung und Umrechnung:** Analoge Rohdaten (z. B. ADC-Werte) werden durch lineare Skalierung in prozentuale Größen oder physikalische Einheiten (z. B. °C, %) umgerechnet. Dies ermöglicht eine einheitliche Bewertungsgrundlage.
- **Plausibilitätsprüfung:** Das System validiert jeden Messwert auf physikalische und technische Plausibilität. Werte außerhalb realistischer Grenzen oder mit fehlender Varianz werden als ungültig klassifiziert.
- **Zeitstempelung:** Jeder Messwert wird mit einem präzisen UTC-Zeitstempel versehen. Diese Zeitmarkierungen sind essenziell für die Synchronisation mit Backend-Systemen sowie für die zeitliche Analyse und Visualisierung der Daten.

Zielwertbasierte Steuerung des Aktors

Die Bodenfeuchtwerte dienen als Auslöser für die Steuerung der Wasserpumpe. Die Steuerung erfolgt nicht binär, sondern adaptiv anhand eines Regelalgorithmus, der eine Pumpdauer in Abhängigkeit der Differenz zwischen gemessenem Feuchtewert und Sollwert bestimmt.

Zur Umsetzung dieser Steuerung sind folgende Anforderungen zu erfüllen:

- **Laufzeitberechnung:** Der Pumpvorgang wird nur ausgelöst, wenn der aktuelle Messwert den definierten Zielwert unterschreitet. Die Pumpdauer ergibt sich aus der Differenz multipliziert mit einem Kalibrierfaktor.
- **Sicherheitsgrenzen:** Um eine Über- oder Unterbewässerung zu verhindern, ist die Pumpdauer durch Minimal- und Maximalwerte begrenzt. Liegt die berechnete Dauer unter dem Minimum, erfolgt kein Pumpvorgang.
- **Entkoppelte Steuerlogik:** Die Ausführung der Pumpensteuerung erfolgt in einer eigenen Task innerhalb des eingesetzten Echtzeitbetriebssystems, die über eine Queue angesteuert wird. Dadurch ist eine zeitlich entkoppelte, thread-sichere Steuerung gewährleistet.

Netzwerkkommunikation und Authentifizierung

Das IoT-Gerät muss in der Lage sein, sich mit einem lokalen Netzwerk zu verbinden und darüber mittels eines publish/subscribe-basierten Kommunikationsprotokolls mit einem entfernten Server oder Message Broker zu kommunizieren. Zusätzlich ist ein sicherer Authentifizierungsmechanismus notwendig, um unautorisierte Zugriffe zu verhindern.

- **Konnektivität:** Das Gerät muss Zugangsdaten speichern und beim Systemstart automatisch versuchen, eine Verbindung mit dem gespeicherten Netzwerk herzustellen.
- **Challenge-Response-Authentifizierung:** Zur Registrierung und Zugriffskontrolle verwendet das System ein sicheres Authentifizierungsverfahren mit symmetrischer Signaturprüfung und zeitlich begrenzten Zugriffsschlüsseln.
- **Strukturierte Datenübertragung:** Sensordaten werden in einem strukturierten, textbasierten Datenformat serialisiert und an den Broker gesendet. Konfigurationsänderungen können in umgekehrter Richtung empfangen, geparsst und verarbeitet werden.

Persistente Speicherung systemrelevanter Daten

Das System muss wichtige Konfigurationsdaten und Zielparameter dauerhaft speichern können, um nach einem Neustart autonom weiterarbeiten zu können. Diese Daten beinhalten:

- Zugangsdaten zu Netzwerk und Broker
- Geräte-Identifikatoren und Modellinformationen
- Zielwerte für Sensorik (Feuchte, Temperatur, etc.)
- Benutzername, Token und Registrierungsstatus

Lokale Konfiguration über Access Point und Webinterface

Bei der Erstinbetriebnahme muss das Gerät ohne externe Tools konfigurierbar sein. Es startet in einem eigenständig betriebenen Konfigurationsmodus und stellt ein Webinterface zur Verfügung, über das der Benutzer Zugangsdaten sowie Benutzerinformationen eingeben kann.

Nach erfolgreicher Konfiguration muss das System automatisch in den Betriebsmodus wechseln. Die Konfiguration wird gespeichert und beim nächsten Systemstart verwendet.

Dieses Setup-Modell erfordert klare Zustandsübergänge und eine zuverlässige Benutzerinteraktion, auch bei Verbindungsabbrüchen oder fehlerhaften Eingaben.

Eine besondere Anforderung besteht darin, dass das Gerät nicht nur im Access-Point-Modus betrieben werden kann, sondern zusätzlich die Fähigkeit besitzt, parallel eine Verbindung zu einem bestehenden drahtlosen Netzwerk aufzubauen. Dieser parallele Dualmodus erlaubt z. B. während des Setup-Prozesses eine nahtlose Rückmeldung über den Verbindungsstatus und ist für einen nutzerfreundlichen Übergang in den Betriebsmodus essenziell.

Systemstatusanzeige und Rücksetzung per Taster

Zur einfachen Diagnose und Nutzerinformation verfügt das Gerät über eine LED, deren Leuchtverhalten verschiedene Systemzustände anzeigt. Zusätzlich ist ein physischer Taster vorgesehen, über den ein kompletter Werksreset ausgelöst werden kann.

- **LED-Rückmeldung:** Eine blinkende LED signalisiert den Setup-Modus, während ein dauerhaftes Leuchten auf eine aktive Netzwerkverbindung hinweist. Die Steuerung erfolgt über eine eigene Task und ist unabhängig von der Hauptlogik.
- **Reset-Taster:** Wird der Taster länger als eine definierte Dauer gedrückt gehalten (z. B. 5 Sekunden), so werden alle gespeicherten Konfigurationen gelöscht, das Netzwerk zurückgesetzt und das Gerät automatisch neu gestartet. Diese Funktion erhöht die Robustheit im praktischen Einsatz und bietet eine Rückfallebene bei Konfigurationsfehlern.

Asynchrone, parallele Ausführung von Systemkomponenten

Die Architektur des Systems muss es ermöglichen, mehrere Aufgaben gleichzeitig und unabhängig voneinander auszuführen. Dies betrifft insbesondere die zyklische Sensorabfrage, die LED-Steuerung, die Pumpensteuerung und Netzwerkoperationen. Die Umsetzung erfolgt über ein Multitasking-fähiges Echtzeitbetriebssystem, wobei jede Hauptfunktionalität in einer eigenen Task ausgeführt wird.

Diese Entkopplung stellt sicher, dass Verzögerungen in einem Teilbereich (z. B. langsame Sensorantworten) nicht den gesamten Systemfluss blockieren. Gleichzeitig ermöglicht sie eine priorisierte Abarbeitung zeitkritischer Operationen.

Die zugrunde liegende Hardwareplattform muss hierfür die gleichzeitige Ausführung mehrerer Tasks mit Prioritätsverwaltung unterstützen. Besonders geeignet sind Systeme mit echter Dual-Core-Architektur, die eine Entlastung zeitkritischer Tasks (z. B. Netzwerk oder Sensorverarbeitung) durch gezielte Lastverteilung auf unterschiedliche Prozessorkerne

ne ermöglichen. Die Unterstützung eines Multitasking-fähigen Echtzeitbetriebssystems (z. B. FreeRTOS) wird vorausgesetzt.

Nicht-funktionale Anforderungen

Ergänzend zu den funktionalen Anforderungen sind mehrere nicht-funktionale Merkmale für einen robusten Einsatz zu erfüllen:

- **Dauerbetrieb und Stabilität:** Das Gerät muss kontinuierlich über lange Zeiträume hinweg funktionsfähig bleiben, auch bei Umwelteinflüssen oder Spannungssfluktuationen.
- **Fehlertoleranz:** Temporäre Verbindungsabbrüche, Sensorsausfälle oder interne Fehler dürfen nicht zum Systemstillstand führen, sondern müssen durch Wiederholungslogik und Rückfallebenen abgefangen werden.
- **Modularität und Wartbarkeit:** Die Softwarearchitektur soll so gestaltet sein, dass zukünftige Erweiterungen, z. B. zusätzliche Sensoren oder Backend-Funktionen, mit minimalem Aufwand integrierbar sind.
- **Effiziente Ressourcennutzung:** Angesichts begrenzter Hardware-Ressourcen auf Mikrocontroller-Ebene sind RAM, CPU-Zeit und Energieverbrauch sparsam und zielgerichtet zu verwenden.
- **Datensicherheit und Integrität:** Übertragene und gespeicherte Daten – insbesondere Token und Konfigurationen – müssen gegen Manipulation und unberechtigten Zugriff geschützt sein.

Das IoT-Gerät bildet das Herzstück des automatisierten Bewässerungssystems und übernimmt sowohl Messung, Steuerung als auch Kommunikation. Die hier analysierten Anforderungen umfassen alle relevanten funktionalen, technischen und organisatorischen Aspekte, die notwendig sind, um einen robusten, sicheren und wartungsarmen Betrieb zu gewährleisten. Sie bilden die Grundlage für die Systemarchitektur und Implementierung, die in den nachfolgenden Abschnitten detailliert behandelt wird.

3.6 Anforderungen an die Peripheriegeräte

3.6.1 Allgemeine Anforderungen an externe Komponenten

Die externen Peripheriegeräte eines automatisierten Bewässerungssystems übernehmen zentrale Aufgaben in der Umweltdatenerfassung, der Ansteuerung von Aktoren sowie der

Stromversorgung einzelner Systembestandteile. Diese Komponenten sind funktional eng mit dem IoT-Gerät gekoppelt, operieren jedoch teils auf voneinander getrennten Stromkreisen und arbeiten mit verschiedenen elektrischen Schnittstellen.

Um einen stabilen, sicheren und skalierbaren Betrieb zu ermöglichen, müssen die eingesetzten Sensoren, Schaltelemente und Energieversorgungsbausteine eine Reihe technischer, funktionaler und physikalischer Anforderungen erfüllen. Im Folgenden werden diese Anforderungen strukturiert analysiert und spezifiziert.

3.6.2 Anforderungen an die Bodenfeuchtemessung

Zur Messung der Bodenfeuchtigkeit wird ein Sensor eingesetzt, der den Feuchtegehalt über die elektrische Leitfähigkeit eines Mediums bestimmt. Die daraus resultierenden Anforderungen betreffen sowohl das elektrische Verhalten als auch die Umgebungsresistenz des Sensors.

Der Sensor muss ein elektrisches Spannungssignal ausgeben, das sich mit geeigneter Auflösung und Genauigkeit durch das Steuergerät digitalisieren und auswerten lässt. Die resultierenden Messwerte sollten eine hinreichende Differenzierung im typischen Feuchtebereich von Zimmerpflanzen (ca. 10–70 %) ermöglichen und dabei möglichst wenig Drift aufweisen.

Da der Sensor dauerhaft in feuchtem Substrat eingesetzt wird, muss er gegen elektrolytische Korrosion resistent sein. Dies betrifft insbesondere die Elektrodenstruktur, welche einer langfristigen Oxidation oder Auflösung entgegenwirken muss. Zusätzlich ist sicherzustellen, dass keine nennenswerten Leckströme vom Sensorsignalpfad in das System zurückkoppeln. Eine elektrische Entkopplung – beispielsweise über Schutzwiderstände oder aktive Puffer – ist empfehlenswert.

Zudem ist die mechanische Robustheit des Sensors zu berücksichtigen. Dieser muss für den dauerhaften Einbau in Pflanzgefäße geeignet sein und darf durch wiederholtes Einsetzen oder Herausziehen keine signifikanten Veränderungen seiner Messeigenschaften zeigen.

3.6.3 Anforderungen an Temperatur- und Luftfeuchtigkeitsmessung

Für die Messung von Temperatur und Luftfeuchtigkeit ist ein kombinierter Sensor vorgesehen, der seine Messwerte über eine digitale Datenleitung an den Mikrocontroller überträgt. Der Sensor verwendet ein zeitkritisches Kommunikationsprotokoll, bei dem ein stabiler Timing-Verlauf und Interruptsteuerung auf der Seite des Controllers erforderlich sind.

Die Messgenauigkeit muss im für Zimmerpflanzen typischen Bereich ausreichend sein. Als Zielwerte gelten $\pm 2^{\circ}\text{C}$ für Temperatur und $\pm 5\%$ relative Luftfeuchte. Die Langzeitsta-

bilität der Messwerte sollte über mehrere Wochen gegeben sein, ohne dass eine Nachkalibrierung notwendig ist.

Aufgrund der relativ geringen Datenrate und des unidirektionalen Kommunikationsverhaltens eignet sich diese Sensorklasse vor allem für einfache Anwendungen mit geringer Messfrequenz. Die Ausfallrate kann durch geeignete Fehlerprüfmechanismen (z. B. Timeout-Überwachung und Retry-Logik) reduziert werden.

3.6.4 Anforderungen an die Lichtstärkemessung

Zur Messung der Lichtintensität im Bereich des sichtbaren Spektrums wird ein Sensor eingesetzt, der über eine serielle Kommunikationsschnittstelle mit dem Steuergerät verbunden ist und eine lux-bezogene Beleuchtungsstärke zurückliefert.

Die Anforderungen umfassen eine zuverlässige Kommunikation über die gewählte digitale Schnittstelle innerhalb der spezifizierten Frequenzbereiche sowie ein geeignetes Adressierungsschema, das keine Konflikte mit anderen Komponenten erzeugt. Eine notwendige Pull-up-Beschaltung auf den Datenleitungen muss systemseitig gewährleistet sein.

Der Messbereich sollte mindestens bis 2000 lx reichen, wobei auch niedrige Intensitäten (unter 100 lx) differenziert erfassbar sein müssen. Die Sensorsauflösung und Dynamik sollten so gewählt sein, dass auch schwache Beleuchtung durch indirektes Sonnenlicht oder künstliche Raumbeleuchtung erkannt werden kann. Eine kontinuierliche Messung bei mäßiger Abtastfrequenz (z. B. alle 10 s) ist ausreichend.

Zudem ist auf eine mechanisch und thermisch stabile Montage zu achten, da Temperaturschwankungen und Lichtabschattungen die Messwerte verfälschen können.

3.6.5 Anforderungen an die Schaltlogik (Relaismodul)

Zur elektrischen Trennung zwischen Mikrocontroller und Pumpe samt separater Spannungsquelle wird ein Relaismodul verwendet, das über eine digitale Steuerspannung geschaltet wird. Das Relaismodul muss mit einer niedrigen Eingangsspannung betrieben werden können und darf im inaktiven Zustand keine Lastspannung auf den Steuerkreis zurückführen.

Wesentliche Anforderungen sind:

- **Kompatibilität:** Die Relaisspule oder deren Ansteuerlogik (z. B. via Optokoppler) muss durch ein digitales Steuersignal mit niedrigem Spannungspegel zuverlässig aktiviert werden können, ohne zusätzliche Verstärker- oder Treiberkomponenten.
- **Trennfestigkeit und Kontaktbelastbarkeit:** Das Relais muss Ströme im Bereich von mindestens 1–2 A schalten können, da elektrische Pumpen bei Inbetriebnahme

kurzzeitig hohe Einschaltströme erzeugen.

- **Sicheres Schaltverhalten:** Das Modul darf keine Prellungen oder unkontrollierten Wiederholimpulse erzeugen. Ein definierter Einschaltimpuls über eine Tasksteuerung ist vorzusehen.
- **Galvanische Trennung:** Zwischen Steuer- und Lastkreis muss eine vollständige galvanische Trennung gewährleistet sein, um Rückkopplungen oder Störungen im Mikrocontroller zu vermeiden.

3.6.6 Anforderungen an den Pumpen-Aktor

Die verwendete Pumpe dient zur kurzzeitigen Wasserförderung und wird in einem Intervallbetrieb aktiviert. Die Anforderungen ergeben sich vor allem aus ihrer elektrischen Belastbarkeit, Laufzeitstabilität und mechanischen Verträglichkeit.

Die Pumpe muss mit einer Versorgungsspannung im Bereich 5–12 V betrieben werden und über Schraub- oder Schlauchanschlüsse für kleine Wassermengen verfügen. Die Fördermenge pro Sekunde muss in einem Bereich liegen, der eine differenzierte Laufzeitsteuerung (typ. 2–10 s) ermöglicht. Eine zu hohe Förderrate würde zu unpräzisem Wasserauftrag führen, eine zu niedrige zu langen Laufzeiten.

Die Stromaufnahme muss innerhalb der durch das Relaismodul spezifizierten Grenzwerte bleiben. Zudem sollte die Pumpe gegen Überhitzung und Blockieren geschützt sein. Ein gewisser Geräuschpegel ist systembedingt tolerierbar, solange er keine funktionalen Einschränkungen verursacht.

3.6.7 Anforderungen an die externe Pumpen-Stromversorgung

Da die Pumpe nicht aus der Mikrocontroller-Stromversorgung betrieben werden kann, ist eine separate Energiequelle erforderlich. Diese muss stabile Spannungs- und Stromwerte liefern, auch bei abruptem Lastwechsel durch das Relais.

Die Ausgangsspannung muss zur Pumpe passen und ausreichend Strom bereitstellen, um Anlauf- und Förderströme zuverlässig bedienen zu können. Dabei sind Schutzmechanismen gegen Überstrom, Verpolung und thermische Belastung vorzusehen. Eine separate Masseführung für Last- und Steuerkreis ist ratsam, um potenzielle Störungen oder Rückströme zu verhindern.

Zur Absicherung empfiehlt sich eine Strombegrenzung über Sicherung oder elektronischen Schalter sowie eine klar getrennte Kabelführung zur Vermeidung elektromagnetischer Einstreuungen in den Mikrocontrollerbereich.

3.6.8 Nicht-funktionale Anforderungen an Peripheriegeräte

Neben den beschriebenen funktionalen Anforderungen ergeben sich für die verwendeten Peripheriegeräte weitere nicht-funktionale Anforderungen, welche maßgeblich zur Betriebssicherheit, Wartbarkeit und Skalierbarkeit des Systems beitragen. Diese betreffen insbesondere qualitative Eigenschaften, die nicht unmittelbar aus der Gerätespezifikation hervorgehen, jedoch für einen stabilen und langlebigen Systembetrieb essenziell sind.

- **Zuverlässigkeit und Lebensdauer:** Alle eingesetzten Komponenten – insbesondere Sensoren und Pumpen – müssen für einen Dauerbetrieb unter haushaltsüblichen Bedingungen geeignet sein. Dazu gehört die Fähigkeit, über viele Zyklen hinweg mechanisch, thermisch und elektrisch stabil zu funktionieren, ohne dass Messdrift, Kontaktprobleme oder mechanischer Verschleiß zu Funktionsverlust führen.
- **Störfestigkeit:** Sensoren und Schnittstellen sollten gegenüber elektromagnetischen Störungen, Leitungslängen oder Spannungsschwankungen möglichst unempfindlich sein. Dies betrifft insbesondere analoge Signale und I²C-Verbindungen, bei denen Pull-up-Widerstände und EMV-gerechtes Routing von besonderer Bedeutung sind.
- **Installationsfreundlichkeit:** Die Peripheriegeräte sollen einfach installierbar und ersetzbar sein. Dies betrifft etwa die Länge und Robustheit von Anschlussleitungen, die Montagemöglichkeiten für Sensoren (z. B. in Blumentöpfen) sowie die Standardisierung von Steckverbindungen oder Lötpads.
- **Wartungsarmut:** Die Komponenten sollen möglichst wenig Pflege bedürfen. Beispielsweise darf die Pumpe keine regelmäßige Entlüftung oder Reinigung erfordern, Sensoren sollen nicht durch Bodenkontakt oder Feuchtigkeit ausfallen. Korrosionsbeständige Materialien und versiegelte Gehäuse sind bevorzugt.
- **Kompatibilität und Austauschbarkeit:** Die eingesetzten Bauteile sollten möglichst marktüblich und standardisiert sein. So kann bei einem Defekt ein Austausch mit minimalem Anpassungsaufwand erfolgen. Dies betrifft sowohl mechanische Bauformen (z. B. Sensorlänge, Schlauchdurchmesser) als auch elektrische Schnittstellen.
- **Verfügbarkeit und Kostenbewusstsein:** Die Peripheriekomponenten sollen auch in Kleinserien oder Einzelstückzahlen gut verfügbar sein. Sie dürfen keine proprietären oder schwer beschaffbaren Spezialbauteile enthalten, die Ersatz oder Erweiterung behindern würden.

Die externen Peripheriegeräte eines Bewässerungssystems müssen funktional und elektrisch auf das zentrale Steuergerät abgestimmt sein. Ihre Auswahl und Integration basie-

ren auf definierten Anforderungen hinsichtlich Signaltyp, Schnittstelle, Stromversorgung, mechanischer Verträglichkeit sowie elektrischer Trennung. Insbesondere die Kombination aus analoger und digitaler Sensorik, gepulster Aktorik über Relais sowie externer Versorgung erfordert eine systematische Auslegung und Absicherung der Komponenten. Die in diesem Kapitel formulierten Anforderungen stellen sicher, dass die Peripherie langfristig stabil, präzise und betriebssicher in das Gesamtsystem eingebunden werden kann.

4 Auswahl der Technologien

4.1 Auswahl von Vue.js für die Implementierung

Im Rahmen der Entwicklung eines webbasierten Frontends für ein intelligentes Bewässerungssystem fiel die Wahl auf Vue.js. Die Entscheidung begründet sich auf mehreren Faktoren:

4.1.1 Modularität und Komponentenstruktur

Vue ermöglicht eine klare Trennung von Funktionalität, Darstellung und Stil durch das Single-File-Component-Modell. Dies unterstützt die Wiederverwendbarkeit und die Wartbarkeit in mittelgroßen Anwendungen wie der vorliegenden [64]. Komponenten lassen sich dabei hierarchisch strukturieren, durch Props und Events miteinander verknüpfen und flexibel wiederverwenden. Die damit verbundene Modularität ist ein zentraler Vorteil im Vergleich zu klassischen Monolith-Strukturen.

4.1.2 Reaktivität und Datenbindung

Die Composition API in Vue 3 erlaubt die strukturierte Wiederverwendung von Logik und bietet eine feinere Kontrolle über Komponentenlebenszyklen. Die Reaktivierung ist deklarativ und effizient, was zu einer reduzierten Komplexität führt [65]. Im Gegensatz zur eher komplexen Reaktivierung in Angular oder den teils manuell zu verwaltenden Hooks in React bietet Vue ein konsistentes Modell, das einfacher zu testen und zu debuggen ist [20]. Insbesondere die automatische DOM-Synchronisierung bei Zustandsänderungen verringert den Entwicklungsaufwand erheblich.

4.1.3 Community, Dokumentation und Lernkurve

Im Vergleich zu Angular bietet Vue eine flachere Lernkurve und ist dennoch umfangreicher als React in seiner Grundausstattung. Besonders für kleine bis mittelgroße Teams ohne dedizierte DevOps- oder Backend-Abteilungen eignet sich Vue durch seine einfache

Integration und das konsistente Ökosystem [66]. Die offizielle Dokumentation von Vue gilt als eine der besten im Bereich der Webframeworks und trägt wesentlich zur schnellen Produktivität bei [64]. Hinzu kommt ein aktives Community-Umfeld mit einer Vielzahl an Open-Source-Bibliotheken und Erweiterungen.

4.1.4 Flexibilität, Integration und Zukunftssicherheit

Ein weiterer Vorteil von Vue ist seine hohe Flexibilität im Hinblick auf Tooling und Integration. Vue-Projekte können leicht mit modernen Build-Tools wie Vite kombiniert werden, welches durch schnelle Entwicklungszyklen und modulare Hot-Reloading-Mechanismen eine effiziente Frontentwicklung ermöglicht. Durch die strikte Trennung von View- und Logikschicht lässt sich Vue problemlos mit REST-APIs, GraphQL oder WebSockets kombinieren. Zudem wird Vue kontinuierlich weiterentwickelt: Die Long-Term-Support-Strategie sowie eine klare Roadmap sprechen für eine hohe technologische Zukunftssicherheit.

4.1.5 Vergleich: Options API vs. Composition API in Vue.js

Vue.js unterstützt zwei zentrale Paradigmen zur Strukturierung von Komponenten: die klassische Options API und die moderne Composition API. Beide Modelle ermöglichen die Definition von Zuständen, Methoden, Lebenszyklus-Hooks und Reaktivität innerhalb einer Single File Component (SFC), unterscheiden sich jedoch fundamental im Aufbau und in der Ausdrucksstärke.

Options API Die Options API stellt das klassische und lange Zeit dominante Paradigma zur Definition von Komponenten in Vue.js dar. Ihr zentraler Vorteil liegt in der klar strukturierten Gliederung der Komponentenlogik nach spezifischen Optionen wie `data`, `methods`, `computed`, `watch` und Lebenszyklus-Hooks. Diese Trennung erleichtert insbesondere Einsteigerinnen und Einsteigern den Zugang zur komponentenbasierten Entwicklung, da die Zuständigkeiten der einzelnen Blöcke unmittelbar nachvollziehbar sind. Die durchgängige Unterstützung in der offiziellen Vue.js-Dokumentation sowie in zahlreichen Community-Plugins trägt zusätzlich zur Zugänglichkeit und zur breiten Akzeptanz dieses Modells bei. Auch historisch bedingt ist die Options API weiterhin vollständig kompatibel und wird aktiv gepflegt, was ihre Relevanz in bestehenden Projekten unterstreicht [64].

```

1 export default {
2   data() {
3     return {
4       counter: 0
5     }
6   },
7   methods: {
8     increment() {
9       this.counter++
10    }
11  }
12}

```

Listing 4.1: Beispiel Options API

Den Vorteilen stehen jedoch mehrere signifikante Einschränkungen gegenüber. Insbesondere bei wachsender Komplexität einer Komponente stößt die Options API an strukturelle Grenzen. Da Zustände und zugehörige Methoden nach Typ gruppiert und nicht funktional zusammenhängend strukturiert werden, entsteht bei umfangreicher Logik schnell eine fragmentierte Darstellung. Diese Fragmentierung erschwert nicht nur die Lesbarkeit, sondern auch die Wartbarkeit und Wiederverwendbarkeit von Code – vor allem dann, wenn sich Logik über mehrere Komponenten hinweg wiederholt. Zudem leidet die Options API unter einer eingeschränkten Typsicherheit im Umgang mit TypeScript, da Kontextinformationen wie `this` nicht ohne Weiteres typensicher aufgelöst werden können. Dies kann zu Laufzeitfehlern führen und behindert die statische Analyse durch TypeScript-Compiler [64].

Composition API Die mit Vue 3 eingeführte Composition API bietet eine moderne und hochgradig modulare Alternative zur klassischen Options API. Sie zielt insbesondere auf eine bessere Wiederverwendbarkeit und thematische Gruppierung von Logik ab. Ein zentrales Merkmal ist, dass Zustände, Methoden und Nebenwirkungen innerhalb der `setup()`-Funktion definiert werden. Dadurch lassen sich zusammenhängende Funktionsblöcke logisch gruppieren und als sogenannte Composables wiederverwenden. Dies erhöht die Wartbarkeit bei wachsender Komponentenkomplexität erheblich [67].

Ein weiterer Fortschritt besteht in der Einführung des sogenannten `<script setup>`-Blocks, der in Vue 3 als syntaktischer Zucker (syntactic sugar) über der Composition API liegt. Im Gegensatz zur expliziten Verwendung von `setup()` in einem klassischen `<script>`-Block vereinfacht `<script setup>` die Struktur, reduziert Boilerplate-Code und macht die Komponenten deklarativer und kompakter. Dabei werden alle im `<script setup>` definierten Variablen automatisch im Template verfügbar gemacht, ohne dass ein `return` erforderlich ist [68].

```

1 <script setup lang="ts">
2   import { ref } from 'vue'
3   const counter = ref(0)
4   const increment = () => { counter.value++ }
5 </script>

```

Listing 4.2: Beispiel für die Composition API mit `<script setup>`

Hier wird ein reaktiver Zustand `counter` über die Funktion `ref()` erstellt, welcher automatisch mit dem DOM synchronisiert wird. Die Funktion `increment` verändert diesen Zustand und steht im Template zur Verfügung.

Die Vorteile dieses Ansatzes liegen auf der Hand: Logik ist gruppiert, leicht extrahierbar und testbar, insbesondere durch Composables. Darüber hinaus bietet die Composition API eine exzellente Typsicherheit, insbesondere im Zusammenspiel mit TypeScript [65].

Allerdings ergeben sich auch gewisse Herausforderungen. Für Neulinge kann der reduzierte strukturelle Rahmen der `setup()`-Funktion zunächst weniger Orientierung bieten als die Options API. Zudem besteht bei unstrukturierter Nutzung die Gefahr einer unübersichtlichen, flachen Anordnung vieler Logikelemente ohne klare Gruppierung, was die Lesbarkeit und Wartbarkeit negativ beeinflussen kann [67].

Während die Options API ihre Stärken in der Klarheit und dem geringeren Einstieg hat, bietet die Composition API insbesondere in Kombination mit `<script setup>`, eine moderne, typsichere und wiederverwendbare Struktur für Vue-Komponenten - insbesondere für mittlere bis große Anwendungen mit komplexer Zustandslogik [67].

Einordnung im Projektkontext In der vorliegenden Anwendung wurde bewusst die Composition API eingesetzt, da sie sich durch eine deklarative, modulare und testfreundliche Struktur auszeichnet. Besonders in Kombination mit Composables, wie etwa für API-Zugriffe oder Formularvalidierungen, ließ sich dadurch eine bessere Trennung von Logik und Darstellung erreichen. Die Integration mit dem State-Management-Tool Pinia ist ebenfalls eng an die Composition API gekoppelt, was die Konsistenz im Projekt stärkt.

4.1.6 Einschränkungen und Gegenmaßnahmen

Vue bringt zwar Einschränkungen hinsichtlich Search-Engine Optimization (SEO) mit sich, insbesondere ohne SSR. Diese lassen sich jedoch durch Techniken wie Prerendering oder den Einsatz von Frameworks wie Nuxt.js abmildern. Für komplexes State-Management stehen mit Pinia und Vuex leistungsfähige Bibliotheken zur Verfügung [69].

Insgesamt stellt Vue.js einen geeigneten Kompromiss zwischen Einfachheit, Leistung und Erweiterbarkeit dar und erweist sich als besonders geeignet für domänenpezifische

Anwendungen mit moderater Komplexität. Die Balance zwischen Einstiegstauglichkeit und technischer Tiefe macht das Framework sowohl für Lernzwecke als auch für professionelle Webentwicklung attraktiv.

4.2 Einsatz von Tailwind CSS und Shadcn-Vue

Moderne Webentwicklung nutzt zunehmend Utility-First-CSS-Frameworks wie Tailwind CSS und komponentenbasierte UI-Bibliotheken wie Shadcn-Vue zur effizienten Erstellung ansprechender Benutzeroberflächen. Diese Werkzeuge bieten im Kontext von Vue.js signifikante Vorteile hinsichtlich Geschwindigkeit, Wartbarkeit und Designkonsistenz [70], [71].

4.2.1 Tailwind CSS: Utility-First-Ansatz

Tailwind CSS ist ein Utility-First-CSS-Framework, das es Entwicklern erlaubt, direkt im HTML Code Styling zu betreiben, anstatt klassische CSS-Klassen zu definieren. Dies erleichtert die Wiederverwendbarkeit und reduziert übermäßige Stylesheet-Komplexität [70].

Vorteile:

- **Performance:** Tailwind eliminiert ungenutztes CSS im Produktionsbuild.
- **Responsives Design:** Vordefinierte Breakpoints erleichtern die Anpassung an verschiedene Bildschirmgrößen.
- **Designkonsistenz:** Einheitliche Farben, Abstände und Größen durch Konfiguration.

Integration in Vue.js Durch vordefinierte Klassen lässt sich Tailwind optimal in Vue-Komponenten einbetten. Der Code bleibt deklarativ und wartbar. Mit Tailwind können komplexe Layouts wie Grid-basierte Dashboards oder mobile Navigationsleisten ohne zusätzliche CSS-Dateien umgesetzt werden.

4.2.2 Shadcn-Vue: Komponentenbibliothek für Vue 3

Shadcn-Vue ist ein Community-getriebener Port der beliebten React-Bibliothek Shadcn/UI für Vue 3. Sie basiert auf Radix UI und Tailwind CSS, womit sie eine hohe Anpassbarkeit bei gleichzeitiger Konsistenz gewährleistet [71].

Modularer Aufbau Im Gegensatz zu anderen UI-Bibliotheken installiert Shadcn-Vue Komponenten selektiv. Dies führt zu einer schlankeren Anwendung und erhöht die Kontrolle über das Styling.

Beispiele aus dem Projekt Die Anwendung verwendet u.a. folgende Shadcn-Komponenten:

- `Tabs`, `Cards`, `DropdownMenus` in der Pflanzen- und Sensordarstellung
- `Switches` und `Tooltips` in der Detailansicht und den Einstellungen
- `AlertDialogs` für Benutzerbestätigungsprozesse

4.2.3 Kombination beider Technologien im Frontend-Projekt

Tailwind CSS dient als stilistisches Fundament, während Shadcn-Vue darauf aufbaut und vorgefertigte interaktive Komponenten zur Verfügung stellt. Das Ergebnis ist eine moderne, performante und barrierearme Benutzeroberfläche, die sowohl für Desktop als auch für mobile Endgeräte optimiert ist.

Fazit Die Kombination von Tailwind CSS und Shadcn-Vue stellt eine moderne und leistungsfähige Lösung für das Design von Vue.js-basierten Single-Page-Anwendungen dar. Sie reduziert Entwicklungsaufwand, erhöht die Designkonsistenz und verbessert die Nutzererfahrung signifikant [22], [70].

4.3 Internationalisierung der Anwendung in Vue.js

Die Internationalisierung (i18n) ist ein zentrales Gestaltungsprinzip moderner Webanwendungen, das es erlaubt, Benutzeroberflächen an verschiedene Sprachen, Kulturräume und regionale Konventionen anzupassen. Sie leistet einen essenziellen Beitrag zur globalen Zugänglichkeit und Nutzerzufriedenheit, indem sie Inhalte sprachlich und kontextuell an die Erwartungen der NutzerInnen anpasst [72].

4.3.1 Motivation und Bedeutung

Internationale NutzerInnen erwarten digitale Produkte in ihrer Muttersprache und kulturell vertrauten Darstellung. Studien und Marktanalysen belegen, dass eine Lokalisierung von Inhalten die Nutzungsbereitschaft sowie die Conversion Rates deutlich erhöht. Zudem werden Missverständnisse vermieden, die sich aus abweichenden Symbolsystemen, Farbcodes oder Datums- und Zahlenformaten ergeben können [72].

Auch aus Sicht der Systemarchitektur trägt Internationalisierung zur Skalierbarkeit und Nachhaltigkeit digitaler Anwendungen bei. Internationale Standards wie Unicode und Locale-Formate ermöglichen eine konsistente Darstellung über verschiedene Plattformen hinweg. Die konsequente Trennung von Quellcode und Textressourcen gilt dabei als Best Practice [73].

4.3.2 Technologiewahl: vue-i18n

Zur Umsetzung der sprachlichen und regionalen Anpassung wurde in der vorliegenden Vue.js-Anwendung das etablierte Plugin `vue-i18n` eingesetzt. Es ist der De-facto-Standard im Vue-Ökosystem zur Internationalisierung und bietet eine modulare Integration für Komponenten, Routing und State Management [73]. Das Plugin erlaubt unter anderem:

- Sprachumschaltung zur Laufzeit mit Reaktivität.
- Einbindung sprachspezifischer JSON-Ressourcen.
- Nutzung der `Intl`-API zur Formatierung von Datum, Uhrzeit und Zahlen.
- Unterstützung von Platzhaltern, Mehrzahlformen und dynamischen Textkomponenten [74].

4.3.3 Fazit

Die Nutzung von `vue-i18n` erwies sich im Projektkontext als leistungsfähig, flexibel und zukunftssicher. Durch die Verbindung mit nativen JavaScript-Standards (`Intl`) sowie die einfache Einbindung in die Vue-Komponentenarchitektur konnten sprachliche Anforderungen effizient umgesetzt werden. Die gewählte Lösung orientiert sich an etablierten Best Practices und stellt eine skalierbare Grundlage für weitere Internationalisierungsschritte dar.

4.4 State-Management mit Pinia und Persistenz

Ein zentraler Bestandteil moderner Frontend-Anwendungen ist das effiziente und wartbare Zustandsmanagement. Im Kontext der Webanwendung zur Steuerung eines smarten Bewässerungssystems kommt die State-Management-Bibliothek `Pinia` zum Einsatz. Dieser Abschnitt behandelt die Motivation, Konzeption und Persistierung des globalen Applikationszustands mittels Pinia in Vue.js.

4.4.1 Einordnung von State Management in SPAs

In Single Page Applications liegt die Verantwortung für das Daten- und UI-Management vollständig im Frontend. Daraus ergeben sich Anforderungen wie die zentrale Verwaltung globaler Zustände etwa für eingeloggte Benutzer oder verbundene Geräte, die Wiederverwendung von Daten über Komponenten hinweg, die Synchronisierung mit Backend-Endpunkten sowie die Notwendigkeit einer dauerhaften Speicherung über Seiten reloads hinaus. Pinia, als moderne und offizielle Ablösung von Vuex in der Vue-3-Welt, erfüllt diese Anforderungen durch seine modulare, typesichere und reaktive Struktur [75] [66].

4.4.2 Persistenz mit `pinia-plugin-persistedstate`

Ein zentrales Merkmal der Anwendung ist die Fähigkeit, den Zustand auch bei einem Seitenreload zu bewahren. Dies wird mithilfe des Plugins `pinia-plugin-persistedstate` erreicht, das es ermöglicht, ausgewählte Stores automatisch im `localStorage` des Browsers zu speichern [69]. Jeder Store konfiguriert explizit, ob und wie persistiert wird.

4.4.3 Bewertung und Grenzen

Im praktischen Einsatz zeigt Pinia deutliche Vorteile hinsichtlich Strukturierung und Wartbarkeit des Zustands. Die modulare Trennung erlaubt eine gute Übersicht und fördert die Wiederverwendbarkeit einzelner Logikkomponenten. Durch TypeScript wird zudem eine hohe Typsicherheit erzielt, was insbesondere in größeren Projekten die Fehleranfälligkeit reduziert. Auch das Debugging gestaltet sich dank der Integration in die Vue DevTools effizient. Die Persistenz erlaubt Offline-Szenarien und schützt vor ungewolltem Datenverlust. Eine Herausforderung besteht in der Handhabung verschachtelter Ressourcenbeziehungen. So referenziert eine Pflanze beispielsweise ihren Raum, dieser wiederum eine Gruppe. Dieses potenzielle Problem wird in der Anwendung durch gezielte Backend-abfragen und das gezielte Laden zusammengehöriger Ressourcenstrukturen bei Bedarf entschärft.

Fazit

Pinia mit Persistenz stellt ein robustes und wartbares State-Management für Vue-Frontends bereit. Es erfüllt die Anforderungen an Konsistenz, Wiederverwendbarkeit, Performanz und Sicherheit und bildet damit die solide Grundlage für ein reaktives und benutzerfreundliches Smart-Gardening-System.

4.5 Mobile Kompilierung mit Capacitor

Ein zentrales Ziel der entwickelten Anwendung ist ihre Nutzbarkeit nicht nur als Web-App, sondern auch als mobile Applikation auf Android-Geräten. Dies wird durch die Integration von Capacitor ermöglicht.

4.5.1 Funktionsweise von Capacitor

Capacitor fungiert als moderne Brückentechnologie zwischen Web-Technologien und nativer Funktionalität. Die Vue-Anwendung wird dabei in eine WebView eingebettet und über ein Plugin-System mit nativen Funktionen verbunden, was eine hybride Nutzung nativer Hardware-Ressourcen und Web-Technologien erlaubt [76], [77]. Der Build-Prozess beginnt mit dem Erzeugen des Produktions-Builds der Vue-App mittels `vite build`. Anschließend wird mit dem Befehl `npx cap add android` ein natives Android-Projekt erstellt. Die WebAssets der Anwendung werden mit `npx cap sync` in das Android-Projektverzeichnis `android/app/src/main/assets` überführt. Danach kann die App entweder direkt in Android Studio geöffnet oder mit der Kommandozeile gestartet werden.

4.5.2 Nutzung nativer APIs

Capacitor stellt eine Reihe nativer APIs zur Verfügung, die aus der Vue-Anwendung heraus direkt verwendet werden können. Dazu zählen unter anderem der Zugriff auf die Kamera, auf das Dateisystem sowie UI-Funktionen wie StatusBar, SplashScreen und Toast-Benachrichtigungen [76]. Diese APIs werden durch offizielle `@capacitor`-Plugins bereitgestellt und erlauben den Zugriff auf Systemfunktionen, ohne dass plattformspezifischer Code geschrieben werden muss.

4.5.3 Besonderheiten und Herausforderungen

Bei der Entwicklung für mobile Geräte ergeben sich mehrere technische Herausforderungen. So muss der Vue Router im sogenannten "History-Modus" betrieben werden, da es sonst zu Problemen mit Deep-Links unter Android kommen kann [78]. Bei falscher Konfiguration können Routen nicht korrekt aufgelöst werden, was 404-Fehler zur Folge hat. Zudem führt die Nutzung der Bildschirmtastatur auf Mobilgeräten dazu, dass Eingabefelder teilweise verdeckt werden. Dieses Verhalten muss durch gezielte Event-Behandlung oder gestalterische Workarounds ausgeglichen werden. Auch der Zugriff auf Systemfunktionen wie Kamera oder Medien setzt unter neueren Android-Versionen explizite Berechtigungsabfragen voraus, die sorgfältig umgesetzt werden müssen [76].

4.5.4 Vor- und Nachteile der Verwendung von Capacitor

Der Einsatz von Capacitor bringt sowohl Vorteile als auch Einschränkungen mit sich. Ein wesentlicher Vorteil besteht darin, dass die Anwendung weiterhin auf Web-Technologien basiert und somit eine einheitliche Codebasis für Web- und Mobile-Plattformen verwendet werden kann. Zudem unterstützt Capacitor Hot Reloading, was die Entwicklung und das Debugging deutlich beschleunigt [76]. Die Anwendung kann darüber hinaus nicht nur als native App, sondern auch als Progressive Web App (PWA) bereitgestellt werden. Auf der anderen Seite ergeben sich durch die Nutzung der WebView geringere Performancewerte im Vergleich zu vollständig nativen Anwendungen [77]. Auch ist der Zugriff auf bestimmte Systemfunktionen eingeschränkt, und die Pflege des nativen Projekts setzt Kenntnisse in Android Studio und dem Android-Entwicklungsprozess voraus.

4.5.5 Bewertung für das Projekt

Für die im Rahmen dieser Arbeit entwickelte Anwendung stellt Capacitor eine sinnvolle und praktikable Lösung dar. Da die gesamte Logik auf Web-Komponenten basiert und der Bedarf an nativer Funktionalität auf wenige Aspekte wie Kameranutzung beschränkt ist, bietet Capacitor eine effiziente Möglichkeit, eine mobile Version mit minimalem Mehraufwand bereitzustellen. Die Integration in den Entwicklungsworkflow verläuft nahtlos, wodurch die mobile Erweiterung der smarten Bewässerungssteuerung sowohl benutzerfreundlich als auch wartbar bleibt.

4.6 Wahl der Programmiersprache

In einem großen Projekt wie Sensorsa ist es notwendig, einheitliche Richtlinien und Guidelines zu etablieren. Dies sorgt dafür, dass einige grundlegende Punkte von allen gleich gehandhabt werden, was wiederum die schnelle Entwicklung, Weiterentwicklung und Wartung von Software fördert und die Benutzbarkeit von Software oder Softwaremodulen erhöht.

Einer der zentralen Aspekte, die unter diesen Richtlinien fallen, ist die Wahl der Programmiersprache. Innerhalb des Sensorsa-Teams gibt es mannigfaltige Fähigkeiten. Eine Bestandsaufnahme der Kenntnisse ergab, dass Python von allen Entwicklern geschrieben und gelesen werden kann. Für stark fragmentierte Module sollte jedoch eine performante Lösung geschaffen werden, die große Lasten leichter und somit kostengünstiger trägt.

Die Bestandsaufnahme lässt schließen, dass Go die dafür am stärksten vertretene Sprache unter den Entwicklern ist. Benchmarks haben jedoch ergeben, dass Rust die deutlich

performantere Sprache ist. [79] Nachfolgend werden die technischen Besonderheiten beider Sprachen diskutiert.

4.6.1 Ein Einblick in Go

Go, auch bekannt als Golang, ist eine moderne Programmiersprache, die in den letzten Jahren stetig an Popularität gewonnen hat. [80] Sie wurde von Google entwickelt und zielt darauf ab, eine klare und prägnante Syntax zu bieten, die gleichzeitig hohe Leistung und Skalierbarkeit ermöglicht. Mit den komplexen Anforderungen an Softwareprodukte und den vielen kooperierenden Teams war die Notwendigkeit einer effizienten und gut skalierbaren Sprache entscheidend. Go sollte insbesondere das Problem der langen Kompilierungszeiten und der Schwierigkeit, parallelen Code zu schreiben, lösen.

Syntax und Struktur

Go zeichnet sich durch eine minimalistische und leicht verständliche Syntax aus. Die Grundstruktur eines Go-Programms ist einfach und übersichtlich, was die Einarbeitung für Entwickler erleichtert. Ein einfaches Hello-World-Programm in Go sieht wie folgt aus:

```
1 package main
2
3 import "fmt"
4
5 func main() {
6     fmt.Println("Hello, World!")
7 }
```

Dieser Code illustriert einige grundlegende Konzepte von Go: die Verwendung von Paketen, die explizite Importierung von Bibliotheken und die Definition der main-Funktion als Einstiegspunkt des Programms.

Der Go-Compiler

Ein wesentlicher Grund für die Effizienz von Go ist der Go-Compiler. Go verwendet einen statischen Typenchecker, was bedeutet, dass Typen zur Kompilierungszeit überprüft werden. Dies trägt zur frühzeitigen Fehlererkennung bei und verbessert die Zuverlässigkeit des Codes. Der Compiler selbst ist äußerst schnell und ermöglicht es Entwicklern, ihre Programme in kürzester Zeit zu kompilieren. Diese Schnelligkeit fördert eine agile Entwicklungsweise, da Änderungen schnell getestet werden können.

Goroutines und Concurrency

Ein herausragendes Merkmal von Go ist die native Unterstützung für Concurrency, also die Fähigkeit, mehrere Aufgaben gleichzeitig auszuführen. Go erreicht dies durch sogenannte Goroutines, die mit dem Schlüsselwort go vor einem Funktionsaufruf gestartet werden:

```
1 go func() {  
2     fmt.Println("Concurrent task")  
3 }()
```

Goroutines sind wesentlich ressourcenschonender als herkömmliche Threads und ermöglichen es, viele davon in einer Anwendung zu betreiben, ohne den Speicher zu überlasten. Die Kommunikation zwischen Goroutines erfolgt über Channels, ein weiteres einzigartiges Konzept von Go, das die Synchronisation und den Datenaustausch vereinfacht. [81]

Was Go ausmacht

Go wurde entwickelt, um sowohl einfach zu erlernen als auch effizient in der Anwendung zu sein. Die Sprache verzichtet bewusst auf komplexe Features wie Vererbung, die in anderen Programmiersprachen oft zu einer steilen Lernkurve führen können. Stattdessen setzt Go auf Komposition, was zu einer besseren Lesbarkeit und Wartbarkeit des Codes führt.

Ein weiterer Vorteil von Go ist die umfangreiche Standardbibliothek, die viele der gängigen Aufgaben der Softwareentwicklung abdeckt, von der Dateiverarbeitung über Netzwerkkommunikation bis hin zur Kryptographie. Diese Standardbibliothek trägt zur Konsistenz und Zuverlässigkeit bei, da Entwickler nicht auf externe Bibliotheken angewiesen sind, die möglicherweise weniger gut gewartet werden.

Go bietet zudem eine hervorragende Cross-Platform-Kompatibilität. Der Go-Compiler erzeugt ausführbare Dateien, die auf verschiedenen Betriebssystemen laufen können, ohne dass Änderungen am Quellcode erforderlich sind. Diese Fähigkeit, plattformübergreifende Anwendungen zu erstellen, ist ein entscheidender Vorteil in einer Umgebung, in der Anwendungen auf verschiedenen Systemen bereitgestellt werden müssen. [82]

Anwendungsbereiche von Go

Go wird in verschiedenen Anwendungsbereichen eingesetzt, die Leistung, Parallelität und Skalierbarkeit priorisieren. Typische Einsatzgebiete umfassen:

Backend-Entwicklung: Go ist ideal für die Entwicklung von Serveranwendungen, insbesondere für Web- und API-Server. Seine Fähigkeit, viele parallele Anfragen zu verarbeiten, macht es zu einer hervorragenden Wahl für hoch skalierbare Systeme.

Cloud-Computing: Dank seiner Effizienz und Einfachheit ist Go eine bevorzugte Sprache für Cloud-native Anwendungen und Microservices.

Netzwerkdienste: Go eignet sich hervorragend für die Entwicklung von Netzwerkdiensten wie Proxies, Gateways und Load Balancers, da es leistungsstarke Netzwerkbibliotheken und native Unterstützung für Concurrency bietet.

Es gibt jedoch auch Anwendungsbereiche, in denen Go weniger geeignet ist:

High-Performance-Grafik und Spieleentwicklung: Obwohl Go schnell ist, gibt es speziellere Sprachen wie C++ oder Rust, die besser für die extremen Leistungsanforderungen der Grafik- und Spieleentwicklung optimiert sind, eine breitere Palette an spezifischen Bibliotheken bieten und besser von Game-Engines unterstützt werden.

Rapid Prototyping und Skripting: Für schnelle Prototypen oder einfache Skripte sind dynamische Sprachen wie Python oder JavaScript aufgrund ihrer Flexibilität und der großen Anzahl an verfügbaren Bibliotheken oft die bessere Wahl.

Grafische Benutzeroberflächen: Go bietet keine einfache Möglichkeit, eine grafische Benutzeroberfläche (GUI) zu erstellen. Hier sind Sprachen wie C# im .NET-Framework oder Java geeigneter.

[83]

4.6.2 Einblicke in Rust

Rust ist eine moderne, systemnahe Programmiersprache, die mit dem Ziel entwickelt wurde, Speichersicherheit, hohe Performance und nebenläufige Programmierung ohne Laufzeit-Overhead zu ermöglichen. Sie wurde ursprünglich von Mozilla Research initiiert und 2015 in Version 1.0 veröffentlicht. Im Zentrum steht ein Ownership-Modell, das Speicherfehler wie Null-Pointer, Use-after-free oder Datenrennen zur Compile-Zeit verhindert – ohne Garbage Collector. Damit schließt Rust eine Lücke zwischen sicherem Hochsprachenkomfort und der Kontrolle traditioneller Low-Level-Sprachen. In der heutigen Softwarelandschaft wird Rust zunehmend in Bereichen wie WebAssembly, Embedded Systems, Systemtools und Backend-Entwicklung eingesetzt. Als Alternative zu C/C++ und Go vereint Rust Systems Programming mit modernen Sprachfeatures und rückt dadurch ins Zentrum aktueller Softwareentwicklung.

4.6.3 Syntax und Struktur

Rust kombiniert eine moderne, präzise Syntax mit klarer Struktur. Die Sprache ist ausdrucksstark, stark typisiert und legt großen Wert auf Lesbarkeit und Sicherheit. Funktionen, Kontrollstrukturen und Fehlerbehandlung orientieren sich an funktionalen und systemnahen Paradigmen. Die Standardstruktur eines Rust-Programms besteht aus Funktionen, Modulen und typstarken Definitionen. Dabei ist der Einstieg einfach und direkt.

```
1 fn main() {  
2     println!("Hello, world!");  
3 }
```

Der Code ist kompakt, typsicher und führt ohne Boilerplate direkt zu einer lauffähigen Applikation.

Diese minimalistische Struktur zeigt Rusts Ziel, Klarheit mit technischer Kontrolle zu verbinden – vom kleinen Programm bis zur komplexen Anwendung.

Grundprinzipien und Sprachdesign von Rust

Kein Garbage Collector: Warum und wie das funktioniert Rust garantiert Speichersicherheit ohne Garbage Collector (GC). Statt automatischer Laufzeitüberwachung verwaltet der Compiler den Speicher zur Compile-Zeit mithilfe des Ownership-Systems. Jede Variable besitzt genau einen Eigentümer; beim Transfer von Ownership wird sicher gestellt, dass es keine doppelten Freigaben oder Dangling Pointers gibt. Referenzen unterliegen Borrowing-Regeln, die eine gleichzeitige, aber kontrollierte Nutzung ermöglichen. Diese statische Analyse erfolgt im sogenannten Borrow Checker, der Teil der rustc-Pipeline ist [84]. Dadurch wird Speicher deterministisch freigegeben, ohne Garbage Collection oder manuelles free.

Performance durch Systemnähe und zero-cost abstractions Rust wird direkt zu Maschinencode kompiliert und verzichtet auf eine virtuelle Maschine oder Interpretations schicht. Die Sprache erlaubt es, Low-Level-Operationen präzise zu steuern (z.B. Speicher layout, Alignment, Inline-Assembler), ohne Sicherheitsgarantien zu opfern. Gleichzeitig ermöglichen Sprachfeatures wie Traits, Pattern Matching und Iteratoren hochabstrakte Konstrukte, die durch Monomorphisierung zur Compile-Zeit in effizienten Code übersetzt werden. Diese sogenannten zero-cost abstractions erzeugen keinen Overhead – was insbesondere für sicherheitskritische oder performance-sensitive Anwendungen relevant ist [85].

Typsicherheit und Fehlervermeidung zur Compile-Zeit Rust setzt auf ein starkes, statisches Typsystem mit expliziten Typannotationen, generischen Typen und

algebraischen Datentypen (enum, Option, Result). Dadurch werden viele Fehler – wie Null-Zugriffe, Typverwechslungen oder unbehandelte Rückgabewerte – bereits beim Kompilieren erkannt. Der Compiler prüft nicht nur Typverträglichkeit, sondern auch die Einhaltung von Lifetime-Regeln und Borrowing-Konflikten. Seit der Einführung Non Lexical Lifetimes (NLL) kann rustc auch kontextabhängige Lebensdauern dynamischer erkennen und erlaubt damit flexiblere, aber weiterhin sichere Referenznutzung [86]. Das Ergebnis ist robustere Software mit weniger Laufzeitfehlern.

Speicher- und Referenzmanagement

Das Ownership-Modell Rusts Ownership-Modell ist zentral für die speichersichere Programmierung ohne GC. Jeder Wert hat genau einen Besitzer. Wird ein Wert einer anderen Variablen zugewiesen oder an eine Funktion übergeben, geht das Eigentum über, und der ursprüngliche Besitzer verliert den Zugriff. Dies verhindert doppelte Freigaben und Dangling Pointers. Beim Verlassen des Gültigkeitsbereichs wird der Wert automatisch freigegeben, was durch das Drop-Trait ermöglicht wird. [87]

Mutable vs. Immutable Borrowing Neben dem Eigentum erlaubt Rust das Ausleihen von Werten durch Referenzen. Es gibt zwei Arten:

Immutable Borrowing `&T`: Erlaubt beliebig viele gleichzeitige, aber nur lesende Zugriffe.

Mutable Borrowing `&mut T`: Erlaubt genau einen schreibenden Zugriff, aber keine weiteren gleichzeitigen Referenzen.

Diese Regeln verhindern Datenrennen und garantieren, dass keine Referenz während einer Mutation auf veraltete Daten zeigt. Der Compiler erzwingt diese Regeln strikt zur Compile-Zeit.

Der Borrow Checker: Statische Analyse zur Compile-Zeit Der Borrow Checker ist ein zentrales Werkzeug des Rust-Compilers, das sicherstellt, dass die Regeln des Ownership- und Borrowing-Modells eingehalten werden. Er analysiert den Code während der Kompilierung und verhindert beispielsweise, dass ein Wert nach einem Move weiterhin verwendet wird oder dass gleichzeitig eine mutable und eine immutable Referenz existieren. Diese statische Analyse eliminiert eine Vielzahl von Speicherfehlern bereits vor der Ausführung des Programms. Medium

Lifetimes und Lebensdauern von Referenzen Lifetimes sind ein Mechanismus in Rust, um die Gültigkeitsdauer von Referenzen zu verfolgen und sicherzustellen, dass sie nicht auf ungültigen Speicher zeigen. Der Compiler kann in vielen Fällen die Lifetimes automatisch ableiten, aber in komplexeren Szenarien müssen sie explizit angegeben werden. Seit der Einführung der NLL ist der Compiler in der Lage, die Lebensdauer von Referenzen flexibler und präziser zu bestimmen, was die Programmierung erleichtert und die Sicherheit erhöht.

Webentwicklung mit actix-web

Actix-web ist eines der populärsten Web-Frameworks in Rust und bekannt für seine hohe Performance, geringe Latenz und speichersichere Architektur. Es basiert auf dem Actor-Modell (über die actix-Laufzeit) und bietet eine asynchrone, eventgesteuerte Serverstruktur. Das Framework nutzt Rusts async/await-Syntax vollständig aus und integriert sich nahtlos mit Tokio oder anderen async-Runtimes.

Die API von actix-web ist modular aufgebaut und erlaubt eine saubere Trennung von Routen, Middleware und Handlern. Dank des Typsystems können viele typische Webfehler (z.B. ungültige Parameter) bereits zur Compile-Zeit verhindert werden. actix-web unterstützt REST, WebSockets, Transport Layer Security (TLS), Body-Parsing, Cross-Origin Resource Sharing (CORS) und viele weitere Funktionen „out of the box“.

4.6.4 Vergleich: Rust vs. Go

Garbage Collection vs. Ownership

Go verwendet eine automatische GC, die Speicher zur Laufzeit freigibt, sobald Objekte nicht mehr referenziert werden. Dies vereinfacht die Entwicklung, kann jedoch zu unvorhersehbaren Pausen führen, insbesondere bei speicherintensiven Anwendungen. [88]

Rust hingegen verzichtet vollständig auf eine GC. Stattdessen setzt es auf ein Ownership-Modell mit statischer Analyse zur Compile-Zeit. Jeder Wert hat einen eindeutigen Besitzer, und der Compiler stellt sicher, dass keine ungültigen Referenzen oder Datenrennen entstehen. Dies ermöglicht eine deterministische Speicherfreigabe ohne Laufzeit-Overhead. [89]

Unterschiede in Sicherheit, Performance und Parallelismus

Sicherheit: Rust bietet durch sein Ownership- und Borrowing-System eine hohe Speichersicherheit. Der Compiler verhindert viele Fehler bereits zur Compile-Zeit, wie

z.B. Null-Pointer-Dereferenzen oder Datenrennen. Go bietet ebenfalls Mechanismen zur Speichersicherheit, jedoch werden bestimmte Fehler erst zur Laufzeit erkannt.

Performance: Rust erzielt in Benchmarks häufig bessere Laufzeiten und geringeren Speicherverbrauch als Go, insbesondere bei CPU-intensiven Aufgaben. [90] Dies liegt an der fehlenden GC und den Zero-Cost-Abstraktionen von Rust.[91]

Parallelismus: Go erleichtert die nebenläufige Programmierung durch Goroutines und Channels, was eine einfache Handhabung von Parallelität ermöglicht. [92] Allerdings liegt die Verantwortung für die Vermeidung von Datenrennen beim Entwickler. Rust hingegen erzwingt durch sein Typsystem und den Borrow Checker sichere Parallelität, indem es Datenrennen bereits zur Compile-Zeit verhindert. [93]

4.7 Datenbankentscheidungen

Die Wahl der Datenbank stellt eine essenzielle Entscheidung für das Projekt dar und ist wegweisend dafür, wie Daten gespeichert und verarbeitet werden können. Um die speziellen Anforderungen an Sensors zu erfüllen, beleuchten wir in diesem Kapitel die verschiedenen Datenbankmodelle und kommen abschließend zu einer fundierten Entscheidung.

Grundlegend lassen sich Datenbanken in zwei Hauptkategorien einteilen: SQL-Datenbanken und NoSQL-Datenbanken.

SQL-Datenbanken, auch als relationale Datenbanken bezeichnet, basieren auf dem relationalen Datenmodell. Diese Datenbanken sind tabellarisch strukturiert, wobei jede Zeile einen Datensatz bildet und jede Spalte ein spezifisches Feld innerhalb dieses Datensatzes repräsentiert. SQL-Datenbanken sind bekannt für ihre ACID-Eigenschaften (Atomicity, Consistency, Isolation, Durability), die gewährleisten, dass Transaktionen zuverlässig und sicher ausgeführt werden. Diese Eigenschaften sind besonders wichtig für Anwendungen, bei denen Datenintegrität und Transaktionssicherheit von höchster Bedeutung sind. SQL-Datenbanken erlauben zudem das Setzen von Beziehungen und Verknüpfungen zwischen Datensätzen, was durch integrierte Kontrollmechanismen unterstützt wird. Die Abfrage-sprache, die in diesen Systemen verwendet wird, ist Structured Query Language (SQL). Beispiele für SQL-Datenbanken sind MySQL, PostgreSQL, Oracle Database und Microsoft SQL Server.

NoSQL-Datenbanken hingegen bieten eine breite Palette von Datenbankmodellen, die nicht auf das relationale Modell beschränkt sind. Sie wurden entwickelt, um einige der Einschränkungen von SQL-Datenbanken zu überwinden und bieten mehr Flexibilität für unterschiedliche Arten von Daten und Anwendungsfällen. Im Gegensatz zu SQL-Datenbanken bieten NoSQL-Datenbanken ein flexibles Datenschema und unterstützen

zen verschiedene Datenmodelle, darunter dokumentenorientierte Modelle, Schlüssel-Wert-Modelle, spaltenorientierte Modelle und graphenbasierte Modelle. Ein weiteres Unterscheidungsmerkmal ist das Prinzip der „eventual consistency“. Dies bedeutet, dass Datenänderungen über die Zeit hinweg konsistent werden, was in verteilten Systemen die hohe Verfügbarkeit gewährleistet, jedoch auch Risiken in Bezug auf Konsistenz und Datenintegrität mit sich bringen kann. Beispiele für NoSQL-Datenbanken sind MongoDB, Cassandra, Couchbase, Redis und Neo4j. [94]

Aufgrund der Expertise im Entwicklerteam sind MongoDB als objektorientierte Datenbank und PostgreSQL als relationale Datenbank die möglichen Datenbanklösungen für Sensorsa. Eine Gegenüberstellung beider Technologien führt zur fundierten Auswahl.

4.7.1 Exkurs: PostgreSQL

Um eine fundierte Entscheidung treffen zu können, ist es notwendig, die Architektur und die technischen Hintergründe der PostgreSQL-Datenbank genauer zu betrachten. PostgreSQL, als objektrelationale Datenbank, bietet eine robuste und skalierbare Grundlage für die Speicherung und Verarbeitung von Daten. Im Folgenden wird der Aufbau von PostgreSQL detailliert beschrieben.

Cluster-Struktur und Prozesse

Ein PostgreSQL-Server besteht aus einem sogenannten Cluster, der mehrere Datenbanken umfasst. Dieser Cluster kann als die übergeordnete Instanz betrachtet werden, in der alle Datenbanken organisiert sind. Auf einem Server läuft also ein PostgreSQL-Cluster, der die Verwaltung mehrerer Datenbanken übernimmt. Diese Struktur ermöglicht eine effiziente Ressourcennutzung und einfache Skalierung.

PostgreSQL besteht aus einer Kombination von Speicher und Prozessen. Bei Unix-Systemen sind diese Prozesse eigenständig, während sie bei Windows als Threads umgesetzt werden. Die wichtigsten Prozesse, die die Funktionalität von PostgreSQL abbilden, sind:

Postmaster: Der Hauptprozess, der die Verwaltung der anderen Prozesse übernimmt.

Checkpointer: Dieser Prozess sorgt dafür, dass alle Änderungen in der Datenbank regelmäßig auf die Festplatte geschrieben werden.

Writer: Verantwortlich für das Schreiben von geänderten Datenblöcken auf die Festplatte.

WAL Writer: Handhabt das Schreiben von Transaktionsdaten in die Write-Ahead Log (WAL).

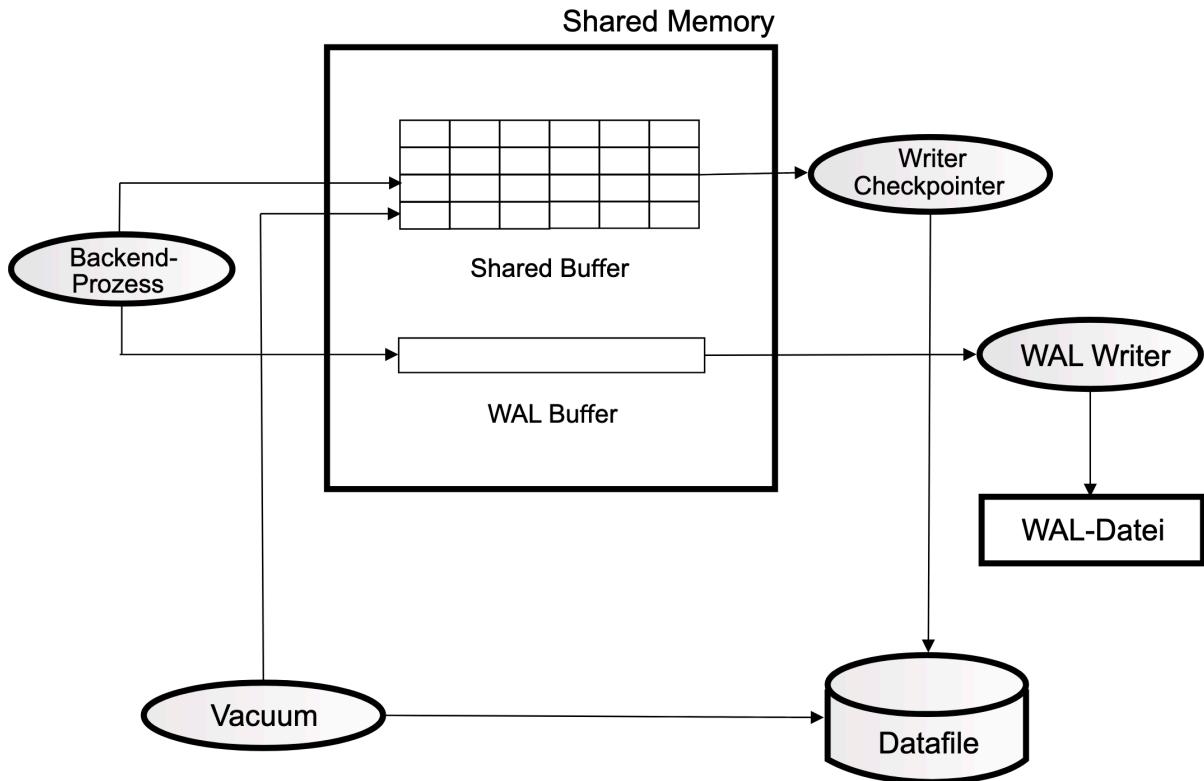


Abbildung 4.1: PostgreSQL Architektur. Quelle: [95] S. 33 Bild 4.1

Autovacuum Launcher: Dieser Prozess führt automatische Bereinigungen der Datenbank durch.

Archiver: Archiviert die WAL-Dateien für eine verbesserte Rückverfolgbarkeit und Monitoring, besonders wichtig in produktiven Systemen.

Stats Collector: Sammelt statistische Daten über die Nutzung von Sessions und Tabellen.

BGWorker: Übernimmt verschiedene Hintergrundaufgaben.

Beim Aufbau einer Verbindung durch einen Client erstellt der Postmaster-Prozess nach erfolgreicher Authentifizierung und Autorisierung einen eigenen Backend-Thread. Die maximale Anzahl gleichzeitig aktiver Backend-Threads ist limitiert (Standard 100), was bedeutet, dass nach Erreichen dieser Grenze weitere Verbindungen abgelehnt werden.

Speicherverwaltung und Performanceoptimierung

Die Verwaltung von Speicher und die Optimierung der Zugriffszeiten auf Daten sind entscheidend für die Leistung von PostgreSQL. Da das Lesen von Daten von einer Festplatte

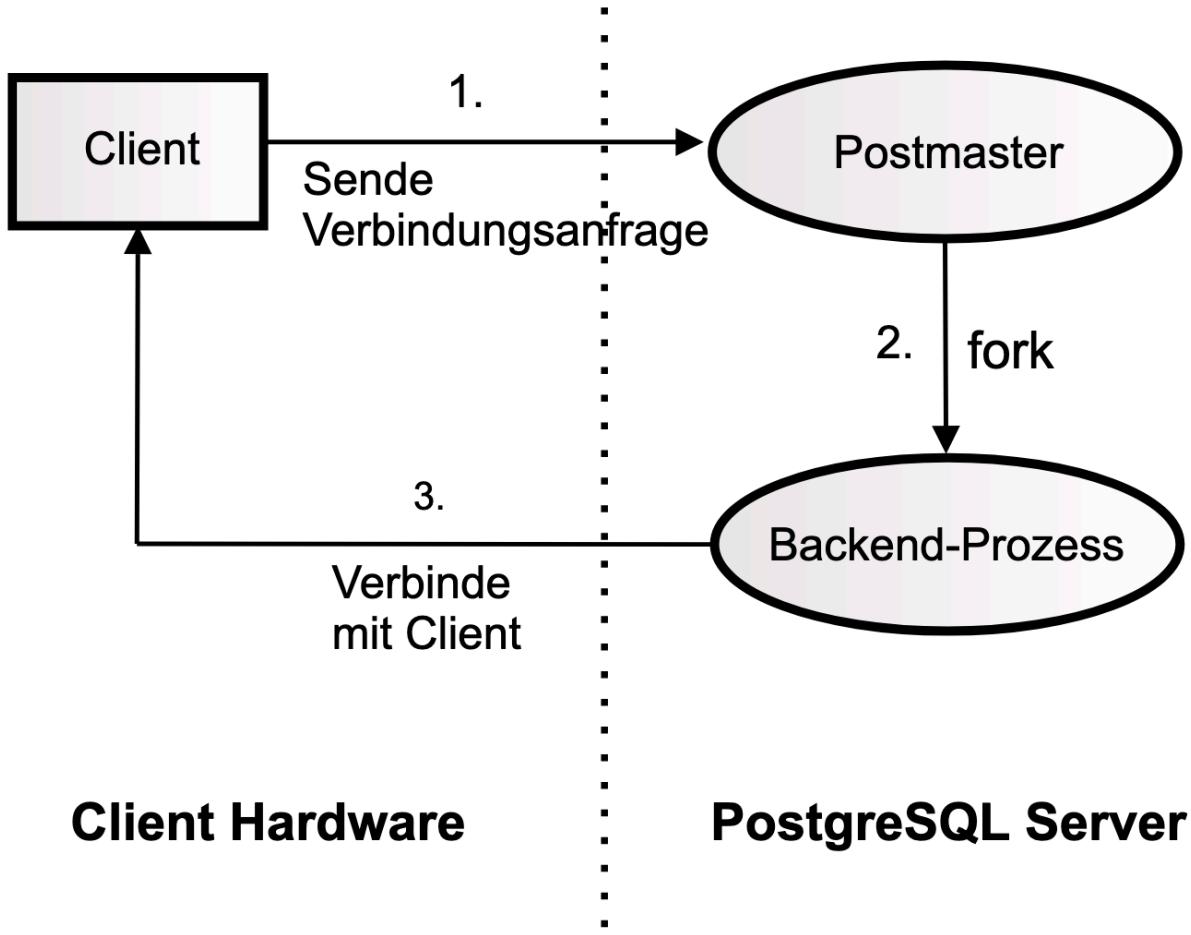


Abbildung 4.2: Verbindungsauftakt von Client zum Server. Quelle: [95] S. 35 Bild 4.3

vergleichsweise zeitaufwendig ist, setzt PostgreSQL auf verschiedene Mechanismen zur Verbesserung der Performance.

Geteilter Speicher (Shared Buffer) Der Shared Buffer in PostgreSQL dient dazu, die Anzahl der Operationen auf der Festplatte zu reduzieren, indem er schnellen Speicher im RAM bereitstellt. Er besteht aus einer Hash-Tabelle, Hash-Elementen, einer Bufferbeschreibung und einer Buffersammlung. Die Hash-Tabelle ermöglicht ein schnelles Auffinden von Datensätzen, indem Hashwerte in Segmenten sortiert werden. Diese Segmente können sehr schnell durchsucht werden, da ihre Länge fest ist, was effiziente Speicherzugriffe ermöglicht.

Die Bufferbeschreibung enthält wichtige Steuerungsvariablen, wie z.B. den `usagecount`, der die Relevanz eines Datensatzes im Speicher angibt. Der Speicherbereinigungsprozess (Eviction Process) durchsucht regelmäßig den Buffer nach irrelevanten Blöcken. Blöcke, die als irrelevant eingestuft werden, können überschrieben werden, sobald neuer Speicher-

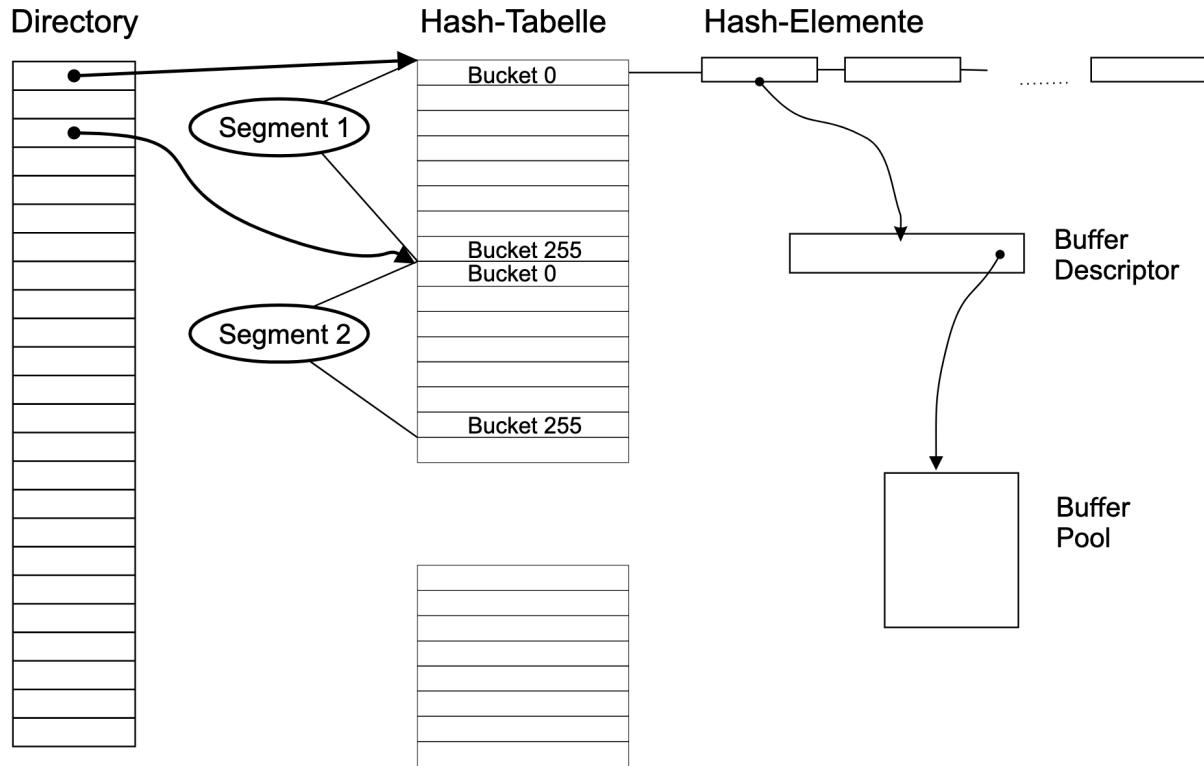


Abbildung 4.3: PostgreSQL Speicherarchitektur. Quelle: [95] S. 37 Bild 4.4

platz benötigt wird.

WAL Buffer und Checkpoints Der Write-Ahead Log (WAL) Buffer ist darauf ausgelegt, Daten effizient auf die Festplatte zu schreiben. Dadurch müssen schreibende Prozesse nicht auf die tatsächliche Schreiboperation warten, sondern können die zu schreibenden Daten in den Buffer verlagern. Der WAL Buffer wird in regelmäßigen Intervallen (Standard 200 Millisekunden) überprüft und die Daten werden im Permanentspeicher gesichert, um die Ausfallsicherheit zu gewährleisten.

Insert-, **Update**- und **Delete**-Anweisungen werden über den WAL-Buffer verarbeitet. Diese Anweisungen werden in sogenannten WAL-Records gespeichert und sichern den Zustand der Daten auch bei einem Systemausfall. Ein Checkpoint, der den Speicherinhalt endgültig auf die Festplatte schreibt, wird unter bestimmten Bedingungen ausgeführt, wie etwa nach Ablauf eines festgelegten Intervalls, Erreichen einer bestimmten Buffergröße oder dem manuellen Auslösen eines Checkpoints.

Mehrere Datenblockversionen und Defragmentierung PostgreSQL unterstützt die gleichzeitige Bearbeitung von Datensätzen durch mehrere Sessions. Um sicherzustellen, dass sich die Sessions nicht gegenseitig beeinträchtigen, erhalten Datensätze eine Versions-

nummer. Diese ermöglicht es, bei gleichzeitigen Abfragen und Bearbeitungen konsistente Daten bereitzustellen. Wenn eine Abfrage gestartet wird, kann anhand der Versionsnummer bestimmt werden, welcher Datensatz zum Zeitpunkt des Abfragestarts aktuell war.

Alte Datensätze, die nicht mehr benötigt werden, werden gelöscht, wodurch Lücken in der Tabelle entstehen. Diese Lücken werden durch den Autovacuum-Prozess freigegeben, sodass der Speicher für neue Datenblöcke verwendet werden kann. Bei Bedarf führt PostgreSQL ein **fullvacuum** durch, der die Tabelle vollständig defragmentiert, um Speicherplatz zurückzugewinnen und die Integrität der Versionsnummern zu gewährleisten.

Permanentspeicherverwaltung

Die Datenbanken von PostgreSQL werden im Verzeichnis **base** gespeichert, während der Tablespace in einem separaten Verzeichnis liegt. Tabellen und Indizes werden in Dateien von maximal einem Gigabyte Größe gespeichert; bei Überschreiten dieser Größe wird eine weitere Datei angelegt. Zusätzlich werden Dateien mit den Endungen **_fsm** (Free Space Map) zur Adressierung des freien Speicherplatzes und **_vm** (Visibility Map) zur Markierung von veralteten Datensätzen gepflegt.

Die Datenblocksammlung (Page) enthält Pointers auf die tatsächlichen Datenblöcke (Tuples). Jeder Datenblock wird durch eine Kombination aus der Datenblocksammlungsnummer und dem Offset des Pointers identifiziert. [95]

4.7.2 Exkurs: MongoDB

MongoDB ist eine führende dokumentenorientierte NoSQL-Datenbank, die sich durch ihre Flexibilität und Skalierbarkeit auszeichnet. Sie wurde entwickelt, um einige der Einschränkungen traditioneller relationaler Datenbanken zu überwinden und bietet eine leistungsfähige Lösung für Anwendungen, die große Mengen an unstrukturierten oder semi-strukturierten Daten verarbeiten müssen. Im Gegensatz zu relationalen Datenbanken, die auf einem festen Schema basieren, erlaubt MongoDB ein dynamisches, schemaloses Datenmodell, das sich ideal für moderne, agile Entwicklungsprozesse eignet. In diesem Exkurs werden die Architektur und die technischen Merkmale von MongoDB detailliert beleuchtet, um deren Eignung für das Notification and Message Tracking Integration Service (NOMTIS)-Projekt zu bewerten.

Speicherstruktur und Organisation

MongoDB speichert Daten im Binary JSON (BSON)-Format, einem binären Format, das speziell für die Speicherung von Dokumenten mit JSON-Syntax entwickelt wurde. BSON bietet eine effiziente Speicherung und Verarbeitung von Dokumenten, da es im Vergleich

zu herkömmlichem JSON kompakter und schneller zu verarbeiten ist. BSON-Dokumente können direkt in JSON konvertiert werden, was die Integration mit Anwendungen erleichtert, die JSON verwenden.

Die Speicherorganisation in MongoDB ist hierarchisch aufgebaut:

Datenbank: An der Spitze steht eine Datenbank, die mehrere Sammlungen (engl. Collections) enthalten kann.

Katalog: Darunter befindet sich der Katalog, der Informationen über die Sammlungen und Indizes speichert.

Sammlung (engl. Collection) : Eine Sammlung ist eine Gruppe von Dokumenten, die thematisch zusammengehören und ähnliche Strukturmerkmale aufweisen.

Dokument: Auf der untersten Ebene steht das Dokument, das die eigentlichen Daten enthält.

Katalog und Speicherverwaltung

Der Katalog von MongoDB ist für die Verwaltung der Metadaten von Sammlungen und Indizes verantwortlich. Es gibt zwei Arten von Katalogen:

Persistenter Katalog: Dieser speichert Informationen dauerhaft auf der Festplatte und stellt sicher, dass Metadaten auch nach einem Neustart des Systems verfügbar bleiben. Der persistente Katalog wird in BSON-Dateien mit der Endung `_mdb_catalog` gespeichert und enthält Details über die Eigenschaften und Indizes der Sammlungen.

Memory-Katalog: Der Memory-Katalog hält die Metadaten im RAM vor, um schnellen Zugriff zu gewährleisten und Ladezeiten zu minimieren. Operationen wie das Erstellen, Suchen, Iterieren und Schließen von Sammlungen werden im Memory-Katalog durchgeführt.

Um Dateninkonsistenzen zu vermeiden, wird der Memory-Katalog regelmäßig mit dem persistenten Katalog synchronisiert. Eine Versionsverwaltung stellt sicher, dass Änderungen korrekt nachvollzogen werden können, ohne dass es zu Lese-Schreib-Kollisionen kommt. MongoDB verwendet dabei das Copy-on-Write-Prinzip, bei dem nur dann eine Kopie des Datensatzes erstellt wird, wenn tatsächlich Änderungen vorgenommen werden. Dies minimiert den Speicherbedarf und erhöht die Effizienz.

Speicherverwaltung und Defragmentierung

Beim Löschen eines Eintrags in MongoDB erfolgt dieser Prozess in zwei Schritten:

1. Löschen des Katalogeintrags: Zunächst wird der Eintrag sowohl im Memory-Katalog als auch im persistenten Katalog entfernt. Ein Verweis auf die Sammlung wird an den sogenannten „Reaper“ übergeben, um sicherzustellen, dass keine neuen Zugriffe auf die Sammlung erfolgen, während aktive Lesezugriffe abgeschlossen werden.
2. Endgültiges Löschen: Nachdem sichergestellt wurde, dass kein Rollback erfolgen kann, der die Daten der Sammlung wiederherstellen könnte, werden die Daten endgültig gelöscht.

Freier Speicher, der durch das Löschen von Daten entsteht, wird in regelmäßigen Abständen für neue Datensätze freigegeben. Wenn der Speicher fragmentiert wird und viele kleine Lücken entstehen, führt MongoDB eine Defragmentierung durch. Dieser Prozess ordnet den Speicher neu an, kann jedoch Lese- und Schreiboperationen blockieren, was die Performance beeinträchtigen könnte. Daher sollte die Defragmentierung möglichst selten durchgeführt werden.

Indizes und Abfragen

Indizes in MongoDB werden hauptsächlich als B-Baum-Strukturen gespeichert, die eine effiziente Suche und Sortierung von Daten ermöglichen. Ein B-Baum-Index speichert die Daten strukturiert, ähnlich wie ein Dokument, und ermöglicht schnelle Zugriffe auf häufig abgefragte Felder. MongoDB erlaubt es auch, benutzerdefinierte Speicherstrukturen für Indizes zu erstellen, um spezifische Anwendungsanforderungen zu erfüllen. [96] [97]

BSON - Binary JSON

BSON ist das zugrunde liegende Format für die Speicherung von Dokumenten in MongoDB. Es handelt sich dabei um ein binäres Format, das sogenannte Key-Value-Paare speichert und direkt in JSON konvertiert werden kann. BSON ist jedoch deterministischer als JSON, da es keine Flexibilität in der Darstellungsform bietet. Dies bedeutet, dass ein Datensatz in BSON immer in einer einzigen, festen Form dargestellt wird.

Ein BSON-Dokument beginnt immer mit einem 32-Bit-Ganzzahlwert, der die Länge des Dokuments angibt, und endet mit einem Null-Byte. Aufgrund dieser Struktur kann ein BSON-Dokument maximal ca. 2,147 Gigabyte an Daten umfassen. Die verschiedenen Datentypen, die BSON unterstützt, umfassen unter anderem:

Int32 und Int64: 4- bzw. 8-Byte lange Ganzzahlen.

Double: 8-Byte lange Gleitkommazahlen nach dem IEEE 754-Standard.

Decimal128: 16-Byte lange Gleitkommazahlen für hohe Präzision.

Array: Ein Feld, das eine Liste von Werten speichert, wobei die Elemente wie Dokumente behandelt werden.

Null-Wert: Ein spezieller Marker, der das Fehlen eines Wertes kennzeichnet.

Min- und Max-Schlüssel: Marker, die den minimalen bzw. maximalen Wert eines Feldes darstellen.

Die strikte Struktur von BSON sorgt für eine konsistente und effiziente Speicherung und Verarbeitung von Dokumenten, was MongoDB zu einer leistungsfähigen Lösung für Anwendungen macht, die flexible und skalierbare Datenbanken benötigen. [98]

4.7.3 Diskurs: Vergleich von PostgreSQL und MongoDB für NOMTIS

Die Auswahl der geeigneten Datenbanklösung ist ein zentraler Aspekt bei der Entwicklung von NOMTIS. Da die internen Richtlinien PostgreSQL und MongoDB als verfügbare Optionen vorsehen, ist es entscheidend, die spezifischen Anforderungen von NOMTIS zu analysieren und zu bestimmen, welche der beiden Datenbanken diese am besten erfüllt.

Technische Anforderungen an NOMTIS

NOMTIS muss Benachrichtigungen effizient speichern, verwalten und flexibel verarbeiten können. Ein Hauptmerkmal von NOMTIS ist die Fähigkeit, Benachrichtigungen mit potenziell komplexen und verschachtelten Datenstrukturen zu verwalten. Diese Datenstrukturen können variieren und sind oft nicht im Voraus vollständig definiert. Zudem sind eine hohe Zuverlässigkeit, Konsistenz und Skalierbarkeit erforderlich, um den Anforderungen eines modernen Benachrichtigungssystems gerecht zu werden.

PostgreSQL: Technische Stärken und Grenzen

PostgreSQL ist eine relationale Datenbank, die für ihre starke Unterstützung von ACID-Transaktionen, Datenintegrität und komplexen relationalen Abfragen bekannt ist. Diese Eigenschaften machen PostgreSQL ideal für Anwendungen, bei denen strenge Konsistenzanforderungen und komplexe Datenbeziehungen im Vordergrund stehen.

Die feste Struktur von PostgreSQL kann jedoch zu Einschränkungen führen, wenn es darum geht, dynamische und verschachtelte Datenstrukturen zu verwalten. Änderungen

am Datenmodell erfordern eine Anpassung des Schemas, was in einem dynamischen Umfeld, wie es NOMTIS erfordert, potenziell problematisch sein kann. Zudem ist PostgreSQL weniger effizient im Umgang mit tief verschachtelten oder unvorhersehbaren Datenstrukturen, was die Flexibilität des Systems einschränken könnte.

MongoDB: Flexibilität und Skalierbarkeit

MongoDB hingegen ist eine dokumentenorientierte NoSQL-Datenbank, die sich durch ihre schemalose Architektur und hohe Flexibilität auszeichnet. Die Speicherung erfolgt im BSON-Format, das eine effiziente Handhabung von JSON-ähnlichen Dokumenten ermöglicht. Diese Struktur erlaubt es, Daten ohne festes Schema zu speichern, was für NOMTIS von Vorteil ist, da die Benachrichtigungen beliebig komplexe und verschachtelte Daten enthalten können.

Ein weiterer zentraler Vorteil von MongoDB ist seine horizontale Skalierbarkeit. MongoDB unterstützt das sogenannte „Sharding“, bei dem Daten über mehrere Server verteilt werden, um eine hohe Verfügbarkeit und Performance sicherzustellen. Diese Fähigkeit zur horizontalen Skalierung ist besonders in einer Umgebung wie der Schwarz IT KG (SIT) von Bedeutung, wo das Datenvolumen und die Anzahl der Nutzer kontinuierlich wachsen können.

Laut einer Untersuchung der MongoDB-Datenbank von Anjali Chauhan bietet MongoDB darüber hinaus eine hohe Leistung durch die Verwendung eingebetteter Datenmodelle, die E/A-Aktivitäten reduzieren, sowie durch Indizes, die schnellere Abfragen ermöglichen. [99] Diese Leistungsmerkmale machen MongoDB besonders geeignet für Anwendungen, die auf hohe Geschwindigkeit und effizienten Datenzugriff angewiesen sind.

Schlussfolgerung: PostgreSQL als geeignete Wahl für Sensora

Im Projekt Sensora ergibt sich aus den strukturellen Anforderungen und den funktionalen Zugriffsmustern ein klarer Bedarf an einem relationalen, transaktional konsistenten Datenbanksystem. Die Datenstruktur ist vordefiniert und weitgehend stabil. Pflanzen, Sensoren, Aktoren, Gruppen und Räume bilden klar abgegrenzte Entitäten mit definierten Beziehungen zueinander. Diese Relationen sind nicht nur logisch konzipiert, sondern stellen funktionale Notwendigkeiten dar – etwa wenn Sensoren bestimmten Pflanzen zugeordnet sind oder Aktionen nur innerhalb definierter Gruppenzugehörigkeiten zulässig sind.

Die Abfrageanforderungen im operativen Betrieb bestehen aus einer Mischung aus punktuellen Zugriffen (etwa auf aktuelle Sensorwerte oder Soll-Ist-Vergleiche) und zeitbasierten Auswertungen über große Datenmengen – insbesondere für die letzten 24 Stunden. Diese Anforderungen sind auf ein konsistentes, indexoptimiertes Schema angewiesen,

das Joins effizient unterstützt und sich nicht durch Schema-Flexibilität, sondern durch strukturelle Integrität auszeichnet. Gleichzeitig ist paralleler Zugriff durch mehrere Microservices erforderlich, sodass Transaktionssicherheit und Isolation nicht nur erwünscht, sondern notwendig sind, um Inkonsistenzen bei konkurrierenden Schreiboperationen zu vermeiden.

PostgreSQL bietet genau für diese Art von Workload die passende Grundlage. Die stark normierten Strukturen des *Sensora*-Datenmodells profitieren von PostgreSQLs ausgereifter Optimierung relationaler Abfragen, der zuverlässigen Durchsetzung referenzieller Integrität sowie der Möglichkeit, durch gezieltes Indexing auf Zeitstempel-Feldern hochfrequente historische Abfragen performant abzubilden. Selbst bei wachsendem Datenvolumen lassen sich durch Partitionierung oder die spätere Ergänzung durch TimescaleDB – ohne Verlassen der PostgreSQL-Basis – die Performanceanforderungen langfristig erfüllen, ohne strukturelle Kompromisse einzugehen.

Die fehlende Notwendigkeit für dynamische Schemas, polymorphe Dokumente oder eingebettete Strukturen eliminiert die Hauptargumente für ein dokumentenbasiertes System wie MongoDB. Vielmehr würde dessen Flexibilität in diesem Kontext eher potenzielle Inkonsistenzen begünstigen und zusätzliche Validierungslogik auf Anwendungsebene erforderlich machen – ein Mehraufwand, der durch das klar strukturierte Datenmodell nicht gerechtfertigt ist.

In Summe ist PostgreSQL damit nicht nur die technisch bessere Wahl, sondern die natürlichere Fortsetzung der bereits im Projektdesign angelegten Prinzipien: strukturierte, konsistente und integrierte Datenhaltung, auf die performant und sicher gleichzeitig von vielen Systemkomponenten zugegriffen werden kann.

4.8 Auswahl der Technologien für die entwickelten Schnittstellen-Services

Die im Rahmen des Projekts *Sensora* entwickelten Schnittstellen-Services übernehmen unterschiedliche Aufgaben innerhalb der Systemarchitektur, stellen jedoch alle zentrale Bausteine zur Kommunikations- und Steuerungsebene dar. Ihre Implementierung erfordert eine wohlüberlegte Auswahl geeigneter Technologien, die sowohl den funktionalen Anforderungen als auch den architekturellen, sicherheitstechnischen und betrieblichen Rahmenbedingungen gerecht werden.

In diesem Kapitel werden die getroffenen Technologieentscheidungen für jeden entwickelten Service einzeln erläutert, mit vergleichbaren Alternativen kontextualisiert und unter Rückbezug auf die definierten Anforderungen begründet.

4.8.1 Technologieauswahl für den Authentifizierungs- und Registrierungsdienst (Auth-Service)

Zielsetzung und Kontext

Der Auth-Service ist eine der sicherheitskritischsten Komponenten des Sensora-Systems. Er ist dafür verantwortlich, neue IoT-Controller kontrolliert ins System aufzunehmen, ihnen eindeutige Kommunikationsidentitäten zuzuweisen und die Authentizität dieser Geräte eindeutig zu überprüfen. Zudem verwaltet er die Registrierung von Controllern in der zentralen Datenbank und erzeugt differenzierte Zugriffskanäle innerhalb der Messaging-Infrastruktur.

Verwendete Programmiersprache: Python

Für die Umsetzung des Auth-Service wurde die Programmiersprache **Python** gewählt. Python bietet eine sehr hohe Ausdrucksstärke bei gleichzeitig niedriger Komplexität in der Syntax[100]. Dies erlaubt eine fokussierte Umsetzung sicherheitskritischer Logik mit hoher Lesbarkeit und reduziertem Fehlerpotenzial. Die Sprache bietet native Unterstützung für REST-APIs (via Flask), HMAC-Berechnungen (via `hashlib`) und JSON-Verarbeitung, was sie ideal für die Umsetzung des Auth-Service macht.

Im Vergleich zu Alternativen wie Java oder Go zeigt Python zwar leistungstechnische Schwächen bei hochfrequenten Systemen, bietet dafür jedoch wesentlich höhere Entwicklungsproduktivität – ein entscheidender Vorteil im Rahmen eines begrenzten studentischen Projektzeitraums.

Alternative Bewertung: Java bietet mit Spring Security zwar eine äußerst robuste Infrastruktur für Authentifizierungsmechanismen, ist jedoch deutlich komplexer im Deployment und benötigt mehr Konfigurationsaufwand. Go bietet starke Performance und native Concurrency-Modelle, jedoch eine im Vergleich zu Python eingeschränkte Bibliothekslandschaft für Security-Workflows.

Begründung der Auswahl: Angesichts der Priorisierung von Entwicklungszeit, Lesbarkeit, Testbarkeit und Verfügbarkeit passender Security-Tools wurde Python als die geeignete Sprache für den Auth-Service identifiziert.

Authentifizierungsmechanismus: HMAC-basiertes Challenge-Response-Verfahren

Für die Authentifizierung der IoT-Geräte wurde ein leichtgewichtiges Challenge-Response-Verfahren auf Basis von HMAC (Hash-based Message Authentication Code) implementiert.

tiert. Diese Entscheidung basiert auf den Anforderungen an ein sicheres, serverseitig validierbares Verfahren ohne Notwendigkeit, Klartext-Zugangsdaten zu speichern oder zu übertragen.

Alternative Technologien:

- **OAuth 2.0:** Standard für Benutzer-Authentifizierung, jedoch komplex in der Implementierung für Maschinen-zu-Maschinen-Kommunikation.
- **JWT (JSON Web Token):** Geeignet für Token-basierte Sessions, jedoch problematisch hinsichtlich Zustandslosigkeit bei Geräten mit hohem Sicherheitsanspruch.
- **Client-Zertifikate:** Sehr sicher, aber schwergewichtig in der Verwaltung für dynamisch zu registrierende IoT-Geräte.

Begründung der Auswahl: Das HMAC-Verfahren ermöglicht die Validierung eines vorab generierten Geräteschlüssels (Token), ohne diesen selbst senden zu müssen. Durch serverseitige Generierung einer Challenge, die vom Gerät korrekt beantwortet werden muss, wird ein sicheres, manipulationsresistentes Protokoll etabliert, das gleichzeitig ressourcenschonend und einfach zu implementieren ist. Die Entscheidung orientiert sich damit an Best Practices für Geräteauthentifizierung in Embedded-Umgebungen [101].

Kommunikationsmodell: REST-basierte API

Der Service stellt seine Funktionalität über HTTP/REST-Endpunkte zur Verfügung. Die Entscheidung für ein REST-Modell basiert auf der Notwendigkeit, den Dienst sowohl von Web-Frontends als auch von anderen Services (z. B. Mail-Service oder Solace-Konfiguration) ansprechbar zu machen.

Alternative Technologien:

- **gRPC:** Hohe Effizienz, jedoch schwerer in Browser-Umgebungen integrierbar.
- **MQTT:** Bietet nur Publish/Subscribe, nicht geeignet für Request/Response-Workflows mit stateless APIs.

Begründung: REST bietet mit seiner stateless Architektur und breiten Toolunterstützung (z. B. Swagger, Postman, curl) die optimale Basis für verteilte Systeme, insbesondere für administrative Operationen wie die Controller-Registrierung.

Interner Konfigurationsspeicher: JSON-Datei

Die Informationen über aktive Challenges und bereits registrierte Geräte werden zusätzlich zur Datenbank in einer strukturierten JSON-Datei gespeichert. Dies ermöglicht schnelle Zugriffe auf temporäre Daten ohne Overhead einer persistierten Transaktion. Dieser pragmatische Kompromiss ist im Kontext studentischer Prototypen vertretbar.

4.8.2 Technologieauswahl für den E-Mail-Verifikationsdienst (Mail-Service)

Zielsetzung und Kontext

Der Mail-Service stellt eine sicherheitsrelevante Verbindung zwischen Benutzerschnittstellen und Backend dar. Seine primäre Aufgabe ist die Versendung von E-Mail-Verifizierungslinks nach der Benutzerregistrierung. Damit fungiert er als zentrale Instanz zur initialen Verifikation von Benutzeridentitäten. Neben der Kommunikation mit einem SMTP-Server stellt der Dienst auch eine kontrollierte API zur Entgegennahme von Verifizierungsanfragen bereit.

Verwendete Programmiersprache: Python

Die Entscheidung für Python basiert auf ähnlichen Argumenten wie beim Auth-Service. Python bietet durch Bibliotheken wie `smtplib` und `email.mime` einfache Schnittstellen zur Realisierung von SMTP-Kommunikation. Zusätzlich erlaubt die Verwendung von Flask eine unkomplizierte REST-Anbindung mit geringen Komplexitätshürden. Im Rahmen des Projekts ermöglichte dies eine schnelle, wartbare und lesbare Implementierung des Dienstes.

Vergleich mit Alternativen: Java (z. B. mit Spring Boot Mail) hätte eine robustere Infrastruktur geboten, wäre jedoch mit erheblich höherem Konfigurationsaufwand verbunden gewesen. Node.js wiederum bietet durch Pakete wie `nodemailer` eine gute Grundlage, ist jedoch im Team hinsichtlich Erfahrung weniger etabliert gewesen.

Begründung der Auswahl: Im Hinblick auf Entwicklungszeit, Lesbarkeit, Bibliotheksunterstützung und Teamkompetenz wurde Python als pragmatische und effektive Lösung gewählt.

E-Mail-Kommunikationsprotokoll: SMTP über TLS

Für den Versand von Verifizierungsnachrichten wurde das Simple Mail Transfer Protocol (SMTP) in Kombination mit einer Transport Layer Security (TLS)-Verbindung eingesetzt. Diese Kombination bietet einen etablierten Standard für ausgehende Mail-Kommunikation mit Basisverschlüsselung.[102]

Alternativen:

- **REST-basierte Mailservices (z. B. SendGrid, Mailgun):** Bieten einfache APIs und statistische Auswertung, erfordern aber Drittanbieterkonten und externe Infrastruktur.
- **SMTP ohne Verschlüsselung:** Unsicher und nicht datenschutzkonform.

Begründung: Die Entscheidung für SMTP über TLS wurde getroffen, da ein Gmail für eine nicht zu Hohe Menge an Mails dies kostenlos ermöglicht und bereits zur Verfügung stand und keine Drittanbieterintegration gewünscht war. Gleichzeitig konnten Sicherheitsanforderungen gewahrt bleiben.

Zugriffsschutz: Pre-Shared Key (PSK)

Um zu verhindern, dass Dritte beliebige Verifizierungsanfragen senden, wurde der REST-Endpunkt des Mail-Service mit einem Pre-Shared Key abgesichert. Nur Systeme, die diesen kennen, können autorisiert E-Mails auslösen.

Alternative Schutzmechanismen:

- **OAuth 2.0 oder API Tokens:** Sicher, aber unnötig komplex für einen geschlossenen Service.
- **Keine Authentifizierung:** Sicherheitsrisiko.

Begründung: Der Einsatz eines PSK ist ein einfacher, aber effektiver Schutzmechanismus, der in einem geschlossenen Systemumfeld wie Sensora praktikabel und ausreichend sicher ist.

4.8.3 Technologieauswahl für den Datenpersistenzdienst (Database Writer)

Zielsetzung und Kontext

Der Database Writer nimmt eine zentrale Rolle in der persistenznahen Verarbeitung eingehender Sensordaten ein. Seine Hauptaufgabe besteht darin, kontinuierlich Datenpakete aus der Messaging-Infrastruktur entgegenzunehmen, diese auszuwerten und in strukturierter Form in das zugrunde liegende Datenbanksystem zu übertragen. Eine besondere Herausforderung besteht dabei in der Notwendigkeit, sowohl hohe Verfügbarkeit als auch Integrität der Messdaten zu gewährleisten, selbst bei temporären Netzwerkproblemen oder inkonsistenten Eingangsnachrichten.

Programmiersprache: Python

Die Implementierung des Database Writers erfolgte in Python. Die Entscheidung basiert auf der Kombination aus vorhandener Expertise im Projektteam, umfangreicher Bibliotheksunterstützung für JSON-Verarbeitung, Messaging-Systeme und Datenbankzugriffe sowie der leichten Wartbarkeit der Servicestruktur. In Python stehen mit Bibliotheken wie `paho-mqtt`[103], `psycopg2` und `json` sofort einsatzfähige und stabile Werkzeuge für alle Teilaufgaben zur Verfügung.

Vergleich mit Alternativen: Eine Implementierung in Go wäre performant und spechereffizient, jedoch mit deutlich höherem Aufwand bei der Bibliotheksintegration verbunden gewesen. Java hätte ebenfalls umfassende JDBC-basierte Anbindungen an relationale Datenbanken geboten, ist aber insbesondere für prototypische Implementierungen überdimensioniert und aufwändig in Bezug auf Konfiguration und Deployment.

Begründung: Die Wahl von Python stellt einen optimalen Kompromiss zwischen Funktionalität, Einfachheit und Flexibilität dar. Darüber hinaus konnte die gemeinsame Sprache mit den übrigen Schnittstellen-Services genutzt werden, was die Homogenität und Wartbarkeit der Gesamtplattform verbessert.

Messaging-Empfang: MQTT

Der Database Writer konsumiert eingehende Nachrichten über das MQTT-Protokoll, das vom zentralen Messaging-System vermittelt wird. MQTT wurde ausgewählt, da es mit seinem Publish/Subscribe-Paradigma und dem minimalen Protokoll-Overhead ideal auf

intermittierende und latenzempfindliche Kommunikationsbeziehungen zwischen Sensoren und Auswertungssystemen zugeschnitten ist.[104]

Alternative Protokolle:

- **AMQP:** Industriestandard für Messaging, jedoch schwergewichtiger und nicht speziell für IoT optimiert.
- **HTTP Push:** Einfach zu implementieren, aber ungeeignet für kontinuierliche, bidirektionale Datenströme.
- **WebSockets:** Echtzeitfähig, jedoch komplexer in der Integration mit klassischen Persistenzsystemen.

Begründung: MQTT bietet ein exzellentes Gleichgewicht zwischen Zuverlässigkeit, Ressourceneffizienz und Verbreitung im IoT-Bereich. Die Persistenzfunktionen des zugrunde liegenden Messaging-Brokers stellen zudem sicher, dass keine Daten durch kurzzeitige Netzwerkausfälle verloren gehen.

Fehlerbehandlung und Wiederholungsmechanismen

Im Database Writer wurden explizite Retry-Mechanismen für die Datenbankanbindung integriert, um Datenverluste bei vorübergehender Nichtverfügbarkeit zu verhindern. Dies erfolgt über ein warteschlangenbasiertes Zwischenspeichern nicht erfolgreicher Schreiboperationen, die periodisch erneut angestoßen werden.

Alternative Ansätze:

- **Fire-and-Forget-Ansatz:** Einfach, aber inakzeptabel bei Sicherheits- oder Zuverlässigkeitserfordernissen.
- **Transaktionsbasierte Protokolle:** Wären robuster, jedoch mit hohem Implementierungsaufwand verbunden.

Begründung: Die gewählte Methode stellt sicher, dass Datenverlust unter realistischen Bedingungen nahezu ausgeschlossen ist, ohne den Entwicklungsaufwand über Gebühr zu erhöhen. Die Wiederholungslogik kann bei Bedarf skalierbar erweitert werden.

4.8.4 Technologieauswahl für die Zielwert-Schnittstelle (Setpoint API)

Zielsetzung und Kontext

Die Setpoint API ermöglicht die gezielte Übertragung von Sollwerten an Steuercontroller, welche wiederum für die Regelung der Wasserzufuhr einzelner Pflanzen verantwortlich sind. Damit bildet sie eine Brücke zwischen Benutzeranwendungen bzw. Systemkomponenten mit Steuerlogik und der dezentralen Aktorik des Systems. Die wesentliche Herausforderung besteht darin, eine flexible, zielgerichtete Kommunikation mit hoher Ausfallsicherheit und Gerätespezifität zu gewährleisten.

Programmiersprache und Framework: Python mit Flask und Flasgger

Die Implementierung erfolgte in Python, da bereits andere Systemkomponenten mit dieser Sprache realisiert wurden und eine hohe Wiederverwendbarkeit von Code sowie Konsistenz bei der Konfiguration gewährleistet werden sollte. Flask wurde als Web-Framework verwendet, da es eine schlanke Struktur mitbringt, sich hervorragend für RESTful-Services eignet und geringe Anforderungen an die Serverinfrastruktur stellt. Ergänzt wurde dies durch die Nutzung von Flasgger zur Generierung einer automatisierten Swagger-Dokumentation der REST-Endpunkte.[105]

Vergleich mit Alternativen: Alternativ wäre FastAPI als modernere REST-Plattform denkbar gewesen, die native OpenAPI-Dokumentation, Validierung und asynchrone Verarbeitung unterstützt. Allerdings wurde Flask aufgrund des stabileren Lern- und Erfahrungsstands im Team sowie vorhandener Funktionalität bevorzugt.

Begründung: Die Kombination aus Flask und Flasgger ermöglichte eine wartbare und klar dokumentierte Schnittstelle mit kurzer Entwicklungszeit und einfacher Erweiterbarkeit. Die REST-Architektur passte gut zur Anforderung, kontrolliert und gezielt Steuerinformationen zu senden.

Kommunikationsmodell: Publish/Subscribe via MQTT über Topics

Für die Weiterleitung der Sollwertnachrichten an den jeweiligen Controller wird das MQTT-Protokoll mit einer thematischen Strukturierung der Topics genutzt. Jeder Controller erhält ein dediziertes Topic, dessen Aufbau die eindeutige Zuordnung der Nachricht ermöglicht. Dies entspricht dem Prinzip der Device-Adressierung in der IoT-Kommunikation.

Vergleich mit Alternativen:

- **HTTP POST:** Wäre theoretisch für Push-Verhalten geeignet, erfordert jedoch persistente Adressen und erschwert dynamische Subscriptions.
- **AMQP:** Komplexer, mit mehr Overhead für das Szenario der gezielten Steuerung einzelner Endpunkte.

Begründung: MQTT ermöglicht eine sehr leichtgewichtige Übermittlung mit der Option, persistente oder volatile Nachrichten zu senden. Die Topic-Struktur bietet eine flexible Adressierung ohne zusätzliche Verwaltungslogik. Die vorhandene Messaging-Infrastruktur mit Solace wurde konsequent weiterverwendet, was den Integrationsaufwand gering hielt.

Nachrichtenerstellung und Serialisierung

Die Struktur der Sollwertnachricht wurde als JSON konzipiert, um sowohl Menschenlesbarkeit als auch Systemkompatibilität zu sichern. Die Erstellung erfolgt mit Hilfe von Python-Bordmitteln, wodurch externe Abhängigkeiten minimiert werden konnten.

Alternative Formate:

- **XML:** Etabliert, aber komplexer in der Verarbeitung.
- **Protobuf:** Effizient, aber für kleinere Projekte überdimensioniert und weniger transparent.

Begründung: JSON bietet einen ausgezeichneten Kompromiss zwischen Standardisierung, Einfachheit und Interoperabilität. Es ist gut durch Firewalls und Broker-Systeme hindurch transportierbar und in praktisch jeder Sprache verarbeitbar.

4.8.5 Technologieauswahl für den Initialisierungsskript-Dienst (Solace Init)

Zielsetzung und Kontext

Der Solace Init-Dienst wurde als automatisierter Initialisierungsmechanismus konzipiert, um beim Start des Systems die für die Kommunikationsarchitektur erforderlichen Messaging-Objekte auf dem Solace Broker anzulegen. Dabei handelt es sich insbesondere um Queues mit spezifischen Topic-Subscriptions, deren Struktur die Grundlage für das geräteindividuelle Messaging im gesamten Sensorsa-System bildet. Ziel war es, eine einmalige, reproduzierbare und konfigurierbare Initialisierung ohne manuelle Eingriffe zu ermöglichen.

Begründung der Broker-Wahl: Solace PubSub+ als Messaging-Plattform

Die Wahl des Message Brokers stellt eine der zentralen Architekturentscheidungen im Sensorsa-Projekt dar. Die Anforderung bestand darin, ein performantes, fehlertolerantes und hochflexibles Messaging-System zu integrieren, das sowohl klassische Publish/Subscribe-Kommunikation als auch spezifische Anforderungen an Filterung, Persistenz und Authentifizierung erfüllt. Nach einem Vergleich mehrerer etablierter Systeme fiel die Entscheidung auf **Solace PubSub+.**[106]

Vergleich mit Alternativen:

- **Apache Kafka:** Hervorragend für Event-Streaming und hohe Datenvolumina, jedoch nicht natürlich auf Topic-basiertes IoT-Publish/Subscribe ausgerichtet und ohne eingebaute Message Routing Features wie Topic Wildcards.[107]
- **RabbitMQ:** Solide AMQP-Implementierung mit guter Dokumentation, jedoch schwächer in Bezug auf native MQTT-Unterstützung, dynamische Topic-Strukturen und granulare Zugriffssteuerung auf Topics.
- **Mosquitto:** Leichtgewichtig und speziell auf MQTT ausgelegt, aber eingeschränkt in Bezug auf Sicherheitsfeatures, Persistence-Mechanismen und Administration auf Enterprise-Niveau.

Stärken von Solace: Solace PubSub+ verbindet als Enterprise-Grade-Plattform mehrere Vorteile, die sich direkt aus den Anforderungen des Sensorsa-Projekts ergeben:

- **Native Unterstützung mehrerer Protokolle:** Solace unterstützt MQTT, REST, AMQP und WebSockets nativ auf einem Broker. Damit konnten sowohl sensornahe Kommunikation (*MQTT*) als auch service-interne Schnittstellen (*REST*) integriert werden, ohne unterschiedliche Middleware-Lösungen kombinieren zu müssen.
- **Feingranulare Topic-Strukturen und Wildcards:** Für die Adressierung einzelner Controller oder Sensortypen wurden hierarchische Topics mit Wildcard-Unterstützung genutzt, wodurch sich flexible Abonnements realisieren ließen.
- **Skalierbare Persistenzmechanismen:** Solace bietet sowohl volatile als auch persistente Delivery-Modes mit garantierter Zustellung, was für zeitkritische Steuerinformationen entscheidend ist.
- **Zentrale Administration via SEMPv2:** Die REST-basierte Konfigurationschnittstelle (SEMPv2) erlaubt automatisierte, containerkompatible Initialisierungsskripte wie den hier beschriebenen Init-Service.

- **MQTT optimiert für IoT:** Die MQTT-Implementierung von Solace ist mit Fokus auf Latenzreduktion, Lastverteilung und Delivery-Garantien implementiert und eignet sich ideal für Embedded-Gerätekommunikation.
- **Security-Features:** ACL-Management, Authentifizierung auf Benutzer- und Topic-Ebene sowie TLS-Unterstützung ermöglichen ein differenziertes Sicherheitskonzept.

Zusammenfassende Begründung: Solace PubSub+ wurde gewählt, da es in einzigartiger Weise hohe Ansprüche an Zuverlässigkeit, Integrationstiefe und Skalierbarkeit erfüllt. Die native Unterstützung von MQTT ist für IoT-Anwendungen essenziell, während die gleichzeitige Verfügbarkeit von Management- und Sicherheitsfunktionen auf Enterprise-Niveau den reibungslosen Betrieb in containerisierten Umgebungen sicherstellt. Darüber hinaus empfiehlt auch der Hersteller Solace selbst den Einsatz von Python für schnelle Prototypenentwicklung im Bereich IoT [108].

Konfigurationsbasis: JSON-Datei

Als zentrales Format für die Definition der zu erstellenden Queues und ihrer jeweiligen Subscriptions wurde eine strukturierte JSON-Datei verwendet. Dieses Format erlaubt eine deklarative Spezifikation der gesamten Messaging-Infrastruktur und kann sowohl von Menschen editiert als auch maschinell verarbeitet werden.[109]

Vergleich mit Alternativen:

- **YAML:** Ebenfalls menschenlesbar, jedoch fehleranfälliger bei komplexeren Strukturen und nicht nativ durch alle Python-Standardbibliotheken unterstützt.
- **XML:** Formal stark, aber syntaktisch schwergewichtig und in der Praxis für Konfigurationszwecke oft überdimensioniert.

Begründung: JSON erfüllt im Kontext des Projekts das optimale Gleichgewicht zwischen Lesbarkeit, Standardisierung und Softwareunterstützung. Es ermöglicht eine flexible Erweiterung der Initialisierungskonfiguration, z. B. durch Hinzufügen weiterer Queues oder komplexerer Subscription-Filter, ohne strukturelle Anpassungen am Code erforderlich zu machen.

Schnittstelle zur Messaging-Infrastruktur: Solace SEMPv2 API

Die eigentliche Initialisierung erfolgt über HTTP-Requests an die SEMPv2-API von Solace PubSub+, einer offiziellen Verwaltungs- und Konfigurationsschnittstelle des Brokers.

Diese REST-basierte Schnittstelle erlaubt das Anlegen, Konfigurieren und Prüfen von Messaging-Komponenten wie Queues, Topics und ACLs im laufenden Betrieb.

Vergleich mit Alternativen:

- **Admin GUI:** Bedienerfreundlich, aber nicht automatisierbar und nicht reproduzierbar.
- **CLI-Tools (solacectl):** Eher für DevOps-Prozesse geeignet, jedoch aufwendiger in der Einbettung in einen containerisierten Microservice.

Begründung: Die REST-API von Solace war für den Projektkontext besonders geeignet, da sie eine serviceinterne und vollautomatische Ansteuerung ermöglichte. Die Authentifizierung konnte über Umgebungsvariablen geregelt werden, die in Docker Compose konfiguriert wurden, sodass die Schnittstelle sowohl sicher als auch einfach nutzbar war. Zudem ermöglicht SEMPV2 die Definition granulare Subscription-Filter direkt beim Queue-Erstellen, was eine exakte Abbildung der Messaging-Logik ermöglicht.

Fehlertoleranz und Wiederanlauf

Das Init-Skript enthält einfache Mechanismen zur Wiederanlaufbarkeit. Existierende Queues werden nicht erneut erstellt, sondern übersprungen oder aktualisiert. Fehler bei der Kommunikation mit dem Solace-Broker werden protokolliert, und das Skript kann bei Bedarf mehrfach ohne Seiteneffekte ausgeführt werden.

Begründung: Diese Idempotenz ist essenziell für automatisierte Umgebungen, etwa bei der Nutzung von Docker Compose oder CI/CD-Pipelines. Durch sie wird vermieden, dass der Initialisierungsprozess bei einem Neustart ungewollt Konfigurationsfehler verursacht oder bestehende Objekte zerstört.

4.9 Vergleichsanalyse: ESP32-WROOM-32D vs. Arduino vs. Raspberry Pi

Für die Steuerung der Bewässerungs- und Sensoreinheit soll ein Mikrocontroller eingesetzt werden. Drei der sehr häufig für solche Aufgaben eingesetzte Mikrocontroller-/Computerplattformen sind: der **ESP32 von Espressi** (z.B der ESP32-WROOM-32D), der **Arduino Nano Every** mit ATmega4809 Chipsatz und der **Raspberry Pi** Modell 4 oder Zero W. Dieses Kapitel vergleicht die Plattformen wissenschaftlich fundiert hinsichtlich den Anforderungen die in Kapitel 3.5 erarbeitet wurde. Explizit werden Vergleiche

hinsichtlich Architektur, Leistung, Peripherie, Netzwerkanbindung, Speicher, Energieeffizienz, Entwicklungsökosystem, Kosten und Projektanforderungen gezogen. Ziel ist es, eine begründete Entscheidung für einen Mikrocontroller/Computerplattform zu treffen und dessen Vorteile gegenüber anderen Plattformen herauszustellen.

4.9.1 Architektur und Rechenleistung

ESP32-WROOM-32D: Dieser Mikrocontroller basiert auf einem *32-Bit Dual-Core Ten-silica LX6 Microprozessor* mit einstellbarer Taktfrequenz bis zu 240 MHz. Der ESP32 verfügt über 520 KB SRAM und einen 4 MB Flash Speicher. [110] Zusätzlich existiert ein Ultra-Low-Power Coprozessor für einfache Aufgaben im Tiefschlafmodus. Die 32-Bit-Architektur und zwei Kerne erlauben deutlich höhere Rechenleistung als 8-Bit-Mikrocontroller. Tatsächlich kann der ESP32 selbst rechenintensive Aufgaben (z. B. DSP, verschlüsselte Kommunikation) ausführen und unterstützt Datenraten bis 150 Mbps über WLAN. Die Kombination aus hohem Takt, großzügigem RAM und Hardware-Beschleunigern (z. B. für Kryptografie) ermöglicht komplexe IoT-Anwendungen, wie etwa gleichzeitiges Sensor-Datenverarbeiten und Netzwerkkommunikation. [111]

Arduino Nano Every (ATmega4809): Der Arduino Nano Every nutzt einen *8-Bit AVR-Mikrocontroller (ATmega4809)* mit Harvard-Architektur und bis zu 20 MHz Taktfrequenz. Er bietet 48 KB Flash, 6 KB SRAM und 256 Bytes EEPROM. [112] Diese Werte liegen um Größenordnungen unter denen des ESP32 und Raspberry Pi (z. B. hat der klassische Arduino Uno nur 2 KB SRAM und 32 KB Flash). [113]

Der ATmega4809 wurde gegenüber älteren AVR-Modellen optimiert, besitzt z. B. einen Hardware-Multiplizierer und ein Event-System für effizientere interne Kommunikation. [112]

Dennoch ist die Rechenleistung aufgrund der 8-Bit-Architektur und begrenzten Frequenz deutlich eingeschränkt. Der Arduino eignet sich vor allem für deterministische, einfache Steuerungsaufgaben. Echtzeitnahe Reaktionen sind durch direkte Hardwaresteuerung möglich, allerdings stößt der ATmega bei großen Algorithmen oder Datenmengen schnell an Speicher- und Leistungsgrenzen. [110]

Raspberry Pi (4/Zero W): Raspberry Pis sind *Single-Board-Microcomputer* mit deutlich leistungsstärkerer CPU. Der Raspberry Pi 4 Model B integriert ein Broadcom *BCM2711 SoC mit Quad-Core ARM Cortex-A72 @ 1,5 GHz* (64-Bit ARMv8-Architektur) und je nach Modell 1–8 GB DDR4-RAM. [114] Selbst der kleine Raspberry Pi Zero W nutzt einen 1 GHz Single-Core ARM11 (ARMv6) mit 512 MB RAM [115] – immer noch hunderte Male mehr RAM als Microcontroller bieten. [110]

Diese Architektur entspricht einem vollständigen Linux-Computer. Der Pi 4 kann somit anspruchsvolle Aufgaben wie Bildverarbeitung oder parallele Prozesse ausführen, die für Microcontroller unmöglich wären. [110] Allerdings ist diese hohe Leistung für eine Bewässerungssteuerung oft überdimensioniert. Zudem basiert der Pi (standardmäßig) auf einem Von-Neumann-Architektur-Prozessor mit Betriebssystem, was Vor- und Nachteile für Echtzeit- und IO-Operationen mit sich bringt.

Zusammenfassung: In puncto Rechenleistung liegt der Raspberry Pi 4 mit seinem 1,5 GHz Quad-Core und Gigabytes an RAM weit vorne – er erreicht PC-ähnliche Performance. [110] Der ESP32 hingegen stellt einen Kompromiss dar: Er ist wesentlich schneller und potenter als ein Arduino (32-Bit @240 MHz vs. 8-Bit @20 MHz; 520 KB vs. 6 KB RAM), bleibt aber ressourcenschonend genug für Embedded-Aufgaben. [113] [112] Der Arduino ATmega4809 ist architekturbedingt am unteren Leistungsende einzuordnen und bewältigt nur einfache, zeitkritische Steuerungsaufgaben in kleinem Maßstab zuverlässig.

4.9.2 Unterstützung von Multitasking / RTOS

Arduino (ATmega4809): Der ATmega4809 arbeitet ohne Betriebssystem. Es läuft ein einziger Main-Thread (Loop) und das Programm muss kooperativ alle Aufgaben sequentiell abarbeiten. Es gibt keine native Multitasking-Unterstützung. Echtzeitfähigkeit wird durch deterministische Ausführung und Interrupts erreicht – z. B. Timer-Interrupts können regelmäßige Tasks auslösen. Allerdings können nicht mehrere Tasks parallel auf dem Single-Core ausgeführt werden.

Ein RTOS lässt sich auf so kleinem AVR zwar theoretisch portieren, praktisch schränken aber 6 KB RAM dies stark ein. Man ist auf einfachen kooperativen Scheduler (oder Libraries wie [Arduino Scheduler](#)) beschränkt. Die Stärken des Arduino liegen in der präzisen Timing-Kontrolle einzelner Tasks (z. B. PWM-Signalgenerierung), nicht in echter Parallelverarbeitung. [110] Sobald man Sensorabfrage, Pumpensteuerung und Netzwerkkommunikation gleichzeitig umsetzen möchte, gerät das Single-Thread-Konzept an Grenzen – eine Aufgabe muss warten, während die andere läuft.

ESP32-WROOM-32D: Der ESP32 unterstützt Multitasking out-of-the-box. Espressif setzt als Betriebsumgebung auf *FreeRTOS* (eingebettet in das ESP-IDF und Arduino-Framework). [111] Dadurch können auf den zwei CPU-Kernen mehrere Threads/Tasks mit Prioritäten laufen. Beispielsweise kann ein Task die Sensoren lesen, während ein anderer parallel die WLAN-Verbindung und MQTT-Kommunikation handhabt – der FreeRTOS-Scheduler sorgt für zeitliches Scheduling nach Priorität.

Diese echte Parallelität (einschließlich *Symmetric Multiprocessing* auf zwei Kernen) ist

ein großer Vorteil gegenüber dem Arduino. Zudem ist FreeRTOS ein *RTOS*, d. h. Tasks können mit definierten Fristen laufen, was für Echtzeitanforderungen relevant ist.

Der ESP32 kombiniert also die Echtzeitfähigkeit eines Mikrocontrollers mit einem Multithreading Betriebssystem in leichter Form. Dies ist ideal, um z. B. zeitkritische Sensor-Auslese-Interrupts und weniger kritische Netzwerkaufgaben sauber zu trennen. Wichtig zu erwähnen: *TLS 1.2*-Unterstützung mit Hardware-Beschleunigung ist ebenfalls im ESP32-OS verankert, [111] was sichere parallele Netzwerkkommunikation ermöglicht. Insgesamt ist der ESP32 für Task-Parallelisierung mit Priorisierung hervorragend geeignet – ein entscheidender Pluspunkt im komplexeren IoT-Projekt.

Raspberry Pi (Linux): Der Raspberry Pi betreibt standardmäßig ein vollständiges Linux (z. B. Raspberry Pi OS). Dieses ist ein präemptives Multitasking-Betriebssystem, das beliebig viele Prozesse/Threads verwalten kann. Für parallele Abläufe (Sensor lesen, Pumpe steuern, Daten senden) bietet Linux *Multi- Threading* und *Multi-Prozess*-Fähigkeiten, weit über FreeRTOS hinaus.

Allerdings ist Linux kein Echtzeitbetriebssystem. Die Planung der Tasks erfolgt nach Fairness und Durchsatz, nicht streng deterministisch. Echtzeitanforderungen können unter Linux zu Verzögerungen führen (z. B. kann ein hoher CPU-Load oder ein ungünstig terminiertes Scheduler-Zeitfenster das Auslesen eines Sensors verzögern). Zwar erlaubt Linux *nice*-Werte und RT-Prioritäten für Threads, doch ohne spezielle Patches (*PREEMPT_RT*) erreicht es nicht die Hard-Real-Time von Mikrocontrollern.

Anders formuliert: Der Pi kann mehrere Aufgaben gleichzeitig ausführen, was für MQTT, Webserver etc. vorteilhaft ist, aber es besteht ein Jitter in zeitkritischen Abläufen. Für dieses Projekt (Sensorintervalle im Sekunden- oder Minutenbereich) ist das unkritisch; die Parallelität wäre sogar nützlich. Dennoch muss man beachten, dass der Pi weitere Software-Overheads hat (z. B. ein ganzes OS, Hintergrunddienste), was zu potenziell geringerer Zuverlässigkeit in Langzeitbetrieb führen kann (etwa, wenn ein Prozess abstürzt).

Zusammenfassung: Der Arduino bietet keine echte Multitasking-Unterstützung – alles läuft in einer Schleife. Der ESP32 nutzt hingegen *FreeRTOS*, was echtes Multithreading mit Priorisierung ermöglicht. [111] Der Raspberry Pi ermöglicht durch sein Linux vielseitiges Multitasking, jedoch ohne Echtzeitgarantie. Für das Vorhaben mit gleichzeitiger Sensor-, Aktor- und Netzsteuerung bringt der ESP32 den besten Mix aus Parallelität und Echtzeitkontrolle (dedizierte Tasks für MQTT, Sensorik, Webserver etc.), während Arduino hier an eine Grenze stößt und der Pi unnötige Komplexität einführt.

4.9.3 Schnittstellenvielfalt und Sensoranbindung

Arduino (ATmega4809): Der ATmega4809 bietet 41 I/O-Pins (im 48-Pin Gehäuse), wovon auf Arduino-Boards typischerweise rund 20 als GPIO herausgeführt sind. Darunter sind 14 digitale I/O (von denen einige PWM-Ausgänge sind) und 8 analoge Eingänge (10-Bit-ADC). [112]

Schnittstellen: $1 \times \text{UART}$, $1 \times \text{SPI}$, $1 \times \text{I}^2\text{C}$ (*TWI*) sind hardwaremäßig vorhanden [112] – auf dem Nano Every ist z. B. UART für USB bereits belegt, I²C und SPI stehen am Header zur Verfügung. Analoge Sensoren können direkt an den Arduino angeschlossen werden (0–5 V Messbereich bei 10-Bit Auflösung). PWM-Ausgänge (z. B. 5 Pins am Nano Every [112]) erlauben die Steuerung analoger Aktoren (in diesem Fall könnte man die Pumpenleistung per PWM steuern, falls nötig).

Der Arduino unterstützt außerdem *externe Interrupts* an allen digitalen Pins [112], was z. B. für das Wake-up durch einen Sensorschwellenwert nutzbar ist. Insgesamt ist die Schnittstellenvielfalt solide, aber begrenzt: Nur je eine Einheit für SPI/I²C/UART bedeutet, dass man mehrere I²C-Sensoren zwar am Bus teilen kann, aber z. B. mehrere UART-Geräte schwierig sind. Für typische Sensoren (Feuchtigkeitssensor analog, evtl. ein I²C-Temp.-Sensor) reicht es aus.

ESP32-WROOM-32D: Der ESP32 integriert zahlreiche Peripherie-Schnittstellen. Insgesamt stehen bis zu 34 programmierbare GPIO-Pins zur Verfügung (abhängig vom Modul-Pinout; viele Module führen rund 25 heraus). Davon können 18 Pins als Analogeingänge (ADC) genutzt werden. [110] Der ADC des ESP32 ist 12-Bit, was eine feinere Auflösung als beim Arduino (10-Bit) bietet – jedoch ist die Genauigkeit in der Praxis etwas nichtlinear, was aber kalibrierbar ist.

Weiterhin besitzt der ESP32 2 DAC-Ausgänge (8-Bit), mit denen man analog Spannungen ausgeben kann – ein Vorteil gegenüber Arduino und Pi (keine DACs).

An Kommunikations-Schnittstellen bietet der ESP32 mehrfache UARTs, SPI- und I²C-Schnittstellen (jeweils mindestens $2 \times$ hardwareseitig verfügbar). Dazu kommen I²S (Audio-Daten), CAN-Bus (TwAI), kapazitive Touch-Eingänge, SD-Karten-Interface (SDIO) und PWM-fähige Timer in großer Zahl. [111]

Diese Vielfalt ist äußerst nützlich: Mehrere Sensoren lassen sich parallel ansteuern (z. B. ein UART für ein serielles Display, ein SPI für einen ADC-Chip etc.). Für dieses Projekt sind vor allem die Analogeingänge wichtig (Bodenfeuchtesensor), die beim ESP32 reichlich vorhanden sind. Auch digitale IOs (für Pumpensteuerung, LEDs etc.) sind mehr als genug da. Aufgrund des 3,3 V-Designs des ESP32 muss man ggf. Pegelwandler nutzen, falls 5 V-Sensoren vorhanden sind – ähnlich wie beim Raspberry Pi, während Arduino (5 V-Version) direkt 5 V lesen kann. Insgesamt übertrifft der ESP32 den Arduino hier

deutlich in der Anzahl und Vielfalt der Schnittstellen.

Raspberry Pi (4/Zero W): Raspberry Pis besitzen einen *40-Pin GPIO-Header* (Standard seit Pi B+, Pi 4 und Zero W sind identisch belegt). Davon sind beim Pi 4 rund 28 Pins als GPIO nutzbar. [114] Diese GPIO können per Software als digitale Ein-/Ausgänge verwendet werden und bieten auch alternative Funktionen: Es gibt mehrere I²C, SPI und UART Hardwarecontroller im SoC, die auf die Pins gemultiplext werden können. [114] In der Praxis stehen 2 I²C-Busse, 5 UARTs (wovon einer vollwertig und oft der Bluetooth-Chip zugewiesen, einer am Pin-Header als serielle Konsole) und 2 SPI zur Verfügung – jedoch teilen sich Funktionen die Pins (man wählt per Software Pin-Funktionen). Wichtig: Der Raspberry Pi hat keine analogen Eingänge an Bord. Analoge Sensoren (z. B. ein resistiver Feuchtigkeitssensor mit analoger Spannung) benötigen einen externen ADC-Chip (oft per I²C oder SPI, z. B. MCP3008). Das ist ein klarer Nachteil für Sensoranbindung, da zusätzlicher Hardware- und Programmieraufwand entsteht.

Für digitale Sensoren/Aktoren ist der Pi hingegen gut gerüstet: I²C- und SPI-Sensoren (z. B. für Temperatur, ADC, etc.) kann man anbinden, und viele GPIO können als PWM-Ausgang genutzt werden (über dedizierte Hardware-PWM auf bestimmten Pins oder notfalls Software-PWM). Der Pi Zero W bietet dieselben GPIO-Möglichkeiten wie der Pi 4, nur mit weniger Rechenleistung. Erweiterungs-HATs können weitere Schnittstellen hinzufügen.

Insgesamt ist die Schnittstellenvielfalt des Raspberry Pi moderat, mit dem Hauptmanko fehlender Analog-Eingänge. Dafür bietet der Pi aber *USB-Ports* (Pi 4: 2 × USB3.0 + 2 × USB2.0, [114] Zero W: Micro-USB OTG), worüber man theoretisch auch Sensor-Devices (oder z. B. Arduino als Sensorinterface) anschließen könnte – was jedoch für einfache IoT-Sensorik meist Overkill ist.

Zusammenfassung: Für typische Sensor-/Aktor-Anforderungen bietet der Arduino solide Grundfunktionen (begrenzte Zahl analoger/digitaler Pins und eine Schnittstelle pro Bus).

Der ESP32 glänzt mit vielen ADC-Kanälen, mehreren UART/I²C/SPI und Spezialinterfaces – ideal für komplexere Sensor-Ausstattung. Der Raspberry Pi hat ebenfalls ausreichend digitale IO und diverse Busse, aber keine eingebauten ADCs, was für analoge Feuchtesensoren ein Nachteil ist.

Beide, ESP32 und Pi, arbeiten mit 3,3 V-Logik (Vorsicht bei 5 V-Sensoren), während ein klassischer Arduino 5 V I/Os bietet – in unserem Projekt sind Sensoren allerdings meist 3,3 V-kompatibel oder benötigen ohnehin eigene Treiber.

4.9.4 Netzwerkfähigkeit

Arduino (ATmega4809): Netzwerkanbindung ist beim ATmega4809 nicht integriert. Weder WLAN noch Bluetooth oder Ethernet sind on-board vorhanden. Ein Arduino Nano Every verfügt lediglich über die Mikrocontroller-Schnittstellen; um Netzwerkkommunikation (z. B. MQTT über WLAN) zu realisieren, müsste man zusätzliche Module einsetzen – z. B. ein ESP8266/ESP32 als WiFi-Modul oder ein Ethernet-Shield. Das bedeutet mehr Hardware und Programmieraufwand (und i. d. R. auch höhere Stromaufnahme).

Der ATmega selbst hat *keinen TCP/IP-Stack*, dieser müsste in Software realisiert werden (z. B. durch die Library des jeweiligen WiFi/Ethernet-Moduls). Das sprengt oft die Ressourcen des AVR. Bluetooth ist ebenfalls nicht vorhanden (ausgenommen separate BT-Module). Kurzum: „Standalone“ ist der Arduino ATmega4809 nicht netzwerkfähig. [116]

Es gibt Arduino-Boards mit integrierter WLAN-Funktion (z. B. Arduino Nano 33 IoT mit SAMD21 + ESP32 als NINA-W102 Modul), aber diese basieren auf anderen MCUs. Im Kontext des Nano Every müsste man zugunsten von Netzwerk einen deutlichen Mehraufwand treiben, weshalb Arduino hier deutlich im Nachteil ist.

ESP32-WROOM-32D: Der ESP32 wurde für IoT-Anwendungen entwickelt und bietet daher eingebautes WLAN und Bluetooth. Konkret unterstützt das Modul *2,4 GHz WLAN (802.11 b/g/n)* sowie *Bluetooth 4.2 + BLE (Dual Mode)*. [110]

Er implementiert einen vollständigen *TCP/IP-Stack (LwIP)* auf dem Chip, [111] wodurch Protokolle wie TCP, UDP, HTTP und MQTT direkt laufen können. MQTT lässt sich auf dem ESP32 mittels verfügbarer Bibliotheken (z. B. AsyncMQTT-Client oder PubSub-Client in Arduino IDE, oder native ESP-IDF MQTT) betreiben – die Unterstützung ist umfassend und praxiserprobт.

Wichtig für sichere IoT-Anwendungen: Der ESP32 beherrscht *TLS 1.2 mit Hardware-Beschleunigung*, [111] d. h. auch verschlüsselte MQTT-Verbbindungen (MQTTSS) oder HTTPS-Webserver sind möglich, ohne den Hauptprozessor zu überlasten.

Ein ESP32 könnte z.B. über WLAN ins lokale Netz eingebunden, könnte sich mit einem MQTT-Broker verbinden und Sensordaten publizieren sowie Befehle empfangen. Die Netzwerk-Performance ist für IoT Zwecke mehr als ausreichend (802.11n bis 150 Mbit/s und 20 dBm Sendeleistung für gute Reichweite). [111]

Zudem kann der ESP32 auch im Access-Point-Modus betrieben werden, um z. B. für die Erstkonfiguration ein eigenes WLAN bereitzustellen (siehe Webinterface). Bluetooth könnte optional für eine Handy-App-Verbindung genutzt werden. Insgesamt ist die Netzwerkfähigkeit des ESP32 hervorragend und speziell auf IoT zugeschnitten – ein entscheidender Vorteil gegenüber dem Arduino.

Raspberry Pi (4/Zero W): Raspberry Pis haben bereits seit Modell 3 WLAN und Bluetooth integriert. Der Raspberry Pi 4 Model B bietet *Dual-Band WLAN (2,4 GHz & 5 GHz, 802.11ac)* sowie *Bluetooth 5.0 + BLE* on-board. [114] Zusätzlich hat der Pi4 einen *Gigabit-Ethernet-Port*, [114] was in stationären Anwendungen sehr zuverlässig und performanzstark ist.

Der Pi Zero W verfügt „nur“ über *2,4 GHz WLAN (802.11n)* und *Bluetooth 4.1 + BLE*, [115] aber für unser Anwendungsszenario ist das vollkommen ausreichend.

Softwaretechnisch läuft auf dem Raspberry Pi ein vollwertiger *Linux-Netzwerkstack* – die Möglichkeiten sind hier am umfangreichsten. MQTT kann direkt auf dem Pi z. B. mittels Python (paho-mqtt Library) implementiert werden; auch komplexe Protokolle oder sogar ein eigener Broker ließen sich betreiben. Der Pi kann gleichzeitig als *MQTT-Client, Webserver, SSH-Server* etc. agieren. Für die Erstkonfiguration könnte man z. B. einen Hotspot einrichten oder per angeschlossenem Monitor konfigurieren.

Allerdings bringt diese Vielfalt auch mehr Angriffsfläche: Ein Linux-System muss gepflegt (Updates, Absicherung) und z. B. mit Firewall konfiguriert werden, während ein ESP32-Gerät weitgehend nur die gezielt programmierte Firmware ausführt. Dennoch ist der Raspberry Pi in puncto Netzwerk sehr leistungsfähig: Insbesondere bei hohem Datenaufkommen oder Bedarf an komplizierten Authentifizierungsmechanismen hat er genug Reserven.

Für unseren Anwendungsfall (WLAN, MQTT) bietet er ähnliche Funktionalität wie der ESP32, jedoch mit höherem Energieverbrauch und System-Overhead.

Zusammenfassung: Der ESP32 und Raspberry Pi verfügen beide über integrierte WLAN und Bluetooth-Fähigkeiten und eignen sich direkt für MQTT-Kommunikation.

Der Arduino ATmega4809 dagegen besitzt keine eingebaute Netzwerkhardware – dies wäre ein erheblicher Nachteil, da zusätzliche Module benötigt würden.

Der ESP32 punktet mit speziell optimiertem IoT-Stack (FreeRTOS + LwIP) und Hardware-TLS, während der Raspberry Pi durch sein Linux nahezu unbegrenzte Netzwerk Softwaremöglichkeiten hat.

Im Kontext des Projekts (WLAN, MQTT) erfüllen ESP32 und Raspberry Pi alle Anforderungen, wobei der ESP32 dies mit deutlich schlankerer Hardware tut.

4.9.5 Speicher- und Persistenzmöglichkeiten

Arduino (ATmega4809): Der Mikrocontroller bietet neben Flash und RAM auch einen kleinen *nichtflüchtigen EEPROM-Speicher* von 256 Bytes. [112] Darin können z. B. Kalibrierungsdaten oder Konfigurationsparameter dauerhaft gesichert werden. 256 Bytes sind allerdings sehr begrenzt – genug für ein paar Einstellungen. WLAN-Credentials z. B.

passen dort kaum hinein, MQTT-Zugangsdaten nur kann wen sie in kompakter Form vorliegen.

Der Flash (48 KB) ist primär für das Programm, könnte aber theoretisch zu einem Teil für Softwarereserven (Konstanten, Logs) genutzt werden – üblich ist das beim Arduino aber nicht, da das Schreiben ins Flash vom Bootloader nicht direkt unterstützt wird.

Externe Speicher lassen sich anbinden: Der Arduino kann über SPI eine SD-Karte ansteuern (es gibt Arduino-SD-Libraries), um z. B. größere Datenmengen zu loggen. Auch FRAM- oder EEPROM-Chips per I²C wären möglich. On-board bleibt es jedoch bei sehr knappem persistentem Speicher. Für eine Bewässerungs-IoT-Anwendung bedeutet das: Einfache Werte (wie zuletzt gemessene Feuchte oder Pumpen-Laufzeit) könnte man im EEPROM halten; komplexe Datenspeicherung (z. B. Verlaufsdaten über Tage) erfordert externe SD-Karte oder das Senden der Daten an einen Server.

ESP32-WROOM-32D: Das ESP32-Modul enthält *4 MB externen Flash*. [110] Darauf liegt der Programmcode, aber typischerweise ist auch Platz für ein Dateisystem (SPIFFS oder LittleFS) sowie den *Nichtflüchtigen Speicher (NVS)* vorgesehen.

Espressif's NVS ist ein Schlüssel-Werte-Speicher im Flash, der vom SDK bereitgestellt wird und ähnlich wie ein EEPROM genutzt werden kann – allerdings mit deutlich größerer Kapazität (üblicherweise einige Kilobyte). Laut Entwicklerdokumentation lassen sich mehr als 2 KB an Daten, wie z. B. WiFi-Zugangsdaten oder Geräteeinstellungen, dauerhaft im NVS ablegen. [117] Dabei sorgt ein Wear-Leveling dafür, dass der Flash nicht durch häufiges Schreiben verschleißt.

Für umfangreichere Daten kann der ESP32 ein SPI-Flash-Dateisystem nutzen; man kann also z. B. Sensorhistorie in einer Datei speichern, wenngleich die Obergrenze der Dateigröße durch 4 MB insgesamt festgesetzt sind.

Darüber hinaus unterstützt der ESP32 wie Arduino auch das Anbinden von SD-Karten (entweder im SPI-Modus oder über das SD/SDIO-Interface) – so kann man theoretisch unbegrenzten Speicher ergänzen, falls gewünscht.

Für dieses Projekt ist vorrangig wichtig, dass Konfigurationen und Zustände (etwa WLAN-Credentials, Schwellwerte für Feuchte) im Gerät gehalten werden können: Dies lässt sich im ESP32 problemlos über NVS realisieren. [117] Der Programmspeicher (Flash) ist mit 4 MB groß genug für unseren Code (zum Vergleich: Arduino: 48 KB, der ESP32 ~4096 KB). Auch der *Arbeitsspeicher (520 KB SRAM)* erlaubt das Puffern von Daten oder Web-Seiten im Speicher, ohne sofort an Grenzen zu stoßen. Insgesamt ist der ESP32 bezüglich Speicher/Persistenz sehr flexibel. Es ist genug Speicher für Konfiguration und kleine Logs vorhanden, der bei Bedarf zusätzlich noch erweiterbar ist.

Raspberry Pi: Der Pi nutzt ein anderes Konzept: Es gibt keinen on-board Flash (au-

ßer einen kleiner EEPROM für den Bootloader, der für Nutzerdatenspeicherung irrelevant ist). Stattdessen dient eine microSD-Karte als Massenspeicher [110] – auf ihr liegen Betriebssystem, Programme und Daten. Diese SD-Karte kann sehr groß sein (theoretisch bis 2 TB, praktisch üblich 8–64 GB oder mehr).

Damit steht dem Raspberry Pi Speicher im Gigabyte-Bereich zur Verfügung – mehr als genug für Logging, Datenbanken, Web-Content und alle Daten die gespeichert werden müssen. Der Pi kann somit lokal sehr viel speichern (z. B. jahrelange Sensorhistorie). Auch Datenbanken oder CSV-Logs könnten direkt auf der SD ablegt werden.

Ein Problembereich ist jedoch die Zuverlässigkeit: SD-Karten können bei häufigen Schreibzugriffen oder unsauberem Herunterfahren kaputtgehen oder Daten korrumpern. In einem 24/7-IoT-Gerät, das potentiell hart vom Strom getrennt werden kann (z. B. Batterie alle), besteht ein Risiko von Dateisystemkorruption. Es gibt zwar Gegenmaßnahmen (Read-Only-System, Watchdog etc.), aber trotzdem ist die Umsetzung komplexer als bei einem Microcontroller, der kein anfälliges FS benötigt.

An Arbeitsspeicher bringt der Pi 4 *mindestens 1 GB RAM* mit – das ist reichlich für Caches und Verarbeitung. Der Pi Zero W hat 512 MB RAM [115], immer noch in einer anderen Liga als ESP32 (0.5 MB) oder Arduino (0.006 MB). Für den gegebenen Anwendungsfall bedeutet das: Der Pi könnte neben Konfiguration auch z. B. Webinterface-Dateien, Zertifikate etc. auf der SD-Karte ablegen. Persistente Einstellungen ließen sich im Dateisystem speichern (etwa in einer Textdatei oder mittels SQLite). Die Speicherbegrenzung ist faktisch kein Thema beim Raspberry Pi. Allerdings erkauft man dies mit dem genannten Verlässlichkeitsproblem und dem Umstand, dass man immer eine einfach zugängliche SD-Karte als Teil des Systems hat.

Zusammenfassung: Arduino: *sehr knapper Persistenzspeicher (256 B EEPROM)* [111] – gerade genug für minimale Konfigurationen, größere Daten extern speichern.

ESP32: *integrierter Flash (4 MB)* mit flexiblem NVS/Filesystem – ausreichend für Konfigurationsdaten und moderate Log-Daten [117]; Erweiterung via SD möglich.

Raspberry Pi: Gigabyte-Speicher auf SD-Karte, quasi unbeschränkt, jedoch mit Dateisystem und dessen Vor- und Nachteilen (viel Platz, aber potenziell störanfällig bei Stromverlust).

Für dieses Projekt sind keine Massendaten vor Ort nötig (MQTT überträgt Sensordaten an einen Server), wichtiger sind robuste Speicherung der Einstellungen (WLAN, Schwellwerte). Das können ESP32 und Pi beide gut – der Arduino nur sehr eingeschränkt.

4.9.6 Energieverbrauch und Batterietauglichkeit

Arduino (ATmega4809): AVR-Mikrocontroller sind für ihre niedrige Leistungsaufnahme bekannt. Der ATmega4809 im Nano Every zieht im aktiven 5V-Betrieb typischerweise rund 19 mA [110] (bei 16–20 MHz Takt), maximal um 50 mA, falls alle Peripherien aktiv sind.

In Schlafmodi kann der Strom auf wenige μA gesenkt werden (z. B. Power-down mit Timer ohne BOD im Sub-10 μA Bereich, genaue Werte laut Datenblatt typ. $<5\text{ }\mu\text{A}$). [112] Das bedeutet, ein Arduino kann sehr effizient mit Batterien laufen, insbesondere wenn er die meiste Zeit schläft und nur periodisch aufwacht. Im gegebenen Anwendungsszenario kann ein Arduino mit geeigneter Funk-Hardware über Monate nur durch eine Batterie mit Energie versorgt laufen. Sobald wir WLAN mit einbezogen wird, ändert sich das Bild. Ein externes WLAN-Modul (z. B. ESP01) benötigt beim Senden deutlich mehr Strom (ESP8266 $\sim 70\text{ mA}$ durchschnittlich beim WiFi-Betrieb). Der Arduino selbst behält zwar seinen geringen Verbrauch, aber das Gesamtsystem mit WiFi ist dann nicht mehr so energiesparend.

Ohne Netzwerkanbindung ist der Arduino trotzdem unschlagbar effizient – für rein lokale Steuerungen ideal. In Summe: Batteriebetrieb ist mit einem Arduino gut machbar, wenn die Funkkommunikation entweder selten erfolgt oder über ein ebenfalls energiesparendes Protokoll (z. B. LoRa) ausgelagert wird. Für Dauer-WLAN-Verbindung ist ein Arduino allein ungeeignet, da das notwendige WiFi-Modul die Batterietauglichkeit stark reduziert.

ESP32-WROOM-32D: Der ESP32 ist leistungsfähiger, benötigt im Betrieb aber mehr Strom als ein 8-Bit AVR. Laut Espressif liegt der typische Betriebsstrom bei $\sim 80\text{ mA}$ (bei eingeschaltetem WLAN und aktiver CPU). [111] Peaks beim Senden können um 200–300 mA liegen (kurzzeitig). Allerdings unterstützt der ESP32 verschiedene Schlafmodi: Im *Deep Sleep* mit ULP-Koprozessor liegt der Verbrauch nur noch bei $\sim 10\text{--}150\text{ }\mu\text{A}$ (je nach Konfiguration). [118] [110] Espressif gibt an, dass der *Schlafstrom* $<5\text{ }\mu\text{A}$ betragen kann. [111] Praktisch sind bei Entwicklungsboards (wegen Spannungsregler, LEDs) um $\sim 100\text{ }\mu\text{A}$ erreichbar.

Im Modem-Sleep (WLAN verbunden aber idle) reduziert sich der ESP32-Verbrauch ebenfalls signifikant. Der aktive Verbrauch von $\sim 80\text{ mA}$ bedeutet, dass der ESP32 im Dauerbetrieb (always on WiFi) mehr Energie benötigt als ein Arduino. Doch für Batteriebetrieb kann das Muster optimiert werden: z. B. 90% der Zeit schlafen, nur kurz aufwachen, messen, Daten senden.

Ein Beispiel: textit20,5 mA aktiv und 0,16 mA im Tiefschlaf führt zu einem durchschnittlichen Strom von wenigen mA, wenn die Wachphasen kurz sind. [110] Damit kann ein

ESP32-System oft monatelang mit einer 18650-Zelle betrieben werden. In diesem Projekt (permanente MQTT-Verbindung vs. periodische Übertragung) muss untersucht werden ob kontinuierliches MQTT-Listening gefordert ist. Dann muss das WiFi ständig an sein, was den Verbrauch erhöht.

Alternativ kann der ESP32 in einen Leichtschlaf gehen und per Beacon-Wakeup die Verbindung halten – oder nur zu Intervallen sich verbinden. Jedenfalls ist der ESP32 darauf ausgelegt, auch batteriebetrieben eingesetzt zu werden, was durch den sehr niedrigen Deep-Sleep-Strom unterstrichen wird. [111]

Viele IoT-Sensorprojekte setzen erfolgreich auf ESP32 + Batterie. Gegenüber Arduino + separatem WiFi hat der ESP32 hier sogar Vorteile, da keine zwei Chips versorgt werden müssen und das Powermanagement integriert ist.

Raspberry Pi (4/Zero W): Hier zeigt sich der größte Unterschied. Raspberry Pis sind nicht primär für den Betrieb mit einer Batterie entworfen. Ein Raspberry Pi 4 Model B hat durch seinen leistungsstarken SoC einen *Leerlauf-Strom von ca. 600 mA (5 V)*, entsprechend etwa 2,85 W Leistungsaufnahme. [114]

Unter Last kann er 1–2 A ziehen (bis ~6–7 W), je nach Nutzung von CPU, GPU und Peripherie. Dieses Niveau ist zwei Größenordnungen höher als bei Microcontrollern. Ein Pi 4 würde einen typischen 2500 mAh Handy-Akku in wenigen Stunden leeren. Selbst der kleinere Pi Zero W ist im Vergleich sehr hoch im Verbrauch: Gemessene Idle-Verbräuche liegen um ~120 mA (0,6 W bei 5 V), mit Peaks bis ~170 mA.[115] Das ist zwar deutlich weniger als der Pi 4, aber immer noch ca. 50-mal höher als ein schlafender ESP32. Zudem gibt es keinen Tiefschlafmodus beim Raspberry Pi im üblichen Sinn – man kann ihn nicht wie einen Microcontroller in Sleep setzen und per RTC wecken. Man könnte den Pi zwar komplett herunterfahren (dann zieht er aber immer noch ~20–30 mA für den PMIC) [114], aber ein Aufwachen erfordert dann ein externes Signal (z. B. Hard-Powercycle). Für einen autarken Batteriebetrieb ist der Raspberry Pi also ungeeignet, es sei denn, man nimmt in Kauf, sehr häufig laden zu müssen oder man implementiert ein aufwändiges Power-Management (etwa ein externes Wakeup durch einen Mikrocontroller).

In stationären Anwendungen ist der Stromverbrauch des Pi 4 ebenfalls zu bedenken: ca. 3 W im Leerlauf bedeutet Wärmeentwicklung und konstanten Energiebedarf. In gegebenen Bewässerungsprojekt kann der Pi nur sinnvoll eingesetzt werden, wenn er ständig an einem Netzteil angeschlossen ist oder man z. B. ein Solarpanel mit einem großen Akku nutzt. Das steigert aber die Komplexität. Der Pi Zero W wäre sparsamer, aber auch hier sprechen wir von ~0,6 W Dauerverbrauch, was z. B. für eine kleine Solaranlage noch tragbar wäre, aber deutlich mehr als ein paar μA in Sleep, wie es Microcontroller bieten.

Zusammenfassung: Energieeffizienz ist eine Domäne der Mikrocontroller. Der Ardui-

no ATmega4809 kann mit minimalem Strom auskommen (wenige μ A in Sleep) und ist damit am besten für langen Batteriebetrieb geeignet – allerdings nur ohne WiFi.

Der ESP32 benötigt im aktiven WLAN-Betrieb mehr Strom (\sim 50–100 mA), kann aber durch Schlafmodi energieoptimiert werden und ist für batteriegespeiste IoT-Geräte konzipiert.

Der Raspberry Pi hingegen hat einen viel höheren Grundverbrauch (Pi 4 ca. 0,6 A, Zero W ca. 0,12 A im Idle) und keine einfachen Sleep-Mechanismen – für Batteriebetrieb ist er die schlechteste Wahl. In der Anwendung (automatisierte Bewässerung) ist typischerweise ein Netzteil vorhanden; falls aber eine netzunabhängige Lösung gewünscht ist, wären Arduino oder ESP32 deutlich geeigneter. Der ESP32 bietet einen guten Kompromiss aus Netzwerkfähigkeit und vertretbarem Stromverbrauch, insbesondere wenn per Deep-Sleep gearbeitet wird.

4.9.7 Ökosystem, Entwicklungsumgebung und Community-Support

Arduino (ATmega4809): Die Arduino-Plattform genießt seit über einer Dekade eine riesige Community und eine einfach zu bedienende Entwicklungsumgebung (Arduino IDE). [110] Für den ATmega4809 wird die Programmierung typischerweise in C++ (Arduino Language) innerhalb der Arduino IDE erfolgen.

Die Arduino IDE abstrahiert viele Hardwaredetails, was einen leichten Einstieg ermöglicht. Es existiert sehr viele Bibliotheken für Sensoren, Aktoren und Module – viele davon ursprünglich für ATmega328 (Uno), aber kompatibel mit ATmega4809, da sich beide die AVR-Architektur teilen. Die Community hat unzählige Tutorials, Forenbeiträge und Beispielprojekte veröffentlicht. [110] Gerade für typische Aufgaben (z. B. Feuchtesensor auslesen, Relais ansteuern) ist es leicht fertige Codebeispiele zu finden. Auch wenn der ATmega4809 etwas neuer ist, gilt: Arduino steht für breite Community-Unterstützung.

Zudem ist das Ökosystem an Hardware groß – Sensor-Shields, Module etc., oft mit Beispielcode für Arduino. Für unseren Kontext heißt das, dass viele relevante Komponenten (Feuchtigkeitssensor, MQTT-Client als PubSubClient-Library, etc.) direkt verfügbar sind. Allerdings ist zu beachten, dass Netzwerk-Libraries (MQTT, WiFi) auf einem klassischen Arduino nur mit externer Hardware funktionieren. Hier würde man ggf. Bibliotheken nutzen, die die Arbeit an ein WiFi-Modul delegieren. Die Dokumentation auf arduino.cc und bei Microchip (Datenblatt) deckt technische Details ab, während die Maker-Community praktische Lösungen liefert.

Insgesamt ist Arduino hinsichtlich Entwicklungsfreundlichkeit top – speziell für Einsteiger. Im professionellen Umfeld kann die limitierte Debugging-Möglichkeit (häufig nur Serial-Print) ein Nachteil sein, aber die Einfachheit und Stabilität der Toolchain ist ein Vorteil.

ESP32-WROOM-32D: Der ESP32 hat in den letzten ~5 Jahren ebenfalls eine große Entwicklergemeinde aufgebaut. Es gibt zwei populäre Wege der Entwicklung: zum einen die Arduino-IDE-Unterstützung für ESP32, zum anderen das *Espressif IoT Development Framework (ESP-IDF)*. [110]

Ersteres erlaubt Arduino-Nutzern, den ESP32 ähnlich wie einen Arduino zu programmieren (mit `setup()`/`loop()` und Zugriff auf viele Arduino-kompatible Libraries). Dadurch profitiert man von der bestehenden Arduino-Bibliothekslandschaft (viele Sensorlibraries wurden schon für ESP32 portiert) und behält eine einfache IDE.

Für tiefergehende Projekte kann man auf das ESP-IDF wechseln, welches in C/C++ umfassende APIs für FreeRTOS, WLAN, BLE, Verschlüsselung etc. bietet. Die Lernkurve ist hier steiler, aber das Potenzial größer.

Community-Support: Es existiert eine aktive Entwickler-Community (Foren, GitHub, Reddit), sowie ausführliche offizielle Dokumentationen und Beispiele von Espressif. [110] Probleme werden häufig in Foren diskutiert – z. B. das offizielle esp32.com-Forum oder auf Arduino/StackExchange. Die Tool-Unterstützung ist moderner als bei AVR: Man kann mit dem IDF eine Debugger-Schnittstelle nutzen, und es gibt Plugins für VS Code (PlatformIO, Arduino-ESP32).

Zusätzlich gibt es alternative Umgebungen wie *MicroPython* und *Lua (NodeMCU)*, mit denen der ESP32 scriptbasiert programmiert werden kann – praktisch für schnelle Experimente. Hinsichtlich Bibliotheken: Für beinahe alle gängigen Sensoren und Protokolle gibt es ESP32-kompatible Libraries (sei es aus der Arduino-Welt oder eigens geschrieben). Spezialfeatures wie BLE, FreeRTOS, HTTPS werden in vielen Blogposts und Dokus behandelt, da ESP32 eben weit verbreitet im IoT ist. Zusammenfassend hat der ESP32 ein sehr lebendiges Ökosystem, das sowohl Maker (über Arduino-IDE) als auch professionelle Entwickler (über ESP-IDF) anspricht. [110]

Im Vergleich zum Arduino ist die Einstiegshürde leicht höher (mehr Konzepte wie WiFi-Stack, RTOS), aber die Community liefert viele Best Practices, Tutorials (z. B. „ESP32 plant watering“ Guides). Für dieses Projekt bedeutet das: Es gibt bereits zahlreiche Projekte und Beispiele für „ESP32 plant watering“ im Netz, an denen man sich orientieren kann.

Raspberry Pi (Linux): Der Raspberry Pi hat vielleicht die größte Community unter den drei – allerdings in etwas anderer Ausprägung. Als vollwertiger Linux-Computer stehen dem Entwickler eine Vielzahl an Programmiersprachen offen (Python ist besonders beliebt für IoT auf dem Pi, aber C/C++, Java, Node.js etc. sind alle möglich). [110]

Die Raspberry Pi Foundation stellt umfangreiche offizielle Dokumentationen bereit (z. B. zur GPIO-Programmierung mit Python `gpiozero` oder C `wiringPi`). [110] Zudem gibt es

offizielle Foren und Tutorials auf [raspberrypi.com](https://www.raspberrypi.com), ein offizielles Magazin und unzählige Community-Projekte. Die Entwicklung auf dem Pi kann direkt auf dem Gerät erfolgen (z. B. via SSH und einem Editor, oder mit IDEs wie Thonny für Python).

Die Hürde zum Start ist potentiell geringer, weil viele Leute mit Linux vertraut sind oder weil Pi als Lehrplattform konzipiert wurde – man findet z. B. Schritt-für-Schritt Anleitungen für einen Webserver auf Pi, für MQTT mit Python etc. Allerdings bedarf es auch Kenntnissen der Linux-Umgebung (Dateisystem, Dienste), was für rein embedded-orientierte Entwickler neu sein kann. Der Pi erlaubt, dank OS, komplexe Software einzubinden – z. B. kann man Node-RED oder Home Assistant auf dem Pi laufen lassen, was weit über Microcontroller-Code hinausgeht.

Der Community-Support ist exzellent, aber oft eher auf High-Level-Fragen ausgerichtet („Wie installiere ich MQTT Broker XY“ oder „Warum funktioniert meine Python-Library nicht“). [110] Ein Vorteil des Pi-Ökosystems: Viele Bibliotheken sind bereits vorhanden, z. B. in Python existiert für nahezu jeden Sensor eine Bibliothek, oft direkt installierbar über pip. Die Integration ins IoT-Ökosystem (MQTT, Cloud) wird durch vorhandene Software erleichtert. Kurz gesagt, der Raspberry Pi bietet maximale Flexibilität und Unmengen an Ressourcen/Support, erfordert aber die Bereitschaft, sich mit einem vollwertigen Computer auseinanderzusetzen (Updates, Linux-Kommandos, etc.).

Für dieses Projekt wären z. B. vorhandene MQTT-Client Libraries in Python oder C++ ein Plus, ebenso Web-Frameworks (Flask, etc.) um das Webinterface schnell zu bauen – all das ist auf dem Pi einfach nutzbar. Die Community ist so groß, dass sehr viel Teilproblem schon gelöst wurde.

Zusammenfassung: Arduino punktet mit einfacher IDE und gigantischer Einsteiger-Community – ideal für schnelle Prototypen und unkomplizierte Sensor/Aktor-Programmierung.

ESP32 hat sich ebenfalls ein starkes Ökosystem geschaffen, mit der Besonderheit, dass er von der Arduino-Welt und gleichzeitig von der professionellen Embedded-Welt unterstützt wird. Man profitiert von vielen vorhandenen IoT-spezifischen Beispielen und offiziellen Espressif-Ressourcen.

Raspberry Pi bietet eine vollständige Linux-Umgebung mit entsprechend breitem Software-Ökosystem; die Community ist riesig, aber man bewegt sich eher im Linux-/Software-Bereich als direkt auf Bare-Metal-Ebene.

Für dieses Projekt ist wichtig: Gute MQTT- und Web-Support – hier haben ESP32 und Pi beide reichlich vorhandene Libraries und Dokus. Arduino hat in diesem Bereich die wenigsten direkten Ressourcen (weil klassischer Arduino selten für Web/MQTT ohne zusätzliche Module eingesetzt wird), aber die generelle Arduino-Community würde dennoch

Hilfestellungen liefern.

Insgesamt ist der Support für ESP32 mittlerweile fast so einsteigerfreundlich wie Arduino, mit dem Vorteil, dass modernere IoT-Themen abgedeckt sind.

4.9.8 Kosten, Verfügbarkeit und Formfaktor

Hardwarekosten: Preislich liegen die drei Optionen recht unterschiedlich. Ein *ESP32-WROOM-32D Modul* (nur das Modul, ohne Dev-Board) kostet rund ~6 € im Einzelkauf. [119] Gängige Entwicklungsboards (NodeMCU-32, ESP32 DevKitC) bewegen sich etwa bei 8–15 € je nach Anbieter.

Ein *Arduino Nano Every* (ATmega4809) wird vom Hersteller Arduino für ca. 15 € angeboten (offizieller Store-Preis). [120] In Mengen oder als inoffizielle Nachbauten könnte er etwas günstiger sein, aber da der ATmega4809 noch nicht so verbreitet ist wie der klassische Uno, sind Clone-Preise begrenzt. Zum Vergleich: Ein traditioneller Arduino Uno Clone ist teils für <5 € zu bekommen, allerdings mit älterem ATmega328. Für das Projekt wäre vermutlich 1 Stück benötigt, daher nehmen wir ~10–15 € als Arduino-Kostenpunkt an.

Ein *Raspberry Pi 4 Model B* ist deutlich teurer: je nach RAM-Ausstattung zwischen ~40 € (1 GB) und ~80 € (8 GB). Ein mittleres 4 GB-Modell lag bei offiziellen Resellern um ~74 € (inkl. MwSt). [121] Hinzu kommen Accessoires (SD-Karte, evtl. Netzteil, Gehäuse).

Der *Raspberry Pi Zero W* hingegen ist konzipiert als Low-Cost-Variante. Sein offizieller Preis beträgt nur \$10 (~10 €). [122] Allerdings war bzw. ist die Verfügbarkeit limitiert (teils 1 Stück pro Kunde), weshalb in der Praxis Zero W oft für ~15–20 € gehandelt werden (inkl. Zubehör/Kits).

Insgesamt ist die ESP32-Lösung am günstigsten für das Gebotene – man bekommt MCU + WiFi für unter 10 €. Der Arduino liegt im ähnlichen Bereich (10–15 €), bietet aber kein WiFi, so dass eigentlich noch ein WiFi-Modul (z. B. 5 €) hinzu gerechnet werden müsste, womit Arduino-Lösung >15 € kommt. Der Raspberry Pi ist am teuersten, außer man nutzt einen Zero. Aber selbst dann benötigt man SD-Karte, was den effektiven Preis ebenfalls auf ~15 € hebt (Zero W ~10 €, SD ~5 €).

Verfügbarkeit: Die letzten Jahre (2021–2023) gab es teils weltweite Chip-Lieferengpässe. Espressif ESP32 Module waren davon weniger betroffen – sie waren meist gut erhältlich über Elektronikdistributoren oder aus asiatischen Quellen.

Die ATmega4809-Chips stammen von Microchip, die AVR-Lieferkette war relativ stabil. Nano Every Boards sind über Arduino, Mouser, etc. meist verfügbar.

Raspberry Pi hatte die gravierendsten Engpässe: Besonders Pi 4 und Zero W waren zeitweise kaum zu kaufen, nur zu stark überteuerten Preisen oder mit langen Wartezeiten.

Gegen Ende 2023 besserte sich die Lage laut Hersteller – man peilt 1 Million produzierte Einheiten pro Monat an, um den Bedarf zu decken. [123] Dennoch kann die Verfügbarkeit eines Raspberry Pi ein Risikofaktor sein, insbesondere wenn das Projekt in Serie gehen soll oder zeitnah umgesetzt werden muss. Zero W sind oft schnell vergriffen. In Deutschland kann man Raspberry Pis bei offiziellen Partnern (RaspberryPi Direct, Reichelt, etc.) kaufen – Stand Anfang 2025 ist eine gewisse Entspannung da, aber Pi 4 sind immer noch höherpreisig als früher.

ESP32-Boards dagegen sind zahlreich von verschiedenen Herstellern verfügbar (Espressif selbst, Adafruit, SparkFun, chinesische Marken), was die Versorgung sichert. Auch Arduino-Boards lassen sich über viele Distributoren beziehen, wobei offizielle Arduino-Produkte teils etwas teurer sind als Klone. Für das Einzelprojekt scheint das zwar vernachlässigbar aber falls später eine Stückzahlenproduktion erwägt werden soll, ist der ESP32 als Modul sehr attraktiv, weil er auch direkt auf eigene PCBs gelötet werden kann (Module im Großkauf teilweise <3 € pro Stück). Ein Raspberry Pi lässt sich nicht so einfach integrieren.

Formfaktor: Platz ist bei einem Sensorgerät oft wichtig. Der *ESP32-WROOM-32D* ist ein kleines SMD-Modul ($\sim 18 \times 25,5 \times 3$ mm) [111], d. h. sehr kompakt. Selbst auf einem Dev-Board bleibt er handlich (DevKitC etwa 50×25 mm).

Der *Arduino Nano Every* misst 45×18 mm [112] – wie ein breiter USB-Stick, sehr klein und leicht auf einem Steckbrett unterzubringen.

Ein *Raspberry Pi 4B* hingegen ist 85×56 [114],mm (plus 18 mm Höhe etwa) – also deutlich größer, in etwa Kreditkartengröße. Für ein Pflanzensor-Gerät wäre das deutlich zu groß.

Der *Pi Zero W* ist mit 65×30 [115],mm deutlich kleiner, aber immer noch flächenmäßig $\sim 4 \times$ größer als ein ESP32-Modul. Auch vom Gewicht sind Pis höher (eine Pi4 Platine ~ 50 g vs. ein ESP32-Modul wenige Gramm).

ESP32 oder Arduino viel einfacher an Pflanzentöpfen zu befestigen und damit hier die beste Wahl.

Zusammenfassung: Kosten/Große sprechen klar für den ESP32 (günstig und kompakt). Der Arduino Nano Every ist ebenfalls klein und im unteren Preisbereich, allerdings ohne inbegriffenes WiFi müsste man Zusatzkosten rechnen. Raspberry Pis sind teurer (Pi4) oder knapp verfügbar (Zero W) und nehmen mehr Platz ein. Für ein skaliertes IoT-Projekt wäre der Unterschied noch deutlicher: Der ESP32 lässt sich für geringe Kosten in Massen beschaffen und integrieren, während Raspberry Pis pro Stück ein Mehrfaches kosten und schwerer in eigene Hardware integrierbar sind. Für den Prototypenbau kann man vorhandene Raspberry Pis nutzen, aber im finalen Einsatz ist das ESP32-Modul die öko-

nomischere und handlichere Lösung.

4.9.9 Eignung der Plattformen für spezifische Projektanforderungen

Zum Abschluss betrachten wird betrachtet, wie gut jede Plattform die konkreten Anforderungen der automatisierten Bewässerungssteuerung erfüllt:

Zyklische Sensorabfrage: Alle drei Plattformen können periodisch Sensordaten erfassen. Arduino und ESP32 haben hier einen Vorteil, da sie *Analogeingänge* besitzen (der Bodenfeuchtesensor liefert typischerweise eine analoge Spannung). Der Arduino kann mittels Timer-Interrupt z. B. alle X Minuten die Feuchte messen – seine 10-Bit ADC-Auflösung ist meist ausreichend. [112] Der ESP32 kann dank 12-Bit ADC und ggf. Hardware-Timer ebenfalls präzise in Intervallen messen; er könnte sogar im Deep Sleep vom ULP geweckt werden, um Strom zu sparen. [111] Der Raspberry Pi hat *kein analoges Input* – für die Feuchtesensorabfrage bräuchte man entweder einen *ADC-Chip* (z. B. ADS1115 via I²C) oder man verwendet digitale Sensoren. Das ist ein Mehraufwand und eine potenzielle Fehlerquelle. [114]

Zeitlich schaffen alle Plattformen z. B. eine Abfrage pro Minute ohne Probleme; der Arduino ist hierbei am nächsten an Echtzeitfähigkeit (Timer läuft unabhängig vom Hauptprogramm), während Linux beim Pi im worst-case ein paar hundert Millisekunden Off-Timing haben kann – für Feuchtemessungen aber irrelevant. Fazit: Arduino und ESP32 sind sehr gut geeignet für zyklische Abfragen (Analog direkt lesbar); bei einem Pi muss man die fehlende ADC-Fähigkeit kompensieren.

Aktorsteuerung (Pumpe): Die Pumpe (eine 6 V-Wasserpumpe) wird über einen Transistor oder ein Relais angesteuert. Alle drei Plattformen haben *GPIO-Ausgänge*, die dafür genutzt werden können. Der Arduino kann einen digitalen Pin setzen, um z. B. ein Transistor-Gate anzusteuern – mit bis zu 20 mA Ausgangsstrom pro Pin [112] reicht das für die meisten Treiber. Der ESP32 liefert 3,3 V Logik, was aber für MOSFET-Gates oder Relaismodule (die oft 3,3 V-tauglich sind) genügt. [112]

Der Raspberry Pi hat ebenfalls 3,3 V GPIO [114] auch hier ist die Ansteuerung eines Relaismoduls gängig (Achtung: Pi-GPIO liefern max ~16 mA, i. d. R. nimmt man 2–3 mA ins Optokoppler-Relais – passt).

Bei zeitkritischer Steuerung (z. B. PWM modulieren, um Pumpenleistung zu regeln) sind Mikrocontroller besser – Arduino und ESP32 sind bei Hardware-PWM sehr präzise. Der Pi könnte PWM per Software oder `pigpio` erzeugen, das ist weniger präzise aber machbar. Im gegebenen Fall wird die Pumpe vermutlich einfach EIN/AUS geschaltet, was trivial

für alle Optionen ist.

Fazit: Alle können die Pumpe ansteuern; keine Plattform hat hier große Vorteile, außer dass Arduino/ESP32 etwas „direkter“ und deterministischer steuern.

Netzwerkkommunikation inkl. MQTT: Hier zeigen sich deutliche Unterschiede. Der Arduino (ohne Zusatzmodul) kann keine MQTT-Kommunikation durchführen, da ihm WLAN/Netzwerk fehlt. Selbst mit einem WiFi-Shield müsste man einen großen Teil der Ressourcen dafür aufwenden und hätte möglicherweise Schwierigkeiten, ein SSL-verschlüsseltes MQTT zuverlässig umzusetzen (RAM- und CPU-Limit).[112]

ESP32 und Raspberry Pi hingegen sind von Haus aus netzwerkfähig und MQTT-tauglich. Der ESP32 kann mittels MQTT-Client-Bibliothek die Sensordaten publizieren und Befehle abonnieren – dank eingebautem TCP/IP-Stack und ausreichend RAM (520 KB) ist auch eine verschlüsselte MQTT-Verbindung möglich. [111]

Der Raspberry Pi kann dank Linux jede MQTT-Client-Software nutzen; z. B. wäre ein Python-Skript mit `paho-mqtt` sehr unkompliziert. Auch Lasttests (viele MQTT-Nachrichten) verkraften beide gut – der Pi hätte hier die höchste Reserve. [114]

Für diese Anwendung (geringe Datenrate) sind beide problemlos geeignet. Da MQTT typischerweise eine konstante TCP-Verbindung zum Broker hält, muss das Gerät dauerhaft netzwerkverbunden sein oder sich periodisch verbinden. Der ESP32 kann das, frisst dann aber permanent ~80 mA (was bei Netzstrom ok ist, bei Batterie schlecht). Der Pi kann das auch, mit dem bekannten Stromhunger. In Bezug auf Implementierungsaufwand ist der ESP32 etwas spezieller (man programmiert es in Code), während man auf dem Pi auch ein fertiges Tool nehmen kann.

Fazit: ESP32 und Raspberry Pi erfüllen MQTT-Kommunikation voll, der Arduino ist hier weit unterlegen (ohne Zusatzhardware quasi ungeeignet).

Authentifizierung: Arduino: Mit einem 8-Bit MCU sind moderne Authentifizierungsmethoden kaum handhabbar. Einfache Benutzer/Passwort Kombinationen können noch gespeichern und gesenden werden, aber z. B. TLS-Zertifikatsprüfung für MQTT übersteigt die Möglichkeiten.

Der ESP32 dagegen hat *Hardware-Krypto und TLS 1.2* Support – er kann sich per WPA2 sicher ins WLAN einloggen und z. B. per Client-Zertifikat an einem MQTT-Broker anmelden. Auch ein Webinterface kann er mit Passwortschutz versehen (Basic Auth oder Form-Login).

Der Raspberry Pi profitiert von Linux – man kann OpenSSL etc. verwenden, also jede erdenkliche Authentifizierungsmethode (z. B. OAuth für Web, Zertifikate, Tokens) implementieren. Für dieses Projekt ist insbesondere wichtig: all drei können *WPA2 WLAN Auth* – Arduino via externem WiFi-Modul und *MQTT-Broker Auth* (Benutzer/Pass-

wort ist trivial auf ESP32 und Pi, Zertifikat-basierte Auth geht auf ESP32 auch, auf Pi natürlich ebenso). Zudem könnte man einen *Lokalen Web-Login* benötigen, falls das Konfigurations-Webinterface geschützt werden soll – das wäre auf dem Pi am einfachsten (man könnte auch Apache+PHP laufen lassen und ein richtiges Login-Formular nutzen). Auf dem ESP32 muss so etwas selbst gecoden werden (z. B. Session Management).

Fazit: ESP32 und Raspberry Pi sind für Authentifizierungs- und Sicherheitsanforderungen deutlich besser gerüstet (ESP32 durch Hardware-TLS, Pi durch pure Rechenleistung und Linux-Security), Arduino ist hier kaum geeignet, wenn es über simple Klartext-Logins hinausgeht.

Webinterface zur Erstkonfiguration: Ein wesentliches Merkmal moderner IoT-Geräte ist die Fähigkeit zur netzwerkbasierten Erstkonfiguration, etwa durch ein lokales Webinterface oder die Bereitstellung eines Captive Portals im Access-Point-Modus. Dieses Webinterface dient typischerweise zur Eingabe von WLAN-Zugangsdaten oder sonstigen benutzerspezifischen Einstellungen.

Die Arduino-Plattform in ihrer klassischen Ausprägung bietet keine native Netzwerkanbindung, sodass ein solches Interface ohne Zusatzhardware grundsätzlich nicht realisierbar ist. Selbst bei Verwendung eines Ethernet-Shields und eines ATmega4809-basierten Arduino-Boards wäre die Bereitstellung eines Webinterfaces stark eingeschränkt. Aufgrund des sehr begrenzten Arbeitsspeichers von lediglich 6 KB könnten höchstens wenige Hundert Byte an HTML ausgeliefert werden, was die Darstellung moderner, benutzerfreundlicher Weboberflächen praktisch ausschließt.

Im Gegensatz dazu verfügt der ESP32 über integriertes WLAN sowie Bluetooth und unterstützt sowohl den Access-Point- als auch den Station-Modus. Dadurch ist es möglich, dass das Gerät im Auslieferungszustand einen eigenen WLAN-Zugangspunkt eröffnet, über den ein Konfigurationsinterface aufgerufen werden kann. In der Praxis erfolgt dies häufig über ein sogenanntes Captive Portal, wie es etwa in der ESP32-Community anhand der weit verbreiteten *WifiManager*-Bibliothek umgesetzt wird. Der Nutzer verbindet sich mit dem vom Gerät erzeugten WLAN und erhält Zugriff auf eine lokale Konfigurationsseite, über die sich beispielsweise SSID und Passwort des Heimnetzwerks eintragen lassen. Nach Speicherung der Daten, etwa im nichtflüchtigen NVS-Speicher, stellt der ESP32 automatisch die Verbindung zum Zielnetzwerk her.

Durch den vorhandenen Speicher von 520 KB SRAM ist der ESP32 in der Lage, auch komplexere HTML-Seiten zu hosten. Dennoch empfiehlt sich ein ressourcenschonendes Design, etwa durch den Verzicht auf große Grafiken oder aufwändige clientseitige Skripte. Neben der initialen Konfiguration kann das Webinterface darüber hinaus im laufenden Betrieb zur Anzeige von Sensorwerten oder zur Änderung von Parametern verwendet wer-

den.

Auch der Raspberry Pi ist grundsätzlich in der Lage, ein solches Interface bereitzustellen. Als vollwertiger Linux-Computer mit zahlreichen Softwarebibliotheken besteht hier die Möglichkeit, einen umfangreichen Webserver wie Apache oder Nginx zu betreiben oder auf Frameworks wie Flask oder Django zurückzugreifen. Die Funktionalität und Gestaltungsfreiheit sind dabei nahezu unbegrenzt. Der Aufwand zur Implementierung ist jedoch höher, da zusätzliche Konfigurationsmaßnahmen erforderlich sind. Der Betrieb im Access-Point-Modus ist möglich, bedarf aber systemseitiger Anpassungen. Zudem ist der Energieverbrauch deutlich höher als bei Mikrocontroller-basierten Lösungen, was insbesondere bei autarken, batteriebetriebenen Geräten nachteilig ist. In der Praxis erfolgt die Konfiguration solcher Systeme daher häufig per SSH oder über eine angeschlossene Benutzeroberfläche (HDMI, Tastatur), was die Benutzerfreundlichkeit im Vergleich zu einem eingebetteten Webinterface reduziert.

Insgesamt ist der ESP32 für den Betrieb eines eingebetteten Webinterfaces besonders gut geeignet. Er vereint kompakte Bauform, ausreichende Ressourcen, integrierte WLAN-Funktionalität und zahlreiche etablierte Softwarelösungen für typische Anwendungsfälle. Der Raspberry Pi bietet zwar theoretisch die umfangreichsten Möglichkeiten zur Webentwicklung, bringt jedoch höheren Energieverbrauch, komplexere Konfiguration und größeren Wartungsaufwand mit sich. Die Arduino-Plattform hingegen ist für diesen Anwendungsfall aufgrund fehlender Ressourcen und Schnittstellen faktisch ungeeignet.

Task-Parallelisierung mit Prioritätssteuerung: Ein wesentliches Kriterium zur Beurteilung der Eignung eines Mikrocontrollers oder Mikrocomputers in eingebetteten Systemen ist die Fähigkeit zur parallelen Ausführung mehrerer Aufgaben sowie zur gezielten Steuerung ihrer Ausführungsriorität. Diese Eigenschaft ist insbesondere bei Anwendungen relevant, in denen neben zyklischen Abfragen auch asynchrone Prozesse, wie etwa Netzwerkkommunikation, zeitlich korrekt behandelt werden müssen.

Die Arduino-Plattform auf Basis des ATmega4809 unterstützt keine echte Parallelverarbeitung. Das zugrunde liegende Architekturmodell ist rein sequenziell, d. h. sämtliche Aufgaben werden innerhalb einer Hauptschleife nacheinander abgearbeitet. Zwar können Interrupts genutzt werden, um zeitkritische Ereignisse wie das Auslesen eines Sensors unabhängig vom Hauptprogramm zu erfassen, eine strukturierte und dynamisch steuerbare Mehrtask-Verarbeitung ist jedoch nur durch kooperatives, manuelles Codieren umsetzbar. Eine echte Priorisierung konkurrierender Prozesse ist damit nicht möglich.

Im Gegensatz dazu bietet der ESP32 durch das integrierte FreeRTOS eine umfassende Unterstützung für präemptives Multitasking mit Prioritätssteuerung. Mehrere Tasks lassen sich explizit definieren und mit unterschiedlichen Prioritäten ausstatten. Dadurch ist

es beispielsweise möglich, die periodische Sensordatenerfassung in einem hoch priorisierten Task zu realisieren, während weniger kritische Vorgänge – etwa das Versenden von MQTT-Nachrichten – in Hintergrundtasks mit niedrigerer Priorität abgewickelt werden. Durch das duale Prozessordesign kann zudem eine Trennung von zeitkritischen Steuerungsaufgaben und netzwerkbezogenen Prozessen auf unterschiedlichen Kernen erfolgen. Dies ermöglicht eine deterministische und reaktionsschnelle Abarbeitung sicherheitsrelevanter oder latenzkritischer Funktionen selbst bei gleichzeitiger Auslastung durch andere Subsysteme.

Auch der Raspberry Pi erlaubt unter seinem Linux-basierten Betriebssystem grundsätzlich eine Priorisierung durch Mechanismen wie `nice`, `SCHED_RR` oder `SCHED_FIFO`. Allerdings ist der Standard-Linux-Kernel nicht für harte Echtzeitanforderungen ausgelegt. Bestimmte Systemprozesse und Interrupts können nicht verdrängt werden, wodurch keine garantierten Latenzen erreicht werden können. In der Praxis kann eine gewisse Bevorzugung einzelner Threads erreicht werden, etwa durch die Zuweisung hoher Prioritäten oder Prozessaffinitäten. Dennoch verbleibt eine inhärente Unschärfe in der zeitlichen Steuerung, insbesondere bei gleichzeitiger Nutzung höherer Programmiersprachen wie Python, deren Laufzeitumgebungen selbst Speicherbereinigung oder Thread-Scheduling beeinflussen können. Für Aufgaben mit moderater Echtzeitanforderung ist dies tolerierbar, in strikt deterministischen Steuerungsszenarien jedoch kritisch zu bewerten.

Insgesamt zeigt sich, dass Mikrocontroller mit integriertem RTOS, wie der ESP32, klare Vorteile hinsichtlich kontrollierter Task-Parallelisierung und Prioritätssteuerung bieten. Die Architektur erlaubt eine explizite Zuweisung von Ausführungsrioritäten und eine deterministische Verarbeitung auch bei konkurrierender Last. Der Raspberry Pi ist in der Lage, mehrere Prozesse gleichzeitig zu bearbeiten, bietet jedoch keine zuverlässige Echtzeitausführung. Der Arduino ist in diesem Kontext auf einfache sequenzielle Logik beschränkt, wodurch komplexere parallele Anwendungen ohne erhebliche Eingriffe in die Systemarchitektur nicht realisierbar sind.

4.9.10 Entscheidung: Warum **ESP32-WROOM-32D**?

Nach der detaillierten Gegenüberstellung wird deutlich, dass der **ESP32-WROOM-32D** für das IoT-Bewässerungsprojekt die beste Wahl ist. Der ESP32 vereint die wichtigsten Anforderungen in einer Plattform und bietet gegenüber Arduino und Raspberry Pi konkrete Vorteile:

- **Integriertes WLAN/Bluetooth:** Nur der ESP32 und der Pi haben WiFi on-board. Im Gegensatz zum Pi benötigt der ESP32 aber keine zusätzliche Infrastruktur (kein OS, keine SD), um eine einfache MQTT-Verbindung herzustellen. Gegenüber dem Arduino

(der gar kein WiFi hat) ist dies ein entscheidender Vorteil – es entfällt zusätzliche Hardware und komplexes Zusammenspiel externer Module.

- **IoT-Optimierte Architektur:** Der ESP32 ist speziell für IoT konzipiert. Er bietet ausreichend Leistung (240 MHz Dual-Core) und Speicher (520 KB RAM) für Netzwerkprotokolle, ohne die Energieaufnahme eines Raspberry Pi. Mit FreeRTOS und LwIP ist der Netzwerkstack hochintegriert, was zuverlässige MQTT-Kommunikation und sogar sichere TLS-Verbindungen ermöglicht – etwas, das der Arduino nicht leisten kann. Gleichzeitig bleibt er energieeffizient genug, um bei Bedarf im Feld mit Akku/Solar betrieben zu werden, was beim Pi praktisch ausgeschlossen ist.
- **Analoge und digitale Schnittstellen in Fülle:** Für die Sensoranbindung (Feuchtigkeitssensor analog, evtl. weitere Sensoren) ist der ESP32 bestens ausgestattet – 18 *Analog-Inputs* bieten viel Spielraum, und dank 12-Bit ADC können Sensordaten präziser erfasst werden als mit dem 10-Bit Arduino. Gegenüber dem Raspberry Pi eliminiert der ESP32 das Problem des fehlenden Analog-Eingangs vollständig. Auch weitere I/Os (Pumpensteuerung per GPIO, ggf. I²C-Sensoren) sind reichlich vorhanden. Das heißt, alle Hardware-Komponenten des Projekts lassen sich direkt an den ESP32 anschließen, ohne zusätzliche Wandler oder Expander.
- **Multitasking und Zuverlässigkeit:** Mit dem ESP32 kann die Software so gestaltet werden, dass parallele Abläufe (Sensor lesen, MQTT publizieren, Webserver bedienen) robust funktionieren. *Echtzeitkritische Abläufe* (z. B. Abschalten der Pumpe bei bestimmten Bedingungen) können mit hoher Priorität laufen, was Betriebssicherheit gibt. Ein Raspberry Pi könnte zwar auch parallel arbeiten, birgt aber Risiken durch sein komplexes OS (ein abgestürzter Dienst könnte das ganze System beeinträchtigen, während beim ESP32 die Firmware aus einem Guss ist). Ein Arduino wiederum könnte mangels echter Parallelität ins Straucheln kommen, wenn z. B. eine Netzoperation länger dauert – beim ESP32 würde diese in einem getrennten Task laufen und die Sensorlogik nicht blockieren.
- **Einfaches Webinterface und Konfiguration:** Der ESP32 erlaubt es, benutzerfreundlich eine Konfigurations-Webseite zu hosten, etwa um initial WLAN-Zugangsdaten einzugeben. Diese Fähigkeit hat er mit dem Pi gemein, aber beim ESP32 lässt sich ein solches Portal sehr schlank implementieren (bereits viele vorhandene Arduino-Libraries dafür). Der Nutzerkomfort (AP-Mode + Handy-Webseite) ist ein großer Pluspunkt, um das Gerät in Betrieb zu nehmen. Arduino könnte das gar nicht, Pi kann es nur mit erheblich größerem Softwarepaket. Zudem kann der ESP32 bei Bedarf auch über BLE konfigurieren (denkbar wäre eine Handy-App über Bluetooth Low Energy). Diese

Flexibilität in der Erstinstallation ist praktisch.

- **Sicherheit:** In Sachen Authentifizierung und Verschlüsselung bietet der ESP32 eine gute Balance – er kann z. B. MQTT über TLS mit Client-Zertifikat fahren, was ein Arduino völlig überfordern würde. Der Pi kann das zwar auch, aber beim ESP32 ist es auf Embedded-Ebene gelöst und erfordert weniger Wartung (z. B. kein Linux, das gepatcht werden muss). Für ein IoT-Gerät, das eventuell über das Internet kommuniziert, sind Security-Features wichtig – der ESP32 hat sie eingebaut (Hardware-RNG, AES, RSA, Secure Boot Option, etc.). Das macht ihn zukunftssicherer als einen Arduino-Ansatz, der solche Dinge nicht berücksichtigt.
- **Kosten und Größe:** Ein ESP32-Modul ist äußerst kostengünstig im Vergleich zu einem Raspberry Pi – man kann mit ~5–10 € pro Gerät rechnen, wohingegen ein Pi ein Mehrfaches kostet. Für ein System, das vielleicht in mehreren Pflanzen eingesetzt werden soll, ist das ein erheblicher Vorteil. Auch die Kompaktheit spricht für den ESP32: Das gesamte Steuergerät kann klein genug gebaut werden, um unauffällig am Blumentopf Platz zu finden. Ein Pi4 bräuchte ein größeres Gehäuse und wäre überdimensioniert.
- **Community und vorhandene Projekte:** Speziell im Bereich „Plant watering IoT“ gibt es bereits viele Umsetzungen mit dem ESP32. Das heißt, man kann von bewährten Lösungen profitieren – sei es Code für kapazitive Feuchtesensoren, Sleep-Wake-Zyklen oder MQTT-Integration. Arduino-Projekte gibt es auch viele, aber meist ohne Netzwerkanbindung. Raspberry Pi-Projekte gibt es ebenfalls, doch oft wird dort ein vollwertiger Pi wegen einfacherem Prototyping genutzt, obwohl ein Mikrocontroller besser geeignet wäre. Der Trend in der Maker- und IoT-Community für solche Einsatzzwecke geht klar zum ESP32, was bedeutet: Unterstützung und Weiterentwicklung sind auf längere Sicht garantiert.

Konkrete Vorteile des ESP32 gegenüber Arduino Nano Every: Der ESP32 bietet Größenordnungen mehr Rechenleistung und Speicher, was ihn fähig macht, die Netzwerkkommunikation (MQTT, TLS) überhaupt erst zu stemmen, während der Arduino das nicht könnte. Er vereint die Funktionen von „Arduino + WiFi-Shield + evtl. Echtzeituhr + mehr ADC + Bluetooth“ in *einem* Modul. Damit reduziert sich die Komplexität der Hardware drastisch. Zudem ist der ESP32 durch 32-Bit-Architektur zukunftssicherer – komplexere Sensoralgorithmen oder Berechnungen (z. B. Mittelwertbildung, Signalfilter) kann er ohne Mühe in Echtzeit erledigen, wo der ATmega an seine Grenzen käme.

Konkrete Vorteile des ESP32 gegenüber Raspberry Pi: Der ESP32 ist viel stromsparender, kann also notfalls autark betrieben werden oder zumindest dauerhaft ohne

nennenswerte Energiekosten laufen. Er ist einfacher aufgebaut, was die Robustheit erhöht. Es gibt keine SD-Karte, kein Betriebssystem, das ausfallen könnte. Damit ist die Wahrscheinlichkeit, dass das Gerät monatelang zuverlässig arbeitet, höher (ein Pi könnte z. B. nach einem Stromausfall Probleme haben, während der ESP32 sofort wieder startet und seine Aufgabe fortsetzt). Außerdem eliminiert der ESP32 unnötige Features: Ein Linux-Pi kann zwar mehr (Grafik, USB, etc.), aber das wird hier nicht gebraucht. Dadurch vermeidet man Angriffsfläche (z. B. muss man keinen Linux-Zugang absichern) und verkürzt die Entwicklungszeit (kein OS-Management, sondern direkte Firmware-Programmierung). Und natürlich: Kostenfaktor – ein ESP32-Lösungsaufbau kostet nur einen Bruchteil eines Pi 4, was insbesondere bei mehreren Einheiten ins Gewicht fällt.

Zusammenfassend erfüllt der ESP32-WROOM-32D alle Anforderungen des Projekts mit Bravour, wo Arduino und Raspberry Pi jeweils Schwächen zeigen:

- Er kann **Sensoren zyklisch auslesen** (Analogeingang vorhanden, Timer möglich) und die **Pumpe steuern** (GPIO/PWM) – ähnlich gut wie ein Arduino in der low-level Steuerung.
- Er bietet **Netzwerkkommunikation (WLAN, MQTT)** auf Embedded-Niveau – vergleichbar mit dem Pi, aber ohne dessen Overhead.
- Er unterstützt **Auth-Verfahren und Web-Konfiguration**, was Arduino nicht kann, und bei Pi mit größerem Aufwand verbunden ist.
- Er ermöglicht **Parallelisierung (FreeRTOS)**, und vermeidet die Nicht-Determinismen des Linux beim Pi.
- Er ist **kostengünstig, kompakt und energiesparend**, was ihn für praktische, ggf. auch batteriebetriebene IoT-Geräte prädestiniert.

Die Entscheidung zugunsten des **ESP32-WROOM-32D** ist daher klar: Diese Plattform liefert die beste Kombination aus Funktionalität, Leistung und Effizienz für eine automatisierte Pflanzenbewässerung im IoT-Kontext. Insbesondere die integrierte Netzwerk-anbindung und das Echtzeit-Betriebssystem machen den ESP32 zur optimalen Lösung, um Sensorik, Aktorik und Cloud-Konnektivität zuverlässig in einem Gerät zu vereinen. Arduino würde an den IoT-Anforderungen scheitern, und Raspberry Pi wäre überdimensioniert und weniger robust. Der ESP32 bietet modernes IoT auf Mikrocontroller-Basis – genau das, was für das Projekt benötigt wird.

4.10 Systemtechnologische Entscheidungen für das ESP32-basierte Bewässerungssystem

4.10.1 Entwicklungsframework

Die Wahl des passenden Entwicklungsframeworks ist entscheidend für die Stabilität, Wartbarkeit und Echtzeitfähigkeit eingebetteter Systeme. Im Kontext dieses Projekts, das auf dem ESP32-WROOM-32D basiert, wurden mehrere Optionen verglichen: das native ESP-IDF-Framework (C/C++), Arduino Core, MicroPython und PlatformIO. Die folgende Tabelle fasst die relevanten Unterschiede basierend auf den Quellen [124]–[127]. zusammen:

Tabelle 4.1: Vergleich von Entwicklungsframeworks für den ESP32

Kriterium	ESP-IDF (C/C++)	Arduino Core	MicroPython	PlatformIO
Performance	Sehr hoch	Mittel	Niedrig	Abhängig vom Backend
Ressourcenverbrauch	Sehr gering	Höher als ESP-IDF	Hoch	Frameworkabhängig
Echtzeitfähigkeit	Ja (nativ mit FreeRTOS)	Begrenzt (abstrahiert)	Nein	Ja, wenn ESP-IDF genutzt
Debugging	JTAG, GDB	Serielle Ausgabe	Print-Debugging	Sehr gut (VSCode, GDB)
Bibliotheksunterstützung	Hoch, technisch fundiert	Sehr hoch (Community)	Begrenzt	Sehr hoch (kombiniert)
Community und Support	Aktiv, technisch orientiert	Sehr groß, einsteigerfreundlich	Wächst, aber kleiner	Sehr aktiv
Komplexität	Hoch	Niedrig	Niedrig	Mittel

Die Tabelle zeigt, dass das ESP-IDF-Framework hinsichtlich Performance, Ressourcenverbrauch und Echtzeitfähigkeit den größten Vorteil bietet. Basierend auf diesen Kriterien fällt die Entscheidung auf **ESP-IDF mit FreeRTOS**. Die native Integration von FreeRTOS ermöglicht eine deterministische Taskplanung und Echtzeitverarbeitung. Gleichzeitig bietet das Framework direkten Zugriff auf hardwarenahe APIs, was für die effiziente Umsetzung von Sensor- und Aktorlogik von Vorteil ist. Die Entscheidung basiert auf technischer Dokumentation von Espressif Systems (vgl.[124]) sowie der offiziellen FreeRTOS-Referenz (vgl.[128]).

4.10.2 Dual-Core-Architektur und Taskmodell

Der ESP32-WROOM-32D basiert auf einer Xtensa LX6 Dual-Core Architektur, die echte Parallelverarbeitung durch zwei unabhängige Kerne erlaubt. Die Aufgabenverteilung erfolgt über FreeRTOS mittels `xTaskCreatePinnedToCore()`, womit Sensor-, Netzwerk- und Steuerungsfunktionen klar voneinander getrennt werden können. Das folgende Konzept wurde umgesetzt:

- Sensorik und Aktorik werden auf einem Kern als priorisierte Tasks verarbeitet.
- Netzwerkkommunikation und MQTT-Client laufen auf dem zweiten Kern.
- Tasks kommunizieren über Queues und Event-Groups.

Dieses Vorgehen verbessert die Reaktionszeit des Systems und stellt sicher, dass kritische Vorgänge (wie die Bewässerung) nicht durch Netzwerkprozesse blockiert werden. Gegenüber Single-Core-Systemen wie dem Atmega328 (Arduino Uno) stellt dies einen erheblichen Fortschritt hinsichtlich Performance und Modularität dar (vgl.[129]).

4.10.3 Sensorik

Im Rahmen des Projekts werden drei Sensoren zur Umweltdatenerfassung verbaut: ein DHT11 (Temperatur/Feuchte), ein BH1750 (Lichtstärke) sowie ein analoger resistiver Bodenfeuchtesensor mit LM393-Komparator. Die Auswahl folgt einer Bewertung nach Genauigkeit, Stromverbrauch, Robustheit und Schnittstellenkompatibilität.

Temperatur- und Luftfeuchtesensoren

Die Daten in Tabelle 4.2 basieren auf Herstellerdatenblättern und technischen Vergleichsstudien (vgl.[130]–[132]).

Tabelle 4.2: Vergleich von Temperatur- und Luftfeuchtesensoren

Sensor	Temp.-Genauigkeit	rF-Genauigkeit	Temp.-Bereich	rF-Bereich	Preis	Schnittstelle
DHT11	$\pm 2^\circ\text{C}$	$\pm 5\%$	0–50	20–80 %	Sehr günstig	1-Wire
DHT22	$\pm 0.5^\circ\text{C}$	$\pm 2\%$	-40–80	0–100 %	Mittel	1-Wire
BME280	$\pm 0.5^\circ\text{C}$	$\pm 3\%$	-40–85	0–100 %	Mittel–hoch	I ² C/SPI
SHT31	$\pm 0.3^\circ\text{C}$	$\pm 2\%$	-40–125	0–100 %	Hoch	I ² C

Der DHT11 wurde trotz moderater Genauigkeit und eingeschränktem Temperaturbereich ausgewählt, da er eine einfache Einbindung und sehr niedrige Stromaufnahme (Standby: 60 µA, Messbetrieb: 0.3mA) bei 3.3–5V ermöglicht [132]. Alternativen wie der DHT22 oder BME280 bieten zwar bessere Messwerte, erfordern jedoch höheren Energieaufwand bzw. komplexere Softwareintegration[130], [131].

Vergleich von Lichtsensoren

Die Spezifikationen stammen aus den jeweiligen Datenblättern und Modulbeschreibungen, insbesondere für den eingesetzten BH1750 [133].

Tabelle 4.3: Vergleich von Lichtsensoren

Sensor	Messbereich (Lux)	Auflösung	Schnittstelle	Bemerkung
BH1750	1–65.000	1 Lux	I ² C	Einfach, direkt kalibriert
TSL2561	0.1–40.000	0.1 Lux	I ² C	Empfindlich, komplexer
VEML7700	0.003–120.000	bis 0.005 Lux	I ² C	Sehr empfindlich, großer Bereich

Der BH1750 überzeugte durch seine hohe Betriebssicherheit, direkte Lux-Ausgabe ohne Kalibrieraufwand sowie die kompakte Integration über I²C. Laut Datenblatt liegt sein Stromverbrauch bei lediglich 0.12mA im aktiven Zustand bei 5V Betriebsspannung. Zudem bietet der Sensor bei sehr einfacher I²C-Anbindung einen für das Projekt ausreichenden Messbereich und eine hohe Zuverlässigkeit. Gegenüber dem VEML7700 fallen hier insbesondere die niedrigeren Kosten und die einfache Library-Integration ins Gewicht (vgl.[133]).

Vergleich von Bodenfeuchtesensoren

Die Vergleichswerte basieren auf allgemeinen Moduldokumentationen sowie dem TI-Datenblatt des LM393 Komparators (vgl.[134], [135]).

Tabelle 4.4: Vergleich von Bodenfeuchtesensoren

Typ	Messprinzip	Lebensdauer	Genauigkeit	Störanfälligkeit	Schnittstelle	Preis
Resistiv (LM393)	Leitfähigkeit	Gering	Niedrig	Hoch (Korrosion, Salze)	Analog/Digital	<1 €
Kapazitiv	Dielektrische Konstante	Hoch	Mittel	Gering	Analog	5 €
Digital (Chirp)	Kapazitiv + Audio	Hoch	Hoch	Gering	I ² C	>8 €

Trotz bekannter Nachteile (Korrosion, Interferenzen) wird ein resistiver Bodenfeuchtesensor, der den LM393-Komparator nutzt, gewählt. Dieser realisiert bei nur 200 µA Versorgung und schneller Schaltzeit (typ. 1 µs) eine einfache Feuchteerkennung (Vgl.[135]). Die Entscheidung basiert auf der Verfügbarkeit und den geringen Kosten für einen Proof-of-Concept (vgl.[134]).

4.10.4 Aktorik und Energieversorgung

Die folgende Tabelle basiert auf den technischen Daten des mechanischen Relais SRD-05VDC-SL-C, ergänzt durch Informationen zu Solid-State-Relais aus dem Opto 22-Datenblatt sowie Spezifikationen von MOSFET-Treibern, insbesondere des LM5112 von Texas Instruments (vgl.[136]–[138]).

Tabelle 4.5: Vergleich von Schaltkonzepten zur Aktorsteuerung

Komponente	Galv. Trennung	Schaltlast	Schaltfrequenz	Komplexität	Bemerkung
Mechanisches Relais	Ja	Hoch (induktiv möglich)	Niedrig	Niedrig	Bewährt, robust, laut
Solid-State-Relais	Ja	Begrenzt (je nach Typ)	Hoch	Mittel	Leise, verschleißfrei
MOSFET-Treiber	Nein	Sehr gut für PWM	Sehr hoch	Hoch	Effizient, aber nicht isoliert

Zur Steuerung der Wasserzufuhr wird sich dazu entschieden, eine kleine Tauchpumpe über ein SRD-05VDC-SL-C Relaismodul anzusteuern. Die Wahl eines mechanischen Relais begründet sich durch dessen galvanische Trennung und einfache Handhabung: die Steuerlogik wird über ein Netzteil gespeist, die Pumpe über eine Batterie. Diese Trennung erhöht die Betriebssicherheit und vermeidet Störeinflüsse. Für Systeme mit häufigen Schaltzyklen könnten SSRs oder MOSFETs in Betracht gezogen werden. Die eingesetzte Pumpe erlaubt unabhängigen Betrieb ohne Leitungsdruck. Das System kann damit flexibel mit einem Wasserreservoir betrieben werden. Eine zukünftige Erweiterung könnte durch Solarstrom autark erfolgen.

Im Rahmen dieses Projekts wird ein funktionsfähiger Prototyp (PoC) zur automatisierten Pflanzenbewässerung realisiert, bei dem die Auswahl der eingesetzten Hard- und Softwarekomponenten vorrangig unter dem Aspekt der funktionalen Erfüllung bei gleichzeitig minimalem Kostenaufwand getroffen wurde.

Als Entwicklungsumgebung wird das **ESP-IDF-Framework** mit FreeRTOS gewählt, da es eine hardwarenahe, echtzeitfähige und performante Steuerung erlaubt. Diese Eigenschaften sind insbesondere bei der parallelen Abarbeitung von Sensor- und Kommunikationsaufgaben essenziell. Die Dual-Core-Architektur des **ESP32-WROOM-32D** wurde hierbei gezielt zur Trennung kritischer Tasks eingesetzt.

Zur Sensorik werden bewusst einfache und kostengünstige Module ausgewählt: der **DHT11** für Temperatur- und Luftfeuchte, der **BH1750** zur Erfassung der Lichtintensität sowie ein **resistiver Bodenfeuchtesensor** auf Basis eines LM393-Komparators. Obwohl diese Komponenten hinsichtlich Genauigkeit und Langzeitstabilität nicht die leistungsfähigsten

auf dem Markt sind, erfüllen sie die Anforderungen eines PoC bei minimalen Kosten und geringem Integrationsaufwand.

Insgesamt erfolgte die Auswahl aller Komponenten bewusst pragmatisch: Ziel war die prototypische Funktionsfähigkeit mit einfacher Einbindung und möglichst niedrigen Kosten. Damit bietet das System eine solide Basis für weiterführende Entwicklungen hin zu einer robusten, autarken und skalierbaren IoT-Lösung.

5 Umsetzung

5.1 Servicearchitektur

In diesem Kapitel wird der Aufbau des Systems erläutert.

5.1.1 Vorteile der Microservice-Architektur im Vergleich zum Monolithen

Die Microservice-Architektur stellt eine moderne Alternative zum klassischen monolithischen Architekturansatz dar. Bei einem Monolithen sind sämtliche Funktionalitäten der Anwendung innerhalb einer einzigen Codebasis und eines gemeinsam deployten Systems gebündelt. Änderungen an einzelnen Komponenten erfordern oftmals eine vollständige Neuveröffentlichung des Gesamtsystems, was zu hohen Wartungsaufwänden und eingeschränkter Flexibilität führt [139].

Im Gegensatz dazu ist die Microservice-Architektur durch eine feingranulare Zerlegung der Anwendung in unabhängige Dienste gekennzeichnet [140]. Jeder dieser Dienste kapselt eine spezifische Funktionalität und kommuniziert über wohldefinierte Schnittstellen (z.B. REST oder Messaging-Systeme wie Solace).

Für das vorliegende System ergeben sich daraus mehrere konkrete Vorteile, die im Vergleich zur monolithischen Architektur besonders relevant sind. Im Bereich der Skalierbarkeit zeigt sich, dass einzelne Microservices – etwa die KI-Komponente oder der IoT-Dienst – unabhängig voneinander hoch- oder herunterskaliert werden können, ohne dass andere Systemteile betroffen sind. Dies ermöglicht eine ressourcenschonende und bedarfsgerechte Nutzung der Infrastruktur. Auch die Wartbarkeit des Systems profitiert erheblich vom modularen Aufbau: Fehlerbehebungen oder Erweiterungen können gezielt innerhalb einzelner Microservices erfolgen, ohne das Gesamtsystem zu beeinträchtigen. Dadurch verringert sich nicht nur das Risiko unerwünschter Nebeneffekte, sondern auch die Ausfallzeiten bei der Weiterentwicklung [139].

Darüber hinaus erlaubt die Microservice-Architektur eine differenzierte Ressourcenzuweisung. Dienste mit geringer Last können ressourcenschonend betrieben werden, was sich

positiv auf die Betriebskosten auswirkt. Diese Effizienz ist bei monolithischen Systemen nur schwer zu erreichen, da dort stets das gesamte System bereitgestellt werden muss. Ein weiterer Vorteil liegt in der technologischen Vielfalt: Da Microservices entkoppelt sind, können unterschiedliche Programmiersprachen und Frameworks eingesetzt werden. So kann beispielsweise die KI-Analyse in Python erfolgen, während das Backend in Rust realisiert ist. Dies fördert die Nutzung spezialisierter Technologien je nach Anwendungsfall [139].

Nicht zuletzt verbessert sich durch Microservices auch die sogenannte Capability-Isolation. Bestimmte Verantwortlichkeiten – etwa die E-Mail-Verifikation oder die Steuerung der Hardware-Kommunikation – sind eindeutig lokalisiert und nur an einer Stelle im System implementiert. Dies erleichtert sowohl die Wartung als auch die gezielte Erweiterung einzelner Funktionen [140].

In Summe ergibt sich eine deutlich höhere Flexibilität, Erweiterbarkeit und Ausfallsicherheit. Für ein wachsendes System mit heterogenen Anforderungen – wie im vorliegenden Projekt – ist der Microservice-Ansatz daher der Monolith-Architektur in mehrfacher Hinsicht überlegen [140].

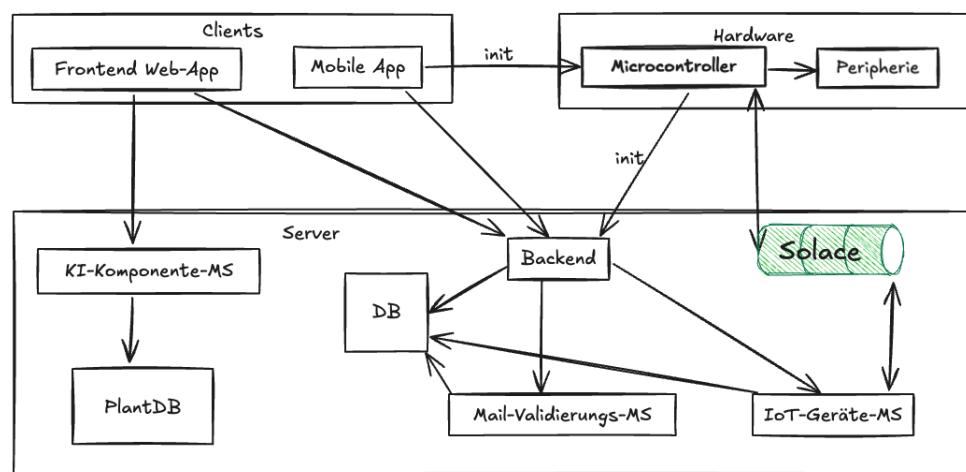


Abbildung 5.1: Kommunikation zwischen den Komponenten von Sensorsora

5.1.2 Hosting und Systemverbindungen

Die Systemarchitektur der Anwendung basiert auf einem verteilten Ansatz, bei dem mehrere spezialisierte Komponenten zusammenwirken. Das Hosting erfolgt Docker basiert, wobei die Dienste in voneinander getrennten Containern betrieben werden. Diese Struktur ermöglicht eine hohe Skalierbarkeit und eine klare funktionale Trennung zwischen einzelnen Bereichen. Zentraler Einstiegspunkt für die NutzerInnen ist die Webanwendung

bzw. die mobile App, welche beide als Client-Komponenten fungieren. Gewöhnlich verlaufen alle Anfragen dieser Clients über das zentrale Backend. Dieses bildet den Kern der Serverarchitektur und übernimmt die Koordination aller systemrelevanten Prozesse. Eine Besonderheit stellt das Hinzufügen neuer Controller dar. In diesem Fall kommuniziert der Client direkt mit dem Mikrocontroller, um eine initiale Verbindung und Konfiguration zu ermöglichen. Diese Kommunikation ist notwendig, um die Geräte korrekt in das System einzubinden, bevor sie in den regulären Datenfluss übergehen. Einige Komponenten wurden bewusst aus dem Haupt-Backend ausgelagert, um Zuständigkeiten zu trennen und das System modular zu halten. Dazu gehört unter anderem die Kommunikation mit den Mikrocontrollern, die über eine eventbasierte Architektur mithilfe des Nachrichtensystems *Solace* realisiert wird. Dabei senden die Controller ihre Sensordaten nicht direkt an das Backend, sondern publizieren diese über Solace. Ein spezialisierter Microservice nimmt diese Nachrichten entgegen und verarbeitet sie asynchron weiter. Auch die E-Mail-Verifikation sowie die Analyse durch die KI-Komponente sind eigenständige Microservices, die über interne APIs in das Gesamtsystem integriert sind. Dies erlaubt eine unabhängige Entwicklung und Wartung dieser Teifunktionen ohne direkte Abhängigkeit vom Kernsystem. Die Datenpersistenz erfolgt über eine zentrale Datenbank, welche alle benutzerbezogenen und systemkritischen Informationen verwaltet. Die KI-Komponente nutzt für ihre Analysen eine separate Datenbasis zur Speicherung pflanzenspezifischer Informationen. Insgesamt ergibt sich ein flexibles, erweiterbares System, das sowohl unmittelbare Benutzerinteraktionen als auch zeitlich versetzte, datengesteuerte Prozesse unterstützt.

5.2 Frontendarchitektur und Datenflüsse im System

Die Frontendarchitektur des smarten Bewässerungssystems wurde nach dem Paradigma komponentenbasierter Webentwicklung realisiert. Ziel war es, eine modular aufgebaute, wartbare und reaktive Benutzeroberfläche zu schaffen, die flexibel auf unterschiedliche Endgeräte und Benutzeranforderungen reagiert. Im Zentrum steht dabei das Framework Vue.js in Verbindung mit dem State-Management-Tool Pinia. Im Folgenden werden die Strukturierung der Views, das Datenflussmodell sowie die Rolle zentraler Technologien im Detail betrachtet.

5.2.1 Komponentenbasierte Struktur und Navigationsmodell

Die Architektur der vorliegenden Anwendung folgt einem komponentenbasierten Ansatz gemäß der Vue.js-Konventionen. Jede View in der Applikation ist als SFC implementiert. Eine SFC vereint die drei wesentlichen Bestandteile einer Webkomponente in einer Datei:

Template, Script und Styles. Das Template definiert die Benutzeroberfläche in HTML-ähnlicher Syntax, das Script implementiert die zugehörige Logik (meist in TypeScript), und der Style-Block regelt das visuelle Layout mittels CSS bzw. Tailwind CSS. Diese Trennung innerhalb einer Datei fördert sowohl die Lesbarkeit als auch die Wiederverwendbarkeit von Komponenten [64].

Die Views der Anwendung (z. B. `HomeView`, `SinglePlantView`, `GroupView`, `SingleSensorView` sowie `PlantListView`) stellen jeweils eigenständige Seiten dar, die durch den Einsatz des Vue Routers dynamisch geladen werden. Jede dieser Views aggregiert untergeordnete Komponenten wie Karten, Dialoge, Navigationsleisten oder Diagramme und bindet dabei die jeweils relevanten Daten aus dem zentralen Zustand.

Die Navigationsstruktur ist hierarchisch aufgebaut. Eine Hauptansicht (`HomeView`) dient als Einstiegspunkt und aggregiert Informationen aus den verschiedenen Kontexten: Räume, Pflanzen und deren Sensordaten. Ausgehend davon ermöglicht das Routing eine Tiefennavigation bis auf Objektebene, z. B. zum Bearbeiten einer bestimmten Pflanze. Dies fördert die kognitive Abbildung realweltlicher Strukturen (Wohnung → Raum → Pflanze) im digitalen Raum.

5.2.2 Pinia als Vermittlungsinstanz zwischen API und Frontend

Zur zentralen Zustandsverwaltung kommt Pinia zum Einsatz, welches das offizielle State-Management-System für Vue 3 darstellt [66], [75]. Anders als bei Vuex erfolgt die Definition eines Stores in Pinia mittels der Funktion `defineStore`, wobei sowohl State als auch Actions und Getters kapsuliert definiert werden. Diese Struktur unterstützt sowohl die Modularität als auch die Wiederverwendbarkeit der Zustandslogik.

Pinia fungiert im Anwendungskontext als Puffer und vermittelnde Instanz zwischen dem Frontend und der REST-API. Die Stores agieren als Cache: sie speichern persistente Daten über Komponentenlebenszyklen hinweg und reduzieren dadurch die Anzahl notwendiger API-Anfragen. Dies verbessert sowohl die Performance als auch die Benutzererfahrung, da viele Interaktionen lokal bedient werden können. Persistiert wird der Zustand mittels `pinia-plugin-persistedstate` im `localStorage`, wodurch Informationen wie eingeloggte Nutzer oder selektierte Objekte auch bei einem Seitenreload erhalten bleiben.

In der Applikation existieren getrennte Stores für Benutzerinformationen (`user.ts`), Authentifizierung (`auth.ts`), Pflanzen (`plant.ts`), Geräte (`device.ts`) und Räume (`room.ts`). Jeder Store definiert spezifische Actions, typischerweise asynchrone Methoden, alle die API-Schnittstellen repräsentieren und einige Weitere. Wenn Änderungen stattfinden, werden diese immer direkt an die API gesendet. Wenn Daten abgefragt werden, wird zuerst geprüft ob sie im Store verfügbar sind, wenn nicht wird erst eine Anfrage ans Ba-

ckend gestellt. Diese neuen Daten werden dann gespeichert und automatisch in die andern Stores synchronisiert.

Optional kann eine Aktion auch mit einem `force`-Flag aufgerufen werden, welches eine explizite Aktualisierung erzwingt. Dies geschieht Beispielsweise bei "Pull-to-Refresh". Diese Strategie erlaubt einen kontrollierten Kompromiss zwischen Reaktivität und Resourceneffizienz.

Um die Daten vor missbräuchlichen Zugriff zu schützen, wurde eine explizite Clear-Strategie implementiert. Bei Logout oder Benutzerwechsel werden alle Stores mittels `clearData()` zurückgesetzt, wodurch Persistenzdaten und Zustand explizit gelöscht werden. Das explizite Löschen ist auch über die Benutzereinstellung möglich.

5.2.3 Datenfluss nach dem Flux-Prinzip

Die Applikation folgt in ihrer Zustandslogik dem Flux-Prinzip, das ursprünglich von Facebook zur Beherrschung komplexer UI-Zustände vorgeschlagen wurde. Charakteristisch für dieses Architekturmodell ist ein strikt unidirektonaler Datenfluss: Interaktionen in der Benutzeroberfläche führen zu sogenannten Actions, die logische Operationen wie API-Aufrufe oder Validierungen auslösen. Die dabei gewonnenen Daten werden im zentralen State-Container gespeichert, welcher wiederum die View reaktiv aktualisiert. Dieser Ablauf lässt sich als Kette beschreiben: `UI → Action → Backend → Store → UI` [141], [142].

Die Trennung der Zuständigkeiten – insbesondere zwischen Anzeige, Logik und Datenhaltung – begünstigt eine konsistente und vorhersehbare Datenverwaltung. Da alle Zustandsveränderungen über dedizierte Actions verlaufen und sich zentral nachverfolgen lassen, verbessert das Modell sowohl die Testbarkeit als auch die Wartbarkeit der Anwendung [141]. In Kombination mit Pinia, das als modernes, modulbasiertes State-Management-Tool agiert, ergibt sich eine Architektur, die eng an Flux angelehnt ist, dabei jedoch die Komplexität traditioneller Implementierungen (z. B. Redux) vermeidet.

5.2.4 Fazit

Die vorliegende Frontend-Architektur basiert auf einem robusten Zusammenspiel modularer Komponenten, zentralisiertem State-Management mit Pinia und asynchroner Datenkommunikation. Die Trennung von Zustandslogik und Darstellung, kombiniert mit der Persistenz und Synchronisationsstrategie, gewährleistet eine wartbare und benutzerfreundliche Applikation.

5.3 Benutzerzentriertes Design und UI/UX im Frontend

Im Bezug auf die theoretische Erläuterung zentraler Konzepte wie UCD und den Usability-Heuristiken nach Nielsen sowie dem UX-Leitbild des Responsive Designs, wird in diesem Abschnitt die konkrete Umsetzung dieser Prinzipien im Rahmen der Vue.js-basierten Anwendung dargestellt. Ziel ist es, die Überführung theoretischer Vorgaben in praktische Gestaltungslösungen nachvollziehbar zu machen und zu zeigen, wie benutzerzentrierte Entwicklung zur Verbesserung der UI-Qualität beiträgt.

5.3.1 Anwendung von UCD im smarten Bewässerungssystem

Die Nutzerforschung erfolgte durch halbstrukturierte Interviews mit VertreterInnen der Zielgruppe (z. B. HobbygärtnerInnen, technikaffine Personen). Daraus wurden mehrere Personas abgeleitet, die unterschiedliche Nutzungsmotive wie einfache Bedienung, Transparenz von Sensordaten und Kooperation abbilden.

Darauf aufbauend wurden Wireframes auf Papier entwickelt, welche die Informationsarchitektur und zentrale Navigationsstrukturen skizzieren. Diese papierbasierten Modelle wurden iterativ angepasst und mit ausgewählten Testpersonen diskutiert. Durch diese formative Evaluation konnte bereits vor der Implementierung auf zentrale Anforderungen reagiert werden.

Auf Basis des Nutzerfeedbacks wurde die Darstellung der Gruppenansicht verbessert. Konkret ergaben sich folgende Anforderungen: Die NutzerInnen wünschten sich eine übersichtliche Darstellung der Gruppen sowie die Möglichkeit, auf einfache Weise die MitgliederInnen einer Gruppe einzusehen, ohne dass zu viele Informationen gleichzeitig auf dem Bildschirm erscheinen. Um diese Bedürfnisse zu erfüllen, wurde die GroupsView als Card-Layout konzipiert.

Diese Card präsentiert auf den ersten Blick nur die wichtigsten Informationen einer Gruppe. Über den Titel oder dem Button können die NutzerInnen die Card bei Bedarf „ausklappen“. Wird der Button gedrückt, erweitert sich die Card dynamisch und zeigt alle zugehörigen MitgliederInnen an. Dies verbessert die Übersichtlichkeit, da nicht alle Details permanent sichtbar sind und die NutzerInnen selbst steuern können, wann sie vertiefte Informationen einsehen.

Besonders benutzerfreundlich ist die neue Lösung auch darin, dass, wenn nur eine einzige Gruppe vorhanden ist, diese Card bereits automatisch ausgeklappt dargestellt wird. Auf diese Weise entfällt ein unnötiger zusätzlicher Klick und der direkte Zugriff auf die Gruppendetails wird erleichtert – ein kleines Detail, das jedoch signifikant zur

Verbesserung der User Experience beiträgt.

Eine weitere Optimierung ist auf der SinglePlantView gemacht worden. Aufgrund Nutzerfeedbacks wurde eine horizontale Linie in das Diagramm integriert, um den Sollwert des Messatzes optisch darzustellen. Die Linie ermöglicht es den Nutzerinnen und Nutzern sofort zu erkennen, wo sich der Sollwert im Vergleich zu den aktuellen Messwerten befindet, sodass Abweichungen zwischen Soll- und Ist-Werten intuitiv nachvollziehbar werden. Dadurch wird nicht nur die Übersicht verbessert, sondern auch die Entscheidungsfindung optimiert, da eine klare visuelle Referenz bereitgestellt wird, anhand derer schneller und fundierter bestimmt werden kann, ob und in welchem Umfang eine Anpassung – beispielsweise im Bewässerungsprozess – erforderlich ist.

Durch die Integration der horizontalen Linie wird die Anwendung benutzerfreundlicher und nachvollziehbarer gestaltet. Gleichzeitig fügt sich die Linie nahtlos in das bestehende Design der SinglePlantView ein, das auf eine klare und konsistente Visualisierung von Daten setzt und damit zentrale Usability-Heuristiken wie die „Sichtbarkeit des Systemstatus“ sowie „Konsistenz und Standards“ unterstützt.

5.3.2 Verwendete Heuristiken in der Anwendung

Im Rahmen der konkreten Umsetzung wurden mehrere der zehn Usability-Heuristiken nach Nielsen gezielt berücksichtigt und systematisch in die Gestaltung der Benutzeroberfläche integriert:

Die Heuristik der Sichtbarkeit des Systemstatus wird durch die Verwendung von Toast-Notifications umgesetzt. Diese erscheinen automatisch bei allen Backend-Abfragen und informieren die NutzerInnen unmittelbar über den Verlauf und das Ergebnis einer Operation. Zusätzlich zeigen Statusindikatoren den Zustand des Sensors an.

Konsistenz und Standards werden durch den Einsatz von Tailwind CSS in Verbindung mit einheitlich definierten Designvariablen gewährleistet. Farben wie `primary`, `secondary`, `destructive` oder `background` kommen konsistent in Buttons, Karten und Formularen zum Einsatz und tragen zu einem kohärenten Erscheinungsbild bei [70].

Zur Umsetzung der Heuristik Fehlervorbeugung wurden alle Formulare mit clientseitiger Validierung ausgestattet. Eingaben werden bereits vor dem Absenden überprüft und Fehler mit Toast-Notifications angezeigt. Leere Eingabefelder enthalten stets einen Platzhalter, der die erwartete Eingabe beschreibt und so die korrekte Nutzung unterstützt.

Die Heuristik Hilfe und Dokumentation wurde durch einige kontextabhängige Tooltips sowie strukturierte Leere-Zustandsanzeigen berücksichtigt. Diese informieren über die nächsten Schritte oder ermöglichen eine direkte Navigation zur entsprechenden Aktion.

Ergänzend wurde auch die Heuristik Entsprechung zwischen System und realer Welt umgesetzt. Die hierarchische Struktur der Anwendung – von der Wohnung über Zimmer

bis zu einzelnen Pflanzen – entspricht einem mentalen Modell aus dem Alltagskontext. Diese logische Ordnung fördert die Orientierung und trägt zu einer intuitiven Navigation bei.

Weitere Heuristiken wie Ästhetisches und minimalistisches Design sind durch das reduzierte Tailwind-basierte UI implizit realisiert worden.

Insgesamt zeigt sich, dass zentrale Usability-Prinzipien systematisch in das UI-Design integriert wurden, um eine benutzerfreundliche und robuste Anwendungserfahrung zu gewährleisten.

5.3.3 Responsive Design

Die Umsetzung des Responsive Designs wurde dabei so angelegt, dass die Anwendung unabhängig von der verwendeten Gerätgröße ein konsistentes und nutzerfreundliches Erlebnis bietet. Mithilfe der Tailwind-Breakpoints `sm`, `md`, `lg` und `xl` können Layout, Typografie und Abstände flexibel an kleinere, mittlere und größere Displays angepasst werden [70]. Dies stellt sicher, dass die einzelnen Interface-Elemente, wie Buttons, Karten und Navigationsmenüs, sich dynamisch skalieren und neu anordnen, um eine optimale Lesbarkeit und Bedienbarkeit zu gewährleisten.

Für alle Gerätetypen wurde zudem eine Bottom-Navigation implementiert, die insbesondere auf mobilen Endgeräten eine intuitive und leicht zugängliche Navigation ermöglicht. Diese Navigation dient als zentrales Steuerelement, das es den NutzerInnen erlaubt, unkompliziert zwischen den Hauptbereichen der Anwendung zu wechseln, ohne auf aufwendige und unübersichtliche Menüstrukturen zurückgreifen zu müssen.

Ein weiteres Gestaltungselement ist das horizontale Scrollen auf der Startseite. Dieses Feature wurde eingeführt, um mehrere Informationskarten kompakt darzustellen, ohne dass der vertikale Platz unnötig beansprucht wird. Durch diese Anordnung können NutzerInnen schnell einen Überblick über verschiedene Inhalte erhalten und bei Bedarf mittels horizontaler Gesten zusätzliche Details abrufen. Insgesamt trägt das responsive Design dazu bei, dass die Anwendung sich flexibel an die individuellen Bedürfnisse und Nutzungsszenarien der AnwenderInnen anpasst und ein nahtloses Nutzungserlebnis über alle Geräte hinweg gewährleistet.

5.3.4 User-Stories und funktionale Umsetzung

Zur nutzerzentrierten Anforderungsdefinition wurden User-Stories eingesetzt, etwa:

- „Als Benutzer möchte ich ein Pflanzenbild hochladen, damit die Pflanze automatisch erkannt wird.“

- „Als Benutzer möchte ich auch Mitglied anderer Gruppen sein, um gemeinsam mit anderen NutzerInnen Pflanzen zu pflegen.“

Diese flossen in die Entwicklung dedizierter Komponenten ein (z. B. UploadPhotoView, GroupsView) und sicherten eine nutzergesteuerte Gestaltung.

5.4 Erweiterte Frontend-Techniken

Im Folgenden werden ausgewählte Techniken vorgestellt, die in modernen Frontend-Architekturen zum Einsatz kommen. Einige dieser Methoden, wie Lazy Loading und Performance-Audits, wurden im Rahmen dieser Arbeit bereits angewendet. Andere Techniken, wie automatisierte Tests, werden exemplarisch vorgestellt, jedoch im Rahmen dieses POC nicht implementiert.

5.4.1 Lazy Loading in der Anwendung

Zur Optimierung der initialen Ladezeit wurde in der entwickelten SPA aktiv *Lazy Loading* eingesetzt. Durch die dynamische Einbindung von Komponenten beim Navigieren zwischen Routen konnte die Bundle-Größe signifikant reduziert und die Interaktivität der Anwendung beschleunigt werden.

Ein Beispiel für Lazy Loading stellt die dynamisch eingebundene Route zur Detailansicht einer Pflanze dar. Die zugehörige View `SinglePlantView.vue` wird erst bei tatsächlichem Aufruf geladen:

```

1  routes: [
2    {
3      path: '/plant/:id',
4      name: 'plantX',
5      component: () => import('../views/SinglePlantView.vue'),
6      meta: { requiresAuth: true, title: 'title.plant' },
7    }
8  ]

```

Listing 5.1: Lazy Loading per Route in Vue Router

Dieses Prinzip wurde konsistent auf alle Unterseiten angewendet. Der verwendete Build-Tool *Vite* unterstützt dabei automatisch Code-Splitting und Tree Shaking, wodurch überflüssiger Code im Produktionsbuild entfernt wird [143], [144].

5.4.2 Frontend-Messung mit Lighthouse

Zur Bewertung der Qualität der entwickelten Anwendung wurden regelmäßig *Lighthouse-Audits* durchgeführt. Diese wurden in den Chrome Developer Tools erzeugt und analysierten zentrale Metriken wie [145]:

- **Performance:** First Contentful Paint, Time to Interactive, Speed Index
- **Accessibility:** Farbkontraste, semantische Struktur, ARIA-Rollen
- **Best Practices:** Ressourcennutzung, HTTPS
- **SEO:** Meta-Tags

Diese Angaben, wurden genutzt um stetig die Anwendung zu verbessern, gleich auch wenn bei einem POC nicht der Schwerpunkt auf SEO oder Accessability liegt. Die Performance wird auf den verschiedenen Seiten teilweise sehr unterschiedlich bewertet, da aber keine starken Verzögerungen bei der Bedienung identifiziert wurden, wurde keine allgemeine Optimierung durchgeführt.

5.5 Aufbau einer spezifischen View als Vertreter

Die Datei `SinglePlantView.vue` bildet das Grundgerüst für die Detailansicht einer einzelnen Pflanze in. Diese View ist modular aufgebaut und umfasst mehrere miteinander koordinierte Komponenten, die sowohl funktional als auch visuell klar voneinander getrennt sind. Die Umsetzung folgt modernen Prinzipien komponentenbasierter Architektur in Vue, wobei jede logische Funktionseinheit in eine eigene Komponente oder ein strukturell abgegrenztes Template-Element eingebettet ist. Die Aufteilung in Subbereiche ergibt sich direkt aus den Bedürfnissen einer klaren Benutzerführung sowie der funktionalen Entkopplung von Darstellung und Logik. Eine Darstellung der kompletten Komponente ist in 5.2 auf der nächsten Seite zu sehen.

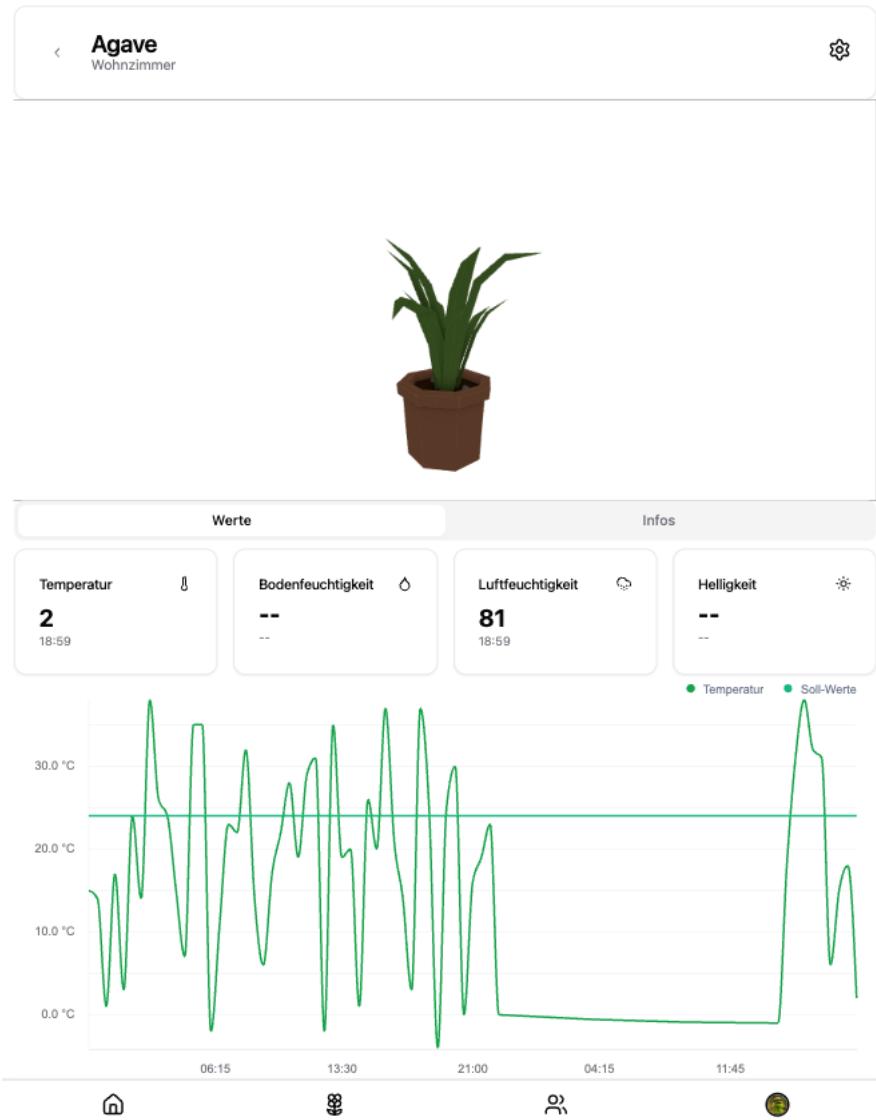


Abbildung 5.2: SinglePlantView in Aktion

Im oberen Abschnitt ist die Komponente `NavCard.vue` zu finden. Die ist für Überschriften mit einfacher Navigation zuständig in mehreren Views. Dieser Bereich ist durch die statische Anzeige des Pflanzennamens („Agave“) sowie der Rauminformation („Wohnzimmer“) gekennzeichnet. Diese Informationen werden direkt aus dem zentralen Datenmodell geladen, welches über ein Pinia-Store-Modul eingebunden ist. Der Benutzer erhält hier sofort kontextuelle Informationen zur Zuordnung der Pflanze im System.

Unmittelbar darunter befindet sich die 3D-Visualisierung der Pflanze, welche als zentrales visuelles Element prominent dargestellt ist. Diese Visualisierung basiert auf einem in der Datei `plantAvatars.ts` definierten Pflanzenmodell, das auf Basis des Pflanzentyps dynamisch geladen wird. Die Komponente zur Darstellung selbst ist als `<plant3d>` eingebettet, wobei ein Canvas-Rendering mit `Three.js` verwendet wird. Die visuelle

Präsentation trägt zur Gamification der Anwendung bei, indem sie die emotionale Bindung und Wiedererkennung der Pflanzen fördern soll [24].

Darunter folgt ein horizontal geteilter Abschnitt mit zwei Tabs: „Werte“ und „Infos“. Der Reiter „Werte“ ist standardmäßig aktiv, was sich in der visuellen Hervorhebung des Tabs zeigt. Innerhalb dieses Tabs sind vier Kartenkomponenten (Card-Komponenten) zu erkennen, die jeweils einen Umweltsensorwert repräsentieren: Temperatur, Bodenfeuchtigkeit, Luftfeuchtigkeit und Helligkeit. Diese sind als wiederverwendbare UI-Komponenten realisiert, die dynamisch Daten einlesen und anzeigen. Die Temperatur- und Luftfeuchtigkeitswerte sind im dargestellten Beispiel verfügbar („18 °C“ und „70 %“), während Bodenfeuchte und Lichtstärke als nicht verfügbar („–“) markiert sind – ein Hinweis auf fehlerhafte oder fehlende Sensoranbindung. Jede Karte zeigt zusätzlich den letzten Messzeitpunkt, was eine präzise Einordnung der Datenqualität ermöglicht.

Der untere Teil der View wird durch die Verlaufsgraphen-Komponente dominiert, die in `PlantMeasuredValuesChart.vue` ausgelagert ist. Diese Komponente nutzt den Wrapper von `shadcn-vue` für `unovis`, um Messwerte über einen Zeitraum hinweg grafisch darzustellen. Im dargestellten Screenshot ist ein Temperaturliniendiagramm zu sehen, das über 24 Stunden Werte anzeigt. Ein horizontaler Zielwert (Soll-Wert) ist ebenfalls dargestellt, was dem Benutzer eine sofortige Einschätzung der Umweltbedingungen erlaubt. Die Auswahl dieses Visualisierungsformats folgt dem Prinzip der kognitiven Entlastung: Durch einfache visuelle Kodierung können Zustände schneller interpretiert werden als durch numerische Tabellen.

Abgeschlossen wird die Komponente durch die `BottomNavBar.vue`, die über vier Icons eine einfache Navigation innerhalb der Anwendung ermöglicht. Diese sind als feste UI-Komponenten realisiert, wobei ein Button speziell dem Rücksprung zur Pflanzenübersicht oder der Startseite dient.

Insgesamt ergibt sich aus dieser strukturierten Aufteilung ein konsistentes, nutzerzentriertes Interface, das sowohl eine einfache Übersicht als auch eine tiefgehende Analyse einzelner Pflanzendaten erlaubt. Die klare funktionale Trennung – Datenanzeige oben, Visualisierung unten, Navigation ganz unten – folgt bewährten Usability-Prinzipien, die sich in wissenschaftlicher Literatur zur Mensch-Computer-Interaktion vielfach bewährt haben [146].

5.6 KI-Komponente zur automatisierten Pflanzenklassifikation

Ziel dieses Moduls war der Aufbau eines robusten Deep-Learning-Modells zur Klassifikation von Pflanzenarten anhand fotografischer Bilddaten. Als Datengrundlage diente der

PlantNet-300K-Datensatz, ein umfassender, realweltlicher Datensatz, der über 300.000 Pflanzenbilder aus verschiedenen Regionen und Perspektiven umfasst [147]. Der Ausgangsdatensatz enthielt über 1.000 Pflanzenklassen, wobei ein starker Klassenunterschied hinsichtlich der Bildanzahl pro Klasse vorlag – von 1 Bildern bis zu über 5.000 Bildern pro Art. Um ein Mindestmaß an statistischer Repräsentation zu gewährleisten und extreme Ausreißerklassen zu vermeiden, wurden für das Training ausschließlich Klassen berücksichtigt, die mindestens 5 Bilder enthielten. Diese Filterung reduzierte das Klassenspektrum auf 837 distinkte Pflanzenarten. Diese Klassen besteht zum Großteils aus nicht sehr verbreiteten Pflanzen in Deutschland.

5.6.1 Modellarchitektur und Trainingsstrategie

Die Trainingspipeline basiert vollständig auf einem vortrainierten 50-layer Residual Network (ResNet50)-Modell, das von Beginn an zur Initialisierung genutzt wurde. Die Wahl von ResNet50 ergibt sich aus mehreren architektonischen und empirisch belegten Vorteilen: Residual Network (ResNet) wurde eingeführt, um ein zentrales Problem tiefer neuronaler Netze zu adressieren – das sogenannte *Degradationsproblem*. Dabei nimmt bei tiefer werdenden Netzen nicht nur die Trainingszeit zu, sondern mitunter sogar die Klassifikationsleistung ab, obwohl das Netz mehr Kapazität besitzt [148].

Der Kernmechanismus zur Lösung dieses Problems sind *Residual-Blöcke*. Statt rohe Ausgaben direkt weiterzureichen, lernen ResNet-Blöcke nur die Abweichung von der Identität[148].

Das ResNet50-Modell besteht aus insgesamt 50 Schichten, die sich aufteilen in:

- eine initiale Convolution-Schicht (7x7 Convolution + MaxPooling),
- 16 sogenannte „Bottleneck“-Blöcke mit je drei Schichten ($1 \times 1 \rightarrow 3 \times 3 \rightarrow 1 \times 1$ Convolution),
- Batch-Normalisierung und ReLU-Aktivierung in jedem Block,
- eine globale Average-Pooling-Schicht,
- sowie eine abschließende Fully-Connected-Schicht zur Klassifikation.

Durch diese Struktur ist ResNet50 nicht nur leistungsfähig, sondern auch besonders übertragbar auf neue Datendomänen – ein Umstand, der in einer Vielzahl an Transfer-Learning-Studien belegt wurde [149]. Die Tiefe erlaubt es dem Modell, auch feine, visuell komplexe Unterschiede zwischen Pflanzenarten zu modellieren, während die Sprungverbindungen die Stabilität im Training erhalten.

Im initialen Training wurden alle Schichten des Netzwerks feinjustiert. Nach einer ersten Konvergenz wurde ein Finetuning durchgeführt, bei dem ein Großteil der Layer eingefroren wurde, um ausschließlich die letzten Klassifikationsschichten anzupassen. Dieses zweistufige Vorgehen ist ein gängiger Transfer-Learning-Ansatz, insbesondere wenn ein großes Ausgangsmodell (wie ResNet) auf eine domänenspezifische Aufgabe adaptiert wird [150].

5.6.2 Regularisierung und Datenkonsolidierung

Zur Verbesserung der Modellrobustheit wurden mehrere datenaugmentierende Verfahren eingesetzt. Dazu zählen insbesondere Mixup [151] und CutMix [152], welche zu besseren Verallgemeinerungseigenschaften führen, indem sie die Entscheidungsgrenzen im Merkmalsraum glätten. Ergänzt wurden diese Verfahren durch RandAugment, eine robuste Augmentierungsmethode ohne komplexe Hyperparametrierung.

Ein zentrales Vorverarbeitungsschritt war die Anwendung des Moduls `create_merge_map.py`, das vor dem Finetuning genutzt wurde, um Duplikate und taxonomisch redundante Pflanzenklassen zusammenzuführen. Diese Maßnahme reduziert das Risiko semantischer Verwirrung im Trainingsprozess und wurde insbesondere bei identischen oder sehr ähnlichen Arten eingesetzt. Die Notwendigkeit solcher Label-Konsolidierungen ist insbesondere in crowd-basierten, multilinguistisch annotierten Datensätzen wie PlantNet belegt [153].

Darüber hinaus wurde zur Kompensation des hochgradig unausgeglichenen Klassenverhältnisses (5–5000 Bilder pro Klasse) ein Weighted Sampling implementiert. Diese Technik erhöht die Wahrscheinlichkeit der Auswahl von Bildern seltener Klassen während des Batch-Trainings und verhindert so die Dominanz überrepräsentierter Arten im Gradientenfluss – ein gängiger Ansatz zur Balancierung von Imbalancen in Klassifikationsaufgaben [154].

5.6.3 Leistung und Interpretation

Die Trainingszeit betrug insgesamt 65 Stunden, wobei 50 Stunden auf das initiale, vollständig entfrorene Training und 15 Stunden auf das anschließende Finetuning entfielen.

Das finale Modell demonstrierte eine hohe Klassifikationsfähigkeiten. Die Top-1-Accuracy von 77% bedeutet, dass das Modell in über drei Viertel aller Fälle die exakte Pflanzenart korrekt identifizierte. Die Top-5-Accuracy von 95% zeigt, dass sich die wahre Klasse in der Mehrheit der Fälle unter den fünf wahrscheinlichsten Vorhersagen befand – ein Maß, das insbesondere in praktischen Anwendungen wie botanischen Bestimmungs-Apps von Relevanz ist. In der Abbildung 5.7 auf der nächsten Seite sind Beispiel zu sehen.



Abbildung 5.3: Korrekt erkannt als *Anthurium Andraeanum* mit 99,72%



Abbildung 5.4: Korrekt erkannt als *Fragaria X Ananassa* mit 90,42%

Abbildung oben: Übersicht einiger markanter Testpflanzen.



Abbildung 5.5: Korrekt erkannt als *Lavandula Stoechas* mit 99,49%

Abbildung unten: Vergleich der Genauigkeit zweier Lavendel Arten



Abbildung 5.6: Korrekt erkannt als *Lavandula Angustifolia* mit 98,05%

Abbildung 5.7: Beispiel für die KI-Erkennung

5.6.4 Rolle der KI-Komponente bei der Pflanzenerstellung

Die KI-Komponente wird im Gesamtsystem insbesondere bei der Erstellung neuer Pflanzeninstanzen eingesetzt. NutzerInnen haben dabei die Möglichkeit, grundlegende Eigenschaften einer Pflanze – wie den Pflanzennamen oder die Artzugehörigkeit – automatisiert durch ein KI-Modul bestimmen zu lassen. Dieser Vorgang kann sowohl durch das Hochladen eines bestehenden Pflanzenbilds als auch direkt über eine Fotoaufnahme im Browser oder auf mobilen Endgeräten ausgelöst werden. Alternativ besteht die Möglichkeit, ohne KI-Unterstützung nach einer Pflanze zu suchen.

Bei Nutzung der KI-Komponente wird das Bild über das Frontend an einen dedizierten Klassifikationsservice übermittelt. Dieser führt eine Inferenz mit dem trainierten ResNet50-Modell durch und schlägt basierend auf der Bildanalyse eine Pflanzenart vor. Neben der wahrscheinlichsten Klasse wird zusätzlich eine Liste mit weiteren möglichen Arten inklusive Vorhersagewahrscheinlichkeiten generiert. Es werden aber nur Pflanzen mit über 50% zurückgegeben. Diese Vorhersage wird visuell im Interface dargestellt und kann von der NutzerIn bestätigt werden.

Der ausgewählte Vorschlag bildet dann die Grundlage für die Vorbelegung der Eingabefelder zur Pflanzenerstellung, sodass beispielsweise der wissenschaftliche Name, die Spezies und Soll-Werte geschätzt werden. Die KI-Komponente unterstützt somit aktiv die Datenerfassung und sorgt für eine beschleunigte, komfortable Erstellung von Pflanzendatensätzen innerhalb des Systems. Die finale Entscheidung über die Auswahl der vorgeschlagenen Pflanze verbleibt stets bei den Nutzenden.

5.7 Beschreibung des Datenbankaufbaus

Die Datenbank des Systems Sensors ist im Schema `sensors` organisiert und verfolgt eine klar strukturierte, relationale Modellierung mit durchdachter Referentialität und Typisierung. Sie unterstützt die zentralen Funktionen des Systems wie Benutzerverwaltung, Gruppenzugehörigkeit, Raum- und Pflanzenzuordnung sowie Sensor- und Steuerdaten.

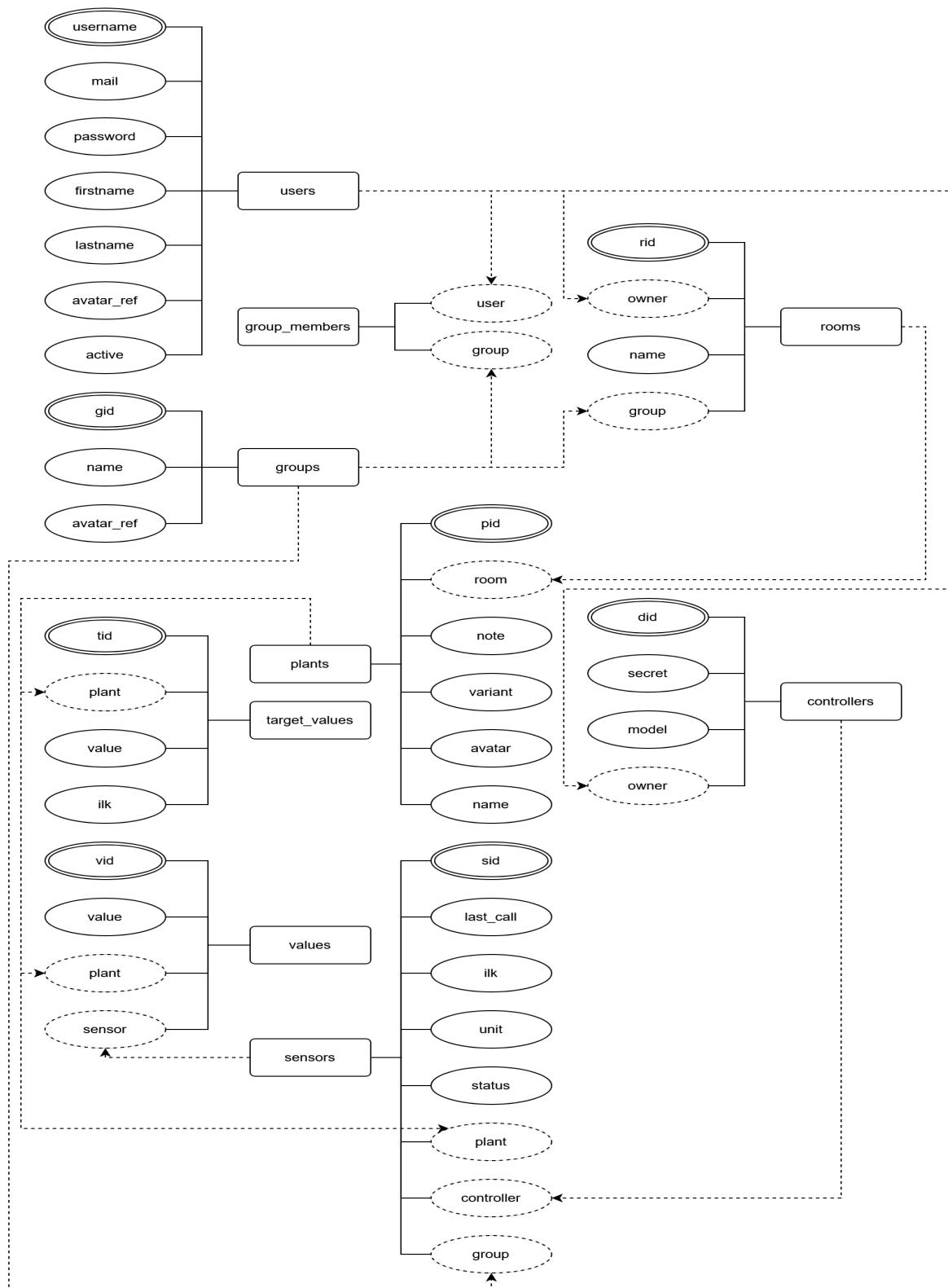


Abbildung 5.8: Sensors Data Bank Struktur

5.7.1 Struktur und Besonderheiten

Benutzerverwaltung: Die Tabelle users bildet die zentrale Entität für Benutzer ab.

Jeder Benutzer besitzt Pflichtangaben sowie einen referenzierten Avatar aus dem dedizierten ENUM-Typ sensora.avatar. Die E-Mail-Adresse ist eindeutig.

Gruppen & Mitgliedschaften: Gruppen (groups) können mehrere Mitglieder haben, realisiert durch die Join-Tabelle group_members. Diese bildet eine klassische Many-to-Many-Beziehung zwischen Nutzern und Gruppen ab.

Räume & Pflanzen: Räume (rooms) können Gruppen zugeordnet sein und besitzen jeweils einen Eigentümer. Pflanzen (plants) sind immer einem Raum zugeordnet und dienen als Ankerpunkt für Messwerte.

Sensorik & Steuerung: Sensoren (sensors) sind mit Controllern (controllers) verknüpft und optional direkt mit einer Pflanze oder Gruppe verbunden. Jeder Sensor verwendet den ENUM-Typ sensora.status, um seinen Zustand zu klassifizieren.

Zielwerte & Messdaten: Pflanzen können über target_values Zielgrößen definieren. Tatsächliche Messwerte werden in der Tabelle values gespeichert und jeweils einem Sensor sowie einer Pflanze zugeordnet.

5.7.2 Technische Merkmale

- Es kommen ENUM-Typen zum Einsatz, um Felder wie Avatar und Status typensicher und standardisiert zu definieren.
- Sämtliche Fremdschlüsselbeziehungen nutzen CASCADE-Strategien zur Pflege von Konsistenz (z.B. beim Löschen von Benutzern oder Pflanzen).
- Indizes auf eindeutige Felder (z.B. mail, secret) erhöhen die Performanz gezielter Abfragen.
- Die Nutzung von Timestamps mit Standardwerten erlaubt eine automatische Protokollierung von Ereignissen wie Sensoraktivität.

Diese Datenbankstruktur ermöglicht eine flexible, erweiterbare und gleichzeitig robuste Grundlage für die Backend-Logik und garantiert eine nachvollziehbare Abbildung der fachlichen Entitäten.

5.8 Auth-Service: Geräteregistrierung und HMAC-Authentifizierung

Der Auth-Service implementiert die sichere Registrierung und Anmeldung von Controllern mittels eines Challenge-Response-Verfahrens auf Basis eines vorab geteilten Geheimnisses (PSK). Dieser Dienst ist als Flask-Webanwendung realisiert und stellt HTTP-Routen für die Controller-Initialisierung, -Verifikation sowie eine Admin-Registrierung bereit. Er bildet damit das sicherheitskritische Bindeglied zwischen neuen Geräten und der Systeminfrastruktur: Geräte erhalten hier ihre individuellen Messaging-Zugangsdaten, sofern sie einen kryptographischen Besitznachweis erbringen. Im Folgenden werden die Architektur und der Ablauf des Auth-Service beschrieben, gefolgt von besonderen Implementierungsdetails wie der PSK-Überprüfung, Wiederanlaufbarkeit und der automatischen Broker-Konfiguration.

5.8.1 Architektur und persistente Datenhaltung

Der Auth-Service ist als Flask-Applikation mit dokumentierter REST-API (via Swagger/Flasgger) umgesetzt. Kern der Implementierung ist eine zentrale Konfigurationsdatei (`auth_config.json`), die folgende Informationen persistent speichert:

1. Authorized Controllers: eine Liste aller registrierten Controller mit deren Controller-ID (eindeutige Kennung), zugehörigem PSK (token) sowie einem Hash dieses Tokens (token_hash). Zusätzlich werden Metadaten wie Modell, Besitzer (Benutzername) und Beschreibung gespeichert. Diese Datei dient als kleine lokale Datenbank, um bereits registrierte Geräte und ihre Geheimnisse nachzuhalten.
2. Active Challenges: temporäre Herausforderungen (Challenges) im laufenden Authentifizierungsprozess. Für jeden angeforderten Auth-Vorgang wird ein zufälliger Challenge-String erzeugt und unter dem Schlüssel des token_hash zwischengespeichert. Dies ermöglicht es, eingehende Antworten eindeutig der zuvor ausgegebenen Challenge zuzuordnen, selbst bei parallelen Anfragen.
3. Solace Credentials: bereits für Controller erstellte Zugangs-Credentials für den Message Broker (Solace). Pro Gerät wird hier der erzeugte Broker-Benutzername und das Passwort abgelegt, um bei wiederholter Authentifizierung nicht erneut einen Broker-Account anlegen zu müssen.

Die Konfigurationsdaten werden auf dem Container-Dateisystem unter `/config/auth_config.json` verwaltet und durch Volume-Mounting persistent gehalten. Dadurch bleiben Registrierung und vergebene Credentials auch beim Neustart des Dienstes

erhalten – ein wesentlicher Aspekt für Wiederanlaufbarkeit. Ergänzend hält der Auth-Service eine Datenbankverbindung (PostgreSQL via psycopg2) bereit, um neue Controller auch in der zentralen Systemdatenbank (sensora) zu registrieren. In der Tabelle sensora.controllers werden u.a. Device-ID, Modell, Besitzer und ein vom System generierter geheimer Schlüssel abgelegt. Letzterer unterscheidet sich vom PSK und dient ggf. anderen Zwecken (z.B. der Kommunikation mit externen Anwendungen), ohne das eigentliche PSK offenzulegen. Die doppelte Ablage (Datei und DB) der Registrierungsdaten mag auf den ersten Blick redundant erscheinen, ermöglicht jedoch sowohl schnelle lokale Zugriffe während des Auth-Handshakes als auch die Integration ins relationale Gesamtdatenmodell.

5.8.2 Ablauf der Geräte-Registrierung und Authentifizierung

Der Auth-Service bietet drei Haupt-Endpoints, die den Lebenszyklus eines Geräts abbilden: (a) Registrierung eines neuen Controllers (nur Admin), (b) Initialisierung des Authentifizierungs-Handshakes durch das Gerät und (c) Verifikation der Challenge-Response.

Der Authentifizierungsablauf erfolgt in mehreren Schritten:

1. Admin-Registrierung: Zunächst muss ein neuer Controller in das System aufgenommen werden. Ein Administrator (oder ein automatisierter Setup-Prozess) ruft dazu den Endpoint /api/admin/controller auf und übermittelt zumindest den gewünschten Besitzer (username) sowie optional eine vorgegebene Controller-ID und Modellbeschreibung. Der Request ist durch einen speziellen Admin-API-Key (Header X-Admin-Key) geschützt, sodass nur berechtigte Instanzen Geräte hinzufügen können. Bei Aufruf generiert der Service serverseitig ein zufälliges Token (PSK) für das Gerät (hier als UUID v4). Zusätzlich wird ein Token-Hash berechnet: mittels HMAC-SHA256 über das Token mit einem globalen Server-Geheimnis (TOKEN_SECRET). Dieser Hash dient als öffentlicher Identifikator des Geräts, der das eigentliche Token nicht preisgibt. Anschließend werden die Gerätedaten in der auth_config.json persistiert (authorized_controllers) und ein Eintrag in der DB-Tabelle controllers erzeugt. Der Response an den Admin enthält Controller-ID, Token und Token-Hash, welche sicher an das Gerät für die Inbetriebnahme weitergegeben werden (z.B. manuell oder per Provisioning-App). Dieser einmalige Out-of-Band-Schritt stellt sicher, dass jedes Gerät ein individuelles geheimes Token besitzt, das dem Server bekannt ist.
2. Challenge-Anforderung (Gerät → Auth-Service): Hat das Gerät vom Admin sein Token erhalten und wird erstmals online genommen, initiiert es den Authentifizierungs-

prozess über den Endpoint /api/controller/init. Dabei sendet das Gerät nur den Hash seines Tokens (token_hash) im Request – das eigentliche Token bleibt geheim und wird nie direkt übertragen. Der Auth-Service prüft den Request und generiert eine kryptographisch zufällige Challenge (16 Byte Hex-String via secrets.token_hex) als Antwort. Diese Challenge wird im Server-Konfigurationsspeicher unter active_challenges zusammen mit einem Zeitstempel abgelegt, indiziert durch den token_hash. Die Response an das Gerät enthält die Challenge als JSON. Aus Sicherheitsgründen findet an dieser Stelle noch keine Verifizierung statt – auch ein unbekannter Token-Hash erhält (vorläufig) eine Challenge. Der eigentliche Abgleich erfolgt erst im nächsten Schritt. Dieses zweistufige Verfahren erhöht die Sicherheit, da ein Angreifer ohne Besitz des PSK aus der Challenge alleine keinen Zugang erlangt. Wichtig ist, dass jede Challenge unvorhersehbar und einmalig ist, um Replay-Angriffe auszuschließen. Der Auth-Service stellt dies durch Verwendung eines Kryptografie-Moduls sicher (Python secrets [155] bietet laut Dokumentation einen sicheren Zufallszahlengenerator für solche Token).

3. Challenge-Response (Gerät → Auth-Service): Nach Erhalt der Challenge berechnet das Gerät die Antwort: Es verwendet sein geheimes Token und wendet darauf die gleiche Hash-Funktion an, die auch der Server kennt. Im Code wird dazu HMAC-SHA256 genutzt, wobei das Token als Schlüssel und der Challenge-String als Nachricht dient. Das Ergebnis ist die Challenge-Response (Hex-String). Diese Antwort schickt das Gerät zurück an den Auth-Service, zusammen mit seinem token_hash (zur Identifikation der Challenge) und meist einem Benutzernamen oder Kontoinformationen des Eigentümers. Der Auth-Service schlägt nun in seiner Konfiguration den Eintrag zum token_hash nach: Dort findet er das ursprünglich registrierte Token (PSK) und die zugehörige Controller-ID. Sollte kein Eintrag existieren, wird der Prozess abgebrochen – das Gerät war nie registriert oder der Hash unbekannt (Fehler 403). Andernfalls vergleicht der Server die vom Gerät gesendete HMAC-Antwort mit dem erwarteten Wert, den er selbst berechnet (hmac.compare_digest() verhindert Timing-Angriffe bei der String-Bewertung). Stimmen Response und eigener Wert überein, ist bewiesen, dass das Gerät das geheime Token besitzt und somit authentisch ist. Dieses Challenge-Response-Verfahren stellt eine sichere Authentifizierung dar[156], ohne das PSK selbst über das Netzwerk zu senden. Ein abgehörter Challenge/Response-Wert kann später nicht wiederverwendet werden, da bei der nächsten Anmeldung eine andere Challenge zum Einsatz kommt (kein statisches Passwort)
4. Broker-Zugangsdaten erstellen: Nach erfolgreicher Verifizierung entfernt der Auth-

Service die verwendete Challenge (Verbrauch der einmaligen Challenge) und fährt mit der Provisionierung des Geräts für das Messaging-System fort. Hier greift eine weitere wichtige Implementierungsentscheidung: Der Service konfiguriert den Solace-MQTT-Broker dynamisch über dessen Management-API (SEMP v2). Konkret wird geprüft, ob für den Controller bereits Broker-Zugangsdaten in solace_credentials vorliegen. Ist dies der erste erfolgreiche Auth-Vorgang für dieses Gerät, generiert der Service mittels der Hilfsfunktion create_solace_user(controller_id) einen dedizierten Broker-Account:

- (a) Es wird ein eindeutiger Client-Username für den Controller erstellt (z.B. controller_ab12... gekürzt auf Basis der ID) und ein zufälliges Passwort (secrets.token_urlsafe(32)).
- (b) Für feingranulare Zugriffskontrolle richtet der Service ein eigenes ACL-Profil auf dem Broker ein. Dies geschieht über HTTP-Aufrufe an die SEMPV2-Konfigurationsschnittstelle des Solace-Brokers. Das ACL-Profil für den Controller erlaubt ausschließlich die für diesen Anwendungsfall nötigen Operationen:
 - i. Publish-Erlaubnis: Das Gerät darf nur auf seinem eigenen Topic-Pfad Messwerte publizieren. Im Prototoll wird das Topic-Muster sensora/v1/send/controller_id verwendet. Über SEMP wird ein Topic Exception hinzugefügt, die genau dieses Topic für Publish freigibt (alle anderen werden per Default “disallow” gesetzt).
 - ii. Subscribe-Erlaubnis: Analog erhält das Gerät das Recht, Sollwert-Nachrichten zu abonnieren, die an sein spezifisches Target-Topic gesendet werden. Dies ist typischerweise sensora/v1/receive/controller_id/targetValues. Auch hierfür wird programmgesteuert eine Ausnahme im ACL-Profil hinterlegt. Damit ist sichergestellt, dass der Controller nur Nachrichten empfängt, die explizit an ihn adressiert sind (und z.B. keine fremden Gerätedaten).
- (c) Nachdem ACL-Profil und Berechtigungen erfolgreich angelegt wurden, erstellt der Service über die SEMP-API den Broker-Client-User und weist ihm dieses ACL-Profil zu. Sollte einer der Schritte fehlschlagen (z.B. aufgrund bereits existierender Einträge oder Verbindungsfehler zum Broker), bricht die Funktion ab und der Auth-Vorgang resultiert in einem Server-Fehler (HTTP 500). Im Erfolgsfall erhält der Auth-Service nun ein Credential-Paket bestehend aus Broker-URL, Username und Passwort für den neuen Controller.

Die Nutzung der Solace Element Management Protocol API ermöglicht es, die Broker-Konfiguration zu automatisieren. Solace SEMP ist eine RESTful-Schnittstelle[42],

die das Anlegen von Objekten (Queues, Benutzer, ACLs etc.) per Skript erlaubt. Dadurch wird eine dynamische Inbetriebnahme neuer Geräte ohne manuelle Eingriffe möglich – ein wichtiger Vorteil im IoT-Kontext. Durch entsprechende Fehlerbehandlung (Auswerten des HTTP-Status: 200 Erfolg, 409 „Conflict“ bei bereits vorhandenen Ressourcen) ist die Funktion weitgehend wiederholbar, ohne Inkonsistenzen zu erzeugen. Beispielsweise würde ein erneuter Aufruf für denselben Controller erkennen, dass dessen Benutzerkonto schon existiert (der Auth-Service speichert dies ja auch in seiner config) und überspringt die Neuanlage. So bleibt der Prozess idempotent und ein Gerät könnte die Authentifizierung bei Bedarf erneut durchlaufen, um z.B. verlorene Credentials abzurufen.

5. Registrierung abschließen und Antwort an Gerät: Zum Abschluss der /verify-Route führt der Auth-Service noch zwei Aktionen aus: (a) Eintrag in der Systemdatenbank: Falls noch nicht geschehen, wird der Controller endgültig in der sensora.controllers-Tabelle der DB vermerkt (inkl. Besitzverknüpfung zum Benutzerkonto, was zuvor im Request mitgesendet wurde). Außerdem kann hier optional ein Default-Sensor für den Controller in der DB angelegt werden (im Code wird bspw. ein Platzhalter-Sensor für Temperatur erstellt, um direkt Messwerte speichern zu können). (b) Secure Credential Delivery: Die vom Broker erzeugten Verbindungsdaten (Username-/Passwort, Host) müssen nun dem Gerät mitgeteilt werden. Da diese Angaben sehr sensativ sind, wurden besondere Maßnahmen getroffen, um sie vertraulich und integer zum Gerät zu übertragen. Der Server generiert zunächst einen einmaligen Session Key (eine randomisierte Byte-Sequenz mittels Fernet.generate_key()), mit dem er die Credentials symmetrisch verschlüsselt (Fernet nutzt intern AES-128 in GCM-Modus[157] mit eingebauter HMAC für Integrität). Die so entstehende Ciphertext-Nutzlast wird als encrypted_credentials bereitgestellt. Zusätzlich erzeugt der Server einen Credential-HMAC (credential_key), indem er den Session Key mit dem ursprünglichen Geräte-PSK mittels HMAC-SHA256 signiert. Anschließend sendet der Auth-Service dem Gerät folgende Daten im JSON-Response:

- (a) session_key: der im Base64-Format kodierte symmetrische Schlüssel,
- (b) credential_key: der HMAC (Hex-String) zur Absicherung,
- (c) encrypted_credentials: die verschlüsselten Broker-Zugangsdaten (Base64-Text).

Diese Konstruktion erlaubt es dem Gerät, die erhaltenen Credentials auf Vertrauenswürdigkeit zu prüfen: Nur wenn es den gleichen HMAC über den Session Key mit seinem PSK berechnet und dieser mit dem credential_key übereinstimmt, stammen die Daten eindeutig vom Auth-Service (der das PSK kennt). Damit ist ein Schutz

gegen Man-in-the-Middle-Angriffe erreicht, selbst wenn kein vollwertiger TLS-Kanal vorhanden wäre – ein Angreifer könnte zwar den Session Key und Ciphertext stehlen, hätte aber ohne PSK keine Möglichkeit, gültige Daten vorzutäuschen. Nach erfolgreicher HMAC-Prüfung entschlüsselt das Gerät mit dem Session Key die Credentials und erhält so seinen persönlichen Broker-Login. Ab diesem Zeitpunkt kann sich der Controller am MQTT-Broker anmelden und regulär Sensordaten austauschen. Der einmalig verwendete Session Key ist nun obsolet.

Abschließend bestätigt der Auth-Service dem Gerät die erfolgreiche Verifikation mit HTTP 200. Intern protokolliert er die erfolgreiche Authentifizierung und der Prozess ist abgeschlossen. Jegliche Fehlersituationen unterwegs (z.B. falscher Token, falsche Response, fehlende Eingabefelder) wurden mit aussagekräftigen HTTP-Codes (400 Bad Request, 403 Forbidden) und Log-Meldungen abgefangen, sodass das Gerät bzw. der Administrator direkt Rückmeldung über den Grund eines Scheiterns erhalten.

5.9 Mail-Service: E-Mail-Verifikation von Benutzerkonten

Der Mail-Service ist ein eigenständiger Webservice, der die Verifizierung von Benutzer-E-Mailadressen übernimmt. Im Gesamtsystem wird dieser Service genutzt, um nach einer Benutzerregistrierung sicherzustellen, dass die angegebene E-Mail dem Nutzer gehört und erreichbar ist – ein gängiges Verfahren, um Kontoaktivierungen durch den Nutzer selbst via Klick auf einen Bestätigungslink durchzuführen. Der Mail-Service wurde mit FastAPI (Python) umgesetzt, was die Erstellung asynchroner HTTP-Handler ermöglicht. Die Hauptaufgaben des Dienstes sind: Empfang der Verifikationsanfrage, Validierung mittels eines Pre-Shared Key (zur Absicherung interner Aufrufe), Generierung eines eindeutigen Bestätigungs-Tokens, Versand einer E-Mail mit Bestätigungslink via SMTP und abschließend die Verarbeitung des Bestätigungs-Clicks (Aktivierung des Kontos in der Datenbank).

5.9.1 Architektur und Ablauf der E-Mail-Verifikation

Der Mail-Service verfügt über zwei wesentliche Endpoints: einen POST-Endpoint `/verify` zum Anfordern einer Verifikationsmail und einen GET-Endpoint `/confirm/username/-token` zum Bestätigen. Intern nutzt der Service eine PostgreSQL-Datenbankverbindung (asynchron via `asyncpg`), um Benutzerdatensätze zu prüfen und zu aktualisieren. Der Ablauf lässt sich wie folgt zusammenfassen:

1. Anfrage zur Verifikation (POST /verify): Diese Schnittstelle wird vom übergeordneten System (z.B. dem Web-Frontend oder einem anderen Service) aufgerufen, sobald ein Benutzer eine Registrierung abgeschlossen hat oder eine E-Mail-Bestätigung angefordert wird. Der Request enthält typischerweise den Benutzernamen und die E-Mail-Adresse des Kontos. Zusätzlich erwartet der Service einen geheimen Schlüssel (key), der mitgeschickt wird. Dieser PSK (Mailservice) ist eine einfache Sicherungsmaßnahme, damit nur autorisierte Systeme (etwa das Frontend-Servermodul) den Versand von Verifizierungs-Mails auslösen können – damit wird verhindert, dass Unbefugte massenhaft Verifikations-E-Mails über die öffentliche API triggern. Der Service prüft also zuerst, ob der mitgesandte Schlüssel mit dem in den Umgebungsvariablen hinterlegten Wert (MAILSERVICE_PSK) übereinstimmt. Ist dies nicht der Fall, wird mit HTTP 403 abgebrochen. Ist die Anfrage autorisiert, wird die angegebene Kombination aus username und mail in der Datenbank gesucht (SELECT * FROM sensora.users WHERE username=%s AND mail=%s). Nur wenn ein entsprechender Benutzeraccount existiert und noch als inaktiv markiert ist (dies wird indirekt geprüft, indem z.B. ein Feld active in der DB auf FALSE stehen sollte – im Code wird bei Nichtexistenz direkt 404 gemeldet), wird der Verifikationsprozess fortgesetzt. Im nächsten Schritt erzeugt der Service ein zufälliges Token als einmaligen Bestätigungscode. Hierzu wird Python secrets.token_urlsafe(16) verwendet[155], was einen 22 Zeichen langen kryptographisch sicheren String liefert. Das Token wird in einer in-memory Datenstruktur (tokens Dictionary) unter dem Schlüssel des Benutzernamens gespeichert. Anschließend wird ein Bestätigungslink erstellt, der die URL des Confirmation-Endpoints enthält (inkl. Pfadparameter für Username und Token). Dieser Link hat z.B. die Form: <https://meinserver/confirm/alice/AbCdEfGh...> – er enthält also das geheime Token. Nun versendet der Mail-Service eine E-Mail an die Adresse des Nutzers. Dafür wird ein SMTP-Server (hier Gmail SMTP auf Port 587) verwendet. Über Pythons smtplib wird eine TLS-geschützte Verbindung aufgebaut, der Mailaccount authentifiziert (SMTP-User und Passwort liegen in den Settings) und dann eine Textnachricht verschickt. Der E-Mail-Inhalt besteht aus einem kurzen Text mit der Aufforderung, den Link anzuklicken, um die Registrierung abzuschließen. Absender und Betreff sind entsprechend gesetzt (z.B. "Bitte bestätige deine E-Mail"). Nach erfolgreichem Versand gibt der/verify-Endpoint eine Erfolgsmeldung zurück ({"message: "Verification email sent."} mit HTTP 200). Fehlerfälle: Wenn die E-Mail-Adresse nicht existiert oder der DB-Zugriff fehlschlägt, wird ein HTTP 404 bzw. 500 zurückgegeben. Ein falscher PSK führt zu 403. Falls der SMTP-Versand scheitert (Exception), wird diese von FastAPI als Serverfehler zurück an den Aufrufer propagiert – in ei-

ner robusteren Version könnte man hier spezifisch mit `HTTPException` antworten, doch im gegebenen Code wird auf die eingebaute Exception-Behandlung vertraut.

2. Bestätigungsaufruf (GET `/confirm/username/token`): Diese Route wird aufgerufen, wenn der Benutzer den Link in der Verifikationsmail anklickt. In einem üblichen Web-Anwendungsfluss würde dieser Link z.B. zu einer Erfolgsmeldungsseite führen. Der Mail-Service übernimmt hier im Hintergrund die Aktivierung des Benutzerkontos. Er prüft zunächst, ob zum gegebenen `username` ein Token in seinem Zwischenspeicher vorliegt und ob es mit dem übermittelten Token übereinstimmt. Ist das Token falsch oder nicht (mehr) vorhanden, wird eine HTTP 400 Fehlermeldung erzeugt (`Invalid or expired token.`). Dies deckt sowohl falsch manipulierte URLs als auch abgelaufene Tokens ab – letzteres, weil der Service das Token nach Gebrauch löscht oder nach einem Neustart vergisst (siehe weiter unten). Wenn das Token stimmt, wird mittels Datenbank-Update das Benutzerkonto aktiviert (`UPDATE sensora.users SET active = TRUE WHERE username = ...`). Danach entfernt der Service den genutzten Token aus seinem `tokens`-Dictionary (damit der Link nicht erneut verwendet werden kann, One-Time Use). Schließlich liefert der Endpoint eine einfache HTML-Antwort zurück, die dem Nutzer bestätigt, dass die E-Mail erfolgreich verifiziert wurde (im Code: Rückgabe eines kleinen `<h1>`-HTML mit Erfolgstext). Dieses HTML wird durch FastAPI mithilfe der `HTMLResponse` direkt ausgegeben – so sieht der Nutzer unmittelbar im Browser eine Bestätigung.

Der Mail-Service arbeitet ereignisgetrieben: Nur bei Bedarf wird eine Mail erzeugt, es gibt keinen dauerhaften Hintergrundprozess außer der DB-Verbindung. Durch FastAPI's `asyncio`-basierte Architektur [158] kann der Service viele Anfragen gleichzeitig abwickeln, ohne dass der Versand einer Mail (der einige Sekunden dauern kann) den gesamten Server blockiert. In unserem Fall wird zwar `smtplib` (synchron) genutzt – was den Event Loop blockiert – doch da der zu erwartende Aufrufdurchsatz gering ist (E-Mails nur bei Registrierung, nicht ständig), wurde auf komplexere nebenläufige Auslagerung verzichtet.

5.10 Database Writer: MQTT-Datenpersistierung in PostgreSQL

Der Database Writer Service ist ein Hintergrunddienst, der eingehende Sensordaten von den Geräten entgegennimmt und diese zuverlässig in der relationalen Datenbank speichert. Er bildet damit das Bindeglied zwischen der Echtzeit-MQTT-Datenebene und der persistenten Speicherung. Aus den Anforderungen geht hervor, dass Messwerte nicht verloren gehen sollen und zeitlich historisiert abrufbar sein müssen. Daher wurde eine Lösung

implementiert, die auf nachrichtenbasierten Warteschlangen und garantierter Zustellung basiert. Der Database Writer subscribiert nicht einfach flüchtig auf MQTT-Themen, sondern nutzt den Solace-Broker mit einer persistenten Queue, um eine ausfallsichere Verarbeitung zu gewährleisten.

5.10.1 Architektur: Dauerhafter Queue-Consumer

Im Gegensatz zu den zuvor beschriebenen Webservices läuft der Database Writer ohne HTTP-Schnittstelle – er startet bei Systembeginn und läuft kontinuierlich als Daemon. Implementiert wurde er in Python unter Verwendung der Solace-eigenen Python API (`solace.messaging`), welche eine JMS-ähnliche Schnittstelle bietet. Die Hauptkomponenten sind:

1. Solace-Verbindung: Beim Start baut der Service zunächst eine Verbindung zum Solace PubSub+ Broker auf. Dafür werden die Verbindungsparameter (Host, VPN, Username, Passwort) aus Umgebungsvariablen gelesen. Im Docker-Setup zeigt z.B. `SOLACE_HOST` auf den internen Broker (`tcp://solace:55555` für non-SSL MQTT über das interne Solace-Protokoll). Der Code versucht bis zu 10 mal in einem Retry-Loop die Verbindung herzustellen, mit Wartezeit, da der Broker evtl. noch am Hochfahren ist. Dieser Mechanismus erhöht die Robustheit: sollte der Broker zum Zeitpunkt des Writer-Starts nicht bereit sein, gibt der Service nicht sofort auf, sondern wartet insgesamt bis zu 50 Sekunden auf eine erfolgreiche Verbindung.
2. Persistente Queue und Konsument: Nach Verbindungsauflauf erstellt der Service einen Consumer auf einer durablen Message-Queue namens `sensor_data`. Diese Queue ist so konfiguriert, dass sie alle relevanten Sensor-MQTT-Nachrichten aufnimmt. Die Zuordnung erfolgt über Subscriptions, die der Queue im Broker zugewiesen sind (dazu später mehr im Solace-Init Teil). Damit fungiert die Queue als Pufferspeicher: eintreffende MQTT-Publishs der Geräte werden vom Broker auf dieser Warteschlange zwischengespeichert, bis der Database Writer sie abholt. Die Verwendung einer persistenten Queue garantiert, dass keine Daten verlorengehen, selbst wenn der Consumer zwischenzeitlich ausfällt oder Netzwerkprobleme auftreten – der Broker hält die Nachrichten vor. Die Queue ist im Compose-Setup als exclusive deklariert, d.h. sie wird nur von einem Consumer genutzt, was sicherstellt, dass genau ein Service-Exemplar alle Daten chronologisch verarbeitet (kein Load Balancing hier gewünscht). Der Database Writer startet einen asynchronen Empfang auf dieser Queue mittels `receiver.receive_async(MessageHandler())`. Hier wird ein benutzerdefinierter MessageHandler (eine Klasse, die eine `on_message`-Methode

überschreibt) verwendet, was dem Entwurf eines Event-Callbacks entspricht: Jede eingehende Nachricht triggert den Aufruf von SensorMessageHandler.on_message.

3. Verarbeitung eingehender Nachrichten: Im on_message-Callback wird die erhaltene Nachricht zuerst vom proprietären Format in einen String dekodiert und dann als JSON geparsst. Die erwartete Struktur der Nachrichten – dies wurde im theoretischen Teil des Datenformats definiert – beinhaltet in der obersten Ebene eine Controller-Kennung (did) und eine Liste von Sensor-Datensätzen (sensors). Jede Sensorstruktur enthält eine Sensor-ID (sid), einen Status (status) und ggf. einen Array von Messwerten (values). Der Database Writer iteriert über alle Sensoren in der Nachricht und führt für jeden folgende Schritte aus:
 - (a) Status-Update (Heartbeat): Unabhängig davon, ob Messwerte vorliegen, wird die Information genutzt, dass ein Sensor Daten gesendet hat. Über die Hilfsfunktion update_last_call(sensor_id, status) wird in der Datenbank der letzte Meldungszeitpunkt (last_call Timestamp) und der Status des Sensors aktualisiert. Dies dient dazu, die Erreichbarkeit bzw. Aktivität von Sensoren nachzuverfolgen. Im Code wird hierbei ein frischer DB-Verbindungszyklus genutzt: update_last_call öffnet eine DB-Verbindung, führt ein UPDATE sensora.sensors SET last_call = NOW(), status = %s WHERE sid = %s, und schließt die Verbindung wieder. Der Status wird auf den vom Gerät gemeldeten Wert gesetzt (typischerweise „active“ bei normaler Meldung). Damit implementiert der Service ein Heartbeat-Monitoring: jedes Gerät signalisiert durch Senden (selbst von Messwerten) seine Aktivität.
 - (b) Messwertspeicherung: Falls der Sensor Messwerte im JSON mitgeliefert hat (values-Array nicht leer), werden diese in der Datenbankpersistiert. Hierzu ruft der Handler die Funktion save_sensor_data(sensor_id, values, controller_id) auf. Innerhalb dieser Routine findet eine detaillierte Behandlung statt:
 - i. Zunächst wird sichergestellt, dass der referenzierte Controller existiert (Datenintegrität). Dazu wird in der Tabelle sensora.controllers per SELECT geprüft, ob did = controller_id vorhanden ist. Ist dies nicht der Fall, wird ein Warnhinweis geloggt und die Speicherung für diesen Sensor abgebrochen – das System ignoriert also Messdaten von unbekannten Geräten. Im Normalfall sollten alle Controller aus dem Auth-Service bekannt sein.
 - ii. Als nächstes wird geprüft, ob der spezifische Sensor bereits in der Datenbank angelegt ist. Die Sensoren sind in der Tabelle sensora.sensors modelliert, mit Primärschlüssel sid. Falls das SELECT ergibt, dass dieser Sensor noch nicht existiert, interpretiert der Service dies als erstmalige

Meldung eines neuen Sensors an diesem Controller. In unserem Systemdesign könnten Sensoren dynamisch erkannt werden (z.B. wenn ein Controller ein neues Sensormodul bekommt). In so einem Fall legt der Database Writer automatisch einen neuen Sensor-Datensatz in der DB an. Hierfür entnimmt er der Nachricht, falls vorhanden, Meta-Informationen über den Sensor (im JSON ggf. enthalten unter „sensor_info“). Im Code wird die erste Value-Nachricht auf sensor_info geprüft und daraus z.B. der SensorTyp (ilk, z.B. „humidity“ oder „temperature“) und Einheit (unit, z.B. „%“, „°C“) extrahiert. Diese werden zusammen mit der Sensor-ID und der Controller-ID in sensora.sensors eingefügt. Dadurch wird der Sensor dem System bekannt gemacht. Wichtig: Beim Insert wird plant = NULL gesetzt, da initial der Sensor noch keiner Pflanze zugeordnet ist. Nach diesem Insert wird sofort ein commit durchgeführt, damit der neue Sensor auch in weiteren Schritten verfügbar ist. Zudem loggt das System die Anlage des Sensors.

- iii. Zuordnungsprüfung: Ein kritischer Aspekt ist, dass Messwerte nur gespeichert werden sollen, wenn klar ist, welcher Pflanze sie zugeordnet sind. Im Datenmodell hat jeder Sensor optional einen Fremdschlüssel auf sensora.plants. Direkt nach dem Insert (oder wenn Sensor schon existierte) wird daher plant_id aus dem Sensor-Datensatz ausgelesen. Ist plant_id NULL (Sensor keiner Pflanze zugeordnet), bricht die Funktion ab ohne die Messwerte zu speichern. Dieser Schritt stellt sicher, dass Daten erst dannpersistiert werden, wenn die organisatorische Verknüpfung hergestellt wurde – um zu vermeiden, dass „verwaiste“ Messwerte in der Datenbank landen, die keiner Pflanze zugeordnet sind. In der Praxis würde ein Nutzer in der Applikation also zunächst einen Sensor einer Pflanze (Topf) zuweisen, bevor Werte fließen. Nicht zugeordnete Sensoren melden zwar ihren Status (wodurch last_call aktualisiert wird), aber ihre Werte werden bis zur Zuweisung verworfen (im Code durch Log SSensor ist keiner Pflanze zugeordnet. Werte werden nicht gespeichert.“ gekennzeichnet).
- iv. Werte-Insert: Falls ein plant_id vorhanden ist, iteriert der Service über alle übermittelten Messwerte im values Array. Jeder Eintrag enthält typischerweise einen Zeitstempel (timestamp) und einen numerischen Wert (value). Wenn kein Timestamp angegeben ist, wird im Code „CURRENT_TIMESTAMP“ als Platzhalter genutzt, was die DB veranlasst, den Einfügezeitpunkt zu nehmen. Für jeden Wert generiert der Service eine eindeutige ID (vid via UUID4) und führt ein INSERT in die

Tabelle sensora.values aus. Dabei werden Wert, Timestamp, Sensor-ID und Plant-ID gespeichert. Die Verwendung einer eigenen UUID für jeden Messwert garantiert, dass Einträge eindeutig sind; alternativ hätte man ein Serien-ID der DB nutzen können – hier zeigte sich aber der Designwunsch nach verteilbar eindeutigen IDs (was in IoT-Systemen mit mehreren Quellen sinnvoll sein kann). Nach dem Schleifendurchlauf über alle Werte werden die Insertionen per conn.commit() in der DB finalisiert. Abschließend wird die DB-Verbindung geschlossen. Im Log erscheint dann pro Sensor eine Bestätigung („Alle Werte für Sensor X gespeichert.“).

- v. Fehler während der DB-Operationen werden aufgefangen und als Fehler geloggt, ohne dass der gesamte Service abstürzt. Sollte z.B. während der Inserts ein DB-Fehler auftreten, würde zwar dieser Aufruf fehlschlagen, aber der Message Handler an sich fängt die Exception und beendet nicht den Prozess (siehe nächster Punkt).
- (c) Message Acknowledgement: Nachdem alle Sensoren einer Nachricht verarbeitet wurden, bestätigt der Database Writer dem Broker den erfolgreichen Empfang mittels receiver.ack(message). Dies ist ein wichtiger Schritt im Zusammenspiel mit der persistenten Queue: Erst durch das Acknowledge wird die Nachricht aus der Queue entfernt. Sollte der Service abstürzen oder es käme zu einem nicht behandelten Fehler vor dem Ack, würde die Nachricht in der Queue bleiben und später erneut zugestellt werden können (garantierte mindestens-einmal-Zustellung). Im implementierten Code wird das Ack nur im erfolgreichen JSON-Verarbeitungsfall aufgerufen. Bei bestimmten Fehlern, z.B. JSON-Parsing-Error oder falls essentielle Felder fehlen, wird kein Ack gesendet, was bedeutet, dass die Nachricht in der Queue verbleibt. Im Log wird ein Hinweis ausgegeben („Nachricht ist kein gültiges JSON oder Ungültige Nachricht. Controller-ID fehlt.“), aber ein Ack fehlt. Dieses Verhalten könnte zu einer erneuten Zustellung führen (je nach Broker-Einstellung) oder die Nachricht blockiert die Queue. Im aktuellen Setup von Solace würde eine nicht bestätigte Nachricht in einer durable Queue verbleiben; der Consumer könnte z.B. nach einem Timeout neu gestartet werden, um es erneut zu versuchen. Hier wäre eventuell eine Verbesserung, solche Nachrichten nach x Versuchen in eine Dead Message Queue zu verschieben – jedoch ist dies im Code nicht implementiert. Somit wird sich darauf verlassen, dass gut formatierte Nachrichten ankommen. Die bewusste Entscheidung, bei unverarbeitbaren Nachrichten kein Ack zu senden, spiegelt einen Anspruch auf Datenintegrität: Lieber bleibt eine fehlerhafte Nachricht liegen (und ein Admin greift ein), als dass sie fälschlich

als verarbeitet markiert wird.

Zusätzlich zur Callback-Verarbeitung hat der Database Writer einen Nebenprozess zur Überwachung der Sensor-Aktivität: Timeout-Überprüfung: Mithilfe eines einfachen Endlosschleife-Timers im Hauptthread wird in regelmäßigen Abständen (jede Minute, gemäß

CHECK_INTERVAL = 60 Sekunden) die Funktion `check_sensor_timeouts()` ausgeführt. Diese öffnet eine DB-Verbindung und führt ein Update auf der Sensors-Tabelle aus, um alle Sensoren, deren `last_call` älter als 5 Minuten ist, auf Status 'error' zu setzen. Damit wird ein Timeout-Mechanismus realisiert: Wenn ein Sensor 5 Minuten lang keine Daten gesendet hat (und vorher auf 'active' stand), gilt er als potenziell offline oder ausgefallen. Im Datenbankmodell wird dies durch `status = 'error'` kenntlich gemacht. Dieser Mechanismus ergänzt das oben erwähnte Heartbeat-Tracking. Der Zähler wird nach jedem Lauf zurückgesetzt. Durch die Schleife mit `time.sleep(1)` wird der CPU-Verbrauch minimal gehalten.

5.11 Setpoint API: Sollwert-Vorgabe via REST und MQTT

Die Setpoint API ermöglicht es, aus dem System heraus Steuerungswerte an die Mikrocontroller zu senden – konkret Sollwerte für bestimmte Sensoren (z.B. Feuchtigkeits-Sollwert für die Bewässerungssteuerung). Damit wird das System bidirektional: Nicht nur melden Sensoren Zustände, sondern Aktoren können angesteuert werden. Der Dienst ist als kleiner Flask-basierter Webservice realisiert, der eine REST-Endpoint bereitstellt, über den ein Sollwert gesetzt werden kann. Intern publiziert der Service diesen Sollwert dann als MQTT-Nachricht auf den Broker, sodass der Ziel-Controller ihn empfängt. Im Grunde handelt es sich also um einen Protokollübergang von HTTP zu MQTT.

5.11.1 Funktionsweise und Ablauf

Der Setpoint-Service stellt den Endpoint /sollwert (HTTP POST) bereit. Der typische Ablauf, um einen Sollwert zu setzen, ist:

1. Eine externe Entität – z.B. eine Web-Frontend-Anwendung oder ein Benutzer via App – sendet einen HTTP-POST an die Setpoint API mit den Parametern:
 - (a) `controller_id`: die Kennung des Ziel-Controllers (Gerät),
 - (b) `sensor_id`: die Kennung des Sensors (bzw. des Aktors) auf diesem Gerät, für den der Sollwert gelten soll,

- (c) sollwert: der anzustrebende Wert (numerisch, z.B. Feuchtigkeit in % oder ein Schwellwert).
2. Die API prüft eingehend, ob alle nötigen Felder vorhanden und gültig sind. Falls etwas fehlt, wird mit HTTP 400 Bad Request geantwortet und ein Fehlerjson zurückgegeben.
 3. Ist die Eingabe valide, generiert der Service eine MQTT-Nachricht. Dazu wird ein JSON-Objekt erstellt, das wie folgt aussieht:

```

1   {
2     "targetValues": [
3       {
4         "did": "<controller_id>",
5         "sid": "<sensor_id>",
6         "value": <sollwert>
7       }
8     ]
9   }

```

Diese Struktur ist angelehnt an das Format, das ggf. vom Gerät erwartet wird (eine Liste von Zielwerten für bestimmte Sensoren/Aktoren). Es wird also der Device-ID und Sensor-ID nochmals eingebettet, damit das Gerät die Nachricht zuordnen kann.

4. Als nächstes bestimmt der Service das Ziel-Topic für die MQTT-Nachricht. Gemäß der in Auth-Service eingerichteten Konvention wird das Topic `sensora/v1/receive/<controller_id>/targetValues` genutzt. Darauf ist der betreffende Controller (bzw. dessen MQTT-Client) berechtigt zu lauschen. Dieses Topic adressiert somit genau den gewünschten Controller.
5. Der Service publiziert die Nachricht über den Solace-Broker: Hierzu nutzt er die vorab aufgebaute Broker-Verbindung und einen Publisher, der als persistent message publisher initialisiert wurde. Die Nachricht wird mit dem oben genannten Topic abgesendet. Der Broker sorgt dann dafür, dass – sofern der Controller online ist und das Topic abonniert hat – die Nachricht an diesen zugestellt wird. Sollte der Controller momentan nicht verbunden sein, greift je nach Broker-Einstellung die Persistierung: Entweder wurde eine Queue für solche Sollwert-Nachrichten eingerichtet (vgl. mögliches `sensor_setpoints` Queue), oder im MQTT-Kontext übernimmt der Broker das Speichern bei QoS>0. Da der Publisher hier als "persistent" konfiguriert ist, lässt sich ableiten, dass die Nachricht als durable versendet wird, was in MQTT-Terminologie etwa QoS 1 entspricht (mindestens einmal Zustellen). Solace bietet für MQTT-Clients mit dauerhafter Session auch an, solche Nachrichten zwischenspeichern, was vermutlich hier genutzt wird.

6. Die Setpoint API gibt dem HTTP-Aufrufer eine Erfolgsmeldung zurück (HTTP 200, JSON {“status: “success”}). Damit ist der Vorgang für den Benutzer abgeschlossen. Im Hintergrund allerdings wird nun der Controller die Nachricht empfangen und z.B. seine Konfiguration anpassen (dies ist Teil der Geräte-Firmware und außerhalb dieses Service-Scopes, aber essentiell für den Regelkreis der Bewässerung).

5.12 Solace Init: Automatisierte Broker-Konfiguration

Der Solace Init Service (bzw. Skript) wurde implementiert, um bei Start des Gesamtsystems sicherzustellen, dass der Solace-Broker über alle notwendigen Persistent Queues und Topic-Weiterleitungen verfügt. Er stellt somit eine Infrastruktur-Komponente dar, die eng mit dem Broker zusammenarbeitet. Da in einem Container-Setup der Broker beim ersten Start völlig jungfräulich ist, muss z.B. die sensor_data Queue angelegt und mit dem entsprechenden Topic verbunden werden. Solace Init erfüllt genau diese Aufgabe über die SEMP v2 Management-API.

5.12.1 Vorgehen und Konfiguration

Solace Init ist als eigenständiges Python-Skript (`init.py`) konzipiert, das beim Hochfahren des Docker-Compose Stacks einmalig ausgeführt wird und sich danach beendet. Seine Aufgaben sind:

1. Einlesen der gewünschten Queue-Konfiguration: Aus einer JSON-Datei (im Code queues.json) werden alle zu erstellenden Queues und ihre Eigenschaften/Subscriptions geladen. Dieses File definiert quasi deklarativ, welche Warteschlangen mit welchen Parametern und Abos existieren sollen. Ein beispielhafter Inhalt könnte so aussehen:

```
1
2 {
3     "queues": [
4         {
5             "queueName": "sensor_data",
6             "accessType": "exclusive",
7             "egressEnabled": true,
8             "ingressEnabled": true,
9             "subscriptions": [
10                 { "subscriptionTopic": "sensorsa/v1/send/>" }
11             ]
12         },
13         {
14             "queueName": "sensor_setpoints",
15             "accessType": "exclusive",
16             "subscriptions": [
17                 { "subscriptionTopic": "sensorsa/v1/receive/>" }
18             ]
19         }
20     ]
21 }
```

```

18     ]
19   }
20   ]
21 }
```

Hier würde z.B. festgelegt, dass es eine Queue sensor_data gibt, die exklusiv ist und sowohl Ingress/Egress aktiviert hat (Standard für persistent Queues), und dass sie alle Topics unter sensora/v1/send abonniert (der > Wildcard deckt alle Unterpfade ab). Ebenso eine Queue sensor_setpoints für alle sensora/v1/receive Nachrichten. Dieses Konstrukt deckt die zuvor diskutierten Pfade ab: Messwerte und Steuerbefehle.

2. Verbindung zur SEMP API: Solace Init nutzt Python requests, um HTTP-POSTs an den Broker zu senden. Die notwendigen Zugangsdaten (Admin-User, Passwort, Host:Port) werden aus Env-Variablen bezogen (SOLACE_USER, SOLACE_PASS, SOLACE_HOST_SEMP). Im Docker Compose sieht man, dass solace-init Container mit depends_on: solace gestartet wird und erst nach 80s (wenn Broker läuft) das Skript ausführt. So ist sichergestellt, dass der Broker Management-Port 8080 erreichbar ist. SEMP v2 erfordert Basic-Auth mit dem Admin-Login[159], was hier als Tuple im requests.post(..., auth=(user, pass)) genutzt wird.
3. Erstellen von Queues (SEMP /config): Für jede in der JSON gelistete Queue baut das Skript die entsprechende URL und das Payload zusammen. Beispiel: POST http://solace:8080/SEMP/v2/config/msgVpns/default/queues mit JSON

```

1 {"queueName": "sensor_data", "accessType": "exclusive", "egressEnabled": true,
...}
```

. Der Broker antwortet mit Status 201 (Created) bei Erfolg. Das Skript prüft den Statuscode:

- (a) 200/201 wird als Erfolg gewertet und entsprechend geloggt ("Queue erfolgreich erstellt").
- (b) 409 (Conflict) bedeutet, die Queue existiert bereits – in diesem Fall loggt das Skript eine Warnung, fährt aber fort. Das ist gewollt, um Idempotenz zu erreichen: Sollte man Solace Init versehentlich zweimal ausführen oder den Broker bereits vorkonfiguriert haben, bricht es nicht ab, sondern erkennt vorhandene Entities.
- (c) Andere Fehlercode führt zu einem Fehlerausdruck und Rückkehr aus der Funktion (somit würde die Subscription-Anlage für diese Queue übersprungen).

4. Hinzufügen von Topic-Subscriptions: Nach dem Anlegen (oder Erkennen) der Queue iteriert das Skript über alle vorgesehenen Subscription-Themen und sendet für jedes ein POST .../queues/{queueName}/subscriptions mit {“subscriptionTopic”: “<topic>”}. Hier gelten ähnliche Statuscodes:
 - (a) 200/201: Subscription hinzugefügt (Logausgabe),
 - (b) 409: bereits vorhanden (Warnhinweis, aber kein Abbruch),
 - (c) andere: Fehler ausgeben.
5. Abschluss: Nachdem alle Queues aus der Liste abgearbeitet sind, gibt das Skript eine Meldung aus, dass alle Queues und Subscriptions verarbeitet wurden, und endet.

5.13 Programmierung des ESP

Die Entwicklung eingebetteter Systeme erfordert eine präzise Abstimmung zwischen hardwarenahem Verhalten und übergeordneten Steuerungsprozessen. Insbesondere bei IoT-Anwendungen, wie dem hier vorgestellten automatisierten Bewässerungssystem, kommt der softwareseitigen Ablaufsteuerung des Mikrocontrollers eine zentrale Rolle zu.

Die Funktionalität des Systems basiert auf dem koordinierten Zusammenspiel von Sensorknik, drahtloser Netzwerkkommunikation, Aktorsteuerung und Datensynchronisation mit einem externen Backend.

Der eingesetzte Mikrocontroller **ESP32-WROOM-32D** erweist sich aufgrund seiner integrierten WLAN-Funktionalität, Mehrkernarchitektur und Echtzeitfähigkeit als besonders geeignet für diese Anwendung.

Zudem bietet das zugrunde liegende ESP-IDF-Framework eine modulare und hardwarenahe Entwicklungsumgebung, die eine präzise Steuerung und effiziente Ressourcenverwaltung ermöglicht.

Dieses Kapitel behandelt die programmtechnische Umsetzung der Systemlogik auf dem verwendeten Mikrocontroller (ESP32-WROOM-32D), wobei die eingesetzte Softwarearchitektur auf dem ESP-IDF-Framework basiert. Der Fokus liegt auf der Initialisierungslogik, der zyklischen Erfassung und Verarbeitung von Sensordaten, der Steuerung der Pumpe sowie der Einbindung in eine gesicherte MQTT-Kommunikationsstruktur. Die Interaktion mit dem Benutzer erfolgt sowohl über ein lokales Webinterface als auch über hardwareseitige Bedienelemente.

Insgesamt lässt sich die Entwicklung in zwei Teilbereiche unterteilen: Die Initialisierung des Mikrocontrollers und der laufende Betrieb. Diese beiden Bereiche sind eng miteinander verknüpft, da die Initialisierung die Grundlage für alle nachfolgenden Prozesse bildet. Ziel dieses Kapitels ist es, die internen Abläufe des Mikrocontrollers in strukturierter

Form darzustellen, um sowohl die logische Abfolge als auch die funktionalen Abhängigkeiten nachvollziehbar zu machen. Die beiden Teilbereiche werden jeweils in ihrem eigenen Kapitel genauer betrachtet. Dazu wird zunächst ein abstrahierter Ablaufplan vorgestellt, welcher die zentralen Prozessschritte visuell abbildet. Dabei stellt ein je ein Eingefärbter Bereich eine Funktion dar, welche durch das erste Element des Bereichs angegeben wird. Die darauffolgenden Elemente werden innerhalb dieser Funktion ausgeführt.

Im Anschluss erfolgt eine schrittweise Low-Level-Beschreibung der internen Ablauflogik auf Basis des realisierten Programmcodes.

5.13.1 Initialisierung des ESP

Die Initialisierungslogik des Mikrocontrollers stellt den operativen Ausgangspunkt für den gesamten Lebenszyklus des automatisierten Bewässerungssystems dar. Unmittelbar nach dem Systemstart des ESP32 übernimmt sie die Aufgabe, einen konsistenten und vollständig betriebsfähigen Ausgangszustand herzustellen. Dazu gehört insbesondere die Inbetriebnahme der netzwerkbasierter Kommunikationsschnittstellen, die Einbindung in den Authentifizierungsprozess mit dem Backend, die Bereitstellung eines Benutzerinterfaces zur Konfiguration sowie die Initialisierung interner Systemdaten.

Die Initialisierung ist in der Funktion `device_init()` im Modul `device_manager.c` implementiert und wird aus dem zentralen Einstiegspunkt `app_main()` in `main.c` aufgerufen. Erst wenn alle Voraussetzungen – insbesondere eine erfolgreiche WLAN-Verbindung und eine gültige Registrierung am Backend – erfüllt sind, gilt die Initialisierungsphase als abgeschlossen. Der Status wird über die Funktion `device_init_done()` überwacht, sodass nachfolgende Systemkomponenten (z. B. Sensorik, Zeitsynchronisation, MQTT-Kommunikation) erst im vollständig initialisierten Zustand gestartet werden.

Darüber hinaus umfasst die Initialisierung eine robuste Reset-Funktionalität, die es ermöglicht, den Mikrocontroller über einen dedizierten GPIO-Pin manuell auf Werkseinstellungen zurückzusetzen. Eine zeitsensitive Tasterauswertung mit visueller Rückmeldung über die Status-LED garantiert dabei die absichtliche Auslösung. Zusätzlich ist die gesamte Netzwerkinitialisierung fehlertolerant gestaltet: Sollte ein Verbindungsversuch zur hinterlegten WLAN-Infrastruktur wiederholt fehlschlagen, wechselt das System automatisch in den Konfigurationsmodus (SoftAP) und stellt dem Benutzer ein lokales Webinterface zur Verfügung. Auf diese Weise wird eine autonome Wiederherstellung der Konnektivität ohne externe Eingriffe ermöglicht.

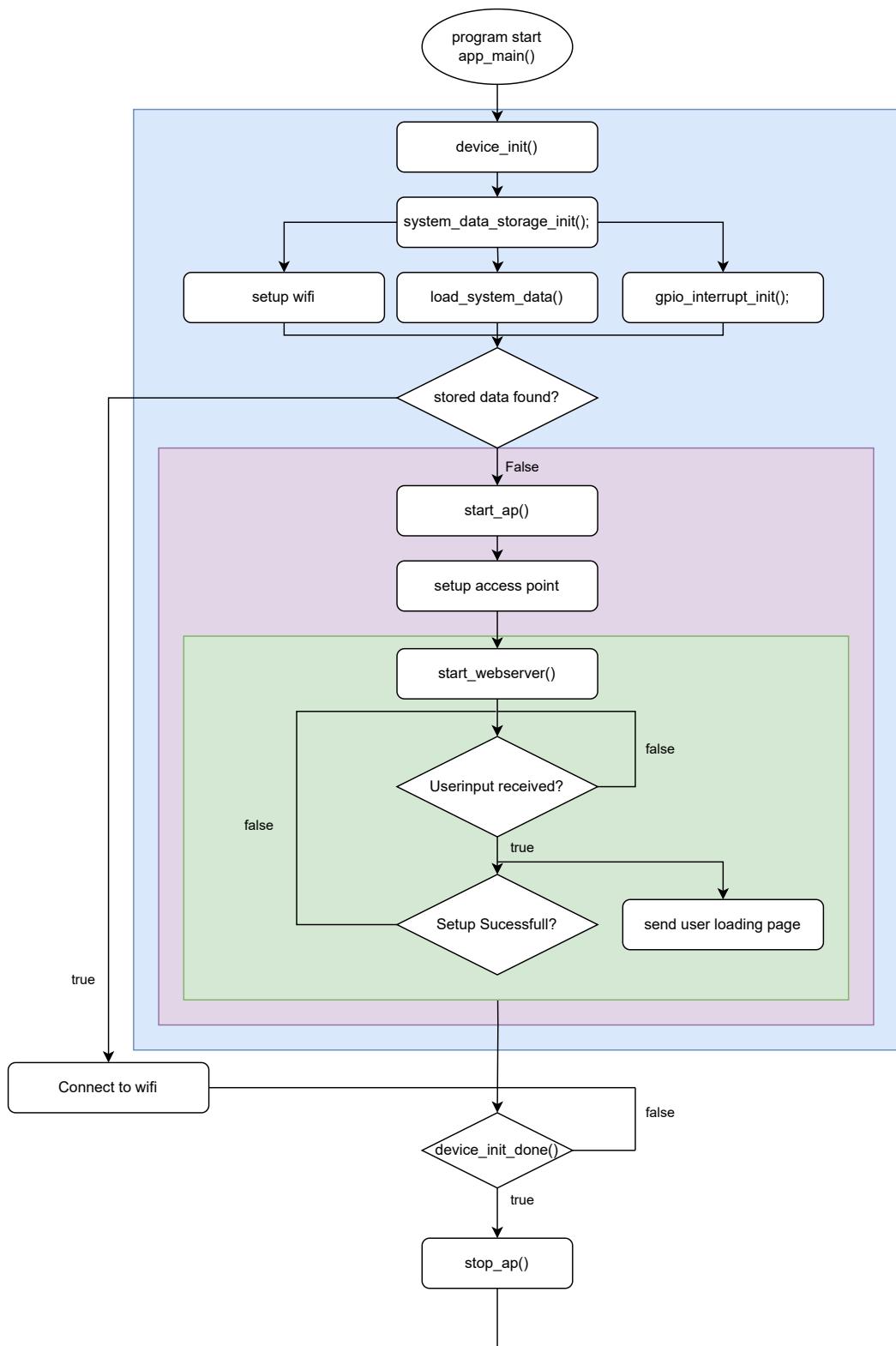


Abbildung 5.9: Ablaufplan Initialisierung des ESP

Initialisierung des persistenten Speichers (NVS):

Der erste Schritt der Initialisierung besteht in der Aktivierung des nichtflüchtigen Speichers. Hierbei wird der sogenannte Non-Volatile Storage (NVS) über die Funktion `nvs_flash_init()` initialisiert. Dieser dient als zentrale, dauerhaft verfügbare Speicherstruktur zur Ablage von Konfigurationsdaten wie WLAN-Zugang, Authentifizierungstoken oder Zielwerten der Sensorik. Sollte der Speicher beschädigt sein oder durch ein Versionsupdate inkonsistent geworden sein, wird dies über die Rückgabecodes

`ESP_ERR_NVS_NO_FREE_PAGES` oder `ESP_ERR_NVS_NEW_VERSION_FOUND` erkannt.

In diesem Fall wird über `nvs_flash_erase()` ein vollständiger Reset des Speichers durchgeführt. Die Architektur ist damit gegen veraltete oder inkonsistente Daten abgesichert und stellt eine fehlerfreie Persistenzlogik sicher.

Laden und Strukturierung der Systemdaten:

Nach erfolgreicher NVS-Initialisierung wird über `load_system_data()` versucht, die Konfigurationsstruktur `system_data_t` aus dem Speicher zu laden. Diese Struktur enthält die meisten betriebskritischen Parameter, darunter:

- Sensors-Nutzername,
- MQTT-Zugangsdaten (Benutzername, Passwort, Topics),
- Controller-Identifikation (ID, Modell),
- Sensor-Zielwerte (z. B. Bodenfeuchte, Temperatur, Luftfeuchte, Licht),
- Token für die Authentifizierung (Hardware und Software),
- Registrierungsstatus.

Ist der Speicherzugriff erfolgreich und die Daten konsistent, werden sie in einer lokalen Instanz gespeichert und stehen so für weitere Verwendung zur Verfügung. Andernfalls bleibt das System im Zustand „unregistriert“ und leitet automatisch in den Setup-Modus über.

Zusätzlich wird versucht die WLAN-Daten aus dem NVS zu laden. Diese sind allerdings direkt vom `esp_wifi` Modul verwaltet und sind daher nicht Bestandteil der `system_data_t`-Struktur. Die WLAN-Daten werden über die Funktion `esp_wifi_get_config()` geladen und in einer lokalen Instanz gespeichert. Diese Daten sind für die Registrierung des Geräts am Backend erforderlich und werden im weiteren Verlauf benötigt. Sind sie nicht vorhanden so wird das System ebenfalls in den Setup-Modus versetzt.

Initialisierung der Status-LED:

Die Status-LED dient als primäres Feedbackinstrument für den Benutzer. Die LED ist über GPIO 2 angebunden und wird über das Modul `led_control.c` gesteuert. Nach `led_init()` ist der Pin als Ausgang initialisiert und standardmäßig deaktiviert. Die Logik unterscheidet mehrere Zustände:

- `led_blink_start()`: SoftAP-Modus aktiv,
- `led_on()`: System betriebsbereit und registriert,
- `led_off()`: Fehlerzustand oder Resetphase.

Die Blinklogik ist als separate FreeRTOS-Task realisiert und damit unabhängig vom Hauptprogrammzyklus ausführbar, was insbesondere für Interrupt-gesteuerte Prozesse wie den Reset-Button entscheidend ist.

Die Aufruf der `led_init()` erfolgt direkt nach der NVS-Initialisierung innerhalb der `device_init()`. Dies stellt sicher, dass die LED-Logik auch bei einem Reset des NVS sofort verfügbar ist.

Einrichtung des Hardware-Resets via GPIO:

Um einen vollständigen Werksreset zu ermöglichen, wurde GPIO 0 als Eingangsquelle mit Pull-Up-Widerstand und fallender Flanke konfiguriert. Wird der Taster gedrückt, löst die Interrupt-Service-Routine `gpio_isr_handler()` eine Notification aus, die an die Task `reset_task()` übergeben wird. Diese prüft, ob der Taster mindestens fünf Sekunden lang gedrückt bleibt. Während der Haltezeit blinkt die LED, um visuelles Feedback zu geben. Wird die Haltezeit überschritten, erfolgt:

1. Stoppen des Wi-Fi-Treibers,
2. Zurücksetzen der WLAN-Konfiguration (`esp_wifi_restore()`),
3. Löschen aller Systemdaten aus dem NVS (`erase_system_data()`),
4. Neustart des Controllers (`esp_restart()`).

Diese robuste, mehrstufige Logik verhindert unbeabsichtigte Rücksetzungen und garantiert eine sichere Wiederherstellung des Auslieferungszustands.

Netzwerkstack und Eventsystem:

Die nächsten Initialisierungsschritte betreffen die Netzwerkkommunikation. Über `esp_netif_init()` wird die TCP/IP-Stack-Grundstruktur geladen. Anschließend wird ein zentraler Event-Loop mit `esp_event_loop_create_default()` erstellt, der später für alle netzwerkbezogenen Ereignisse zuständig ist. Zwei Interfaces werden parallel konfiguriert:

- `esp_netif_create_default_wifi_sta()` für den Station-Modus,
- `esp_netif_create_default_wifi_ap()` für den SoftAP-Modus.

Durch die parallele Konfiguration kann der Controller sowohl als Client im Heimnetzwerk als auch als Access Point zur Erstkonfiguration agieren.

Wi-Fi-Initialisierung und Event-Handler:

Das Wi-Fi-Modul wird über `esp_wifi_init()` mit einer Default-Konfiguration aktiviert. Die Registrierung von Event-Handlern erfolgt für zwei Eventtypen:

- `WIFI_EVENT`: Reaktion auf Verbindungsstart, -abbruch und Authentifizierungsfehler.
- `IP_EVENT_STA_GOT_IP`: Signalisiert erfolgreiche IP-Vergabe im Station-Modus.

Sobald eine Verbindung hergestellt ist und eine IP-Adresse bezogen wurde, wird der Status in der LED reflektiert. Bei mehrfachen Verbindungsabbrüchen (>5) wird automatisch in den AP-Modus zurückgeschaltet, um einen Konfigurationsfehler beheben zu können.

Entscheidung über den Betriebsmodus (STA vs. SoftAP):

Die Entscheidung, ob das Gerät im Betriebsmodus (WLAN-Client) oder Setup-Modus (Access Point) starten soll, basiert auf den geladenen Systemdaten. Sind sowohl WLAN-Zugangsdaten als auch ein positiver Registrierungsstatus im NVS gespeichert, erfolgt der Start im STA-Modus. Der Controller versucht über `esp_wifi_set_mode(WIFI_MODE_STA)` und `esp_wifi_start()` eine Verbindung aufzubauen. Andernfalls startet das System im SoftAP-Modus über `start_ap()`.

SoftAP-Modus und Webserver zur Konfiguration:

Im Setup-Modus stellt das System einen offenen WLAN-Access-Point mit dem Namen „Sensora“ bereit. Parallel dazu wird ein HTTP-Webserver gestartet, der ein in `wifi_setup_html.h` eingebettetes Formular zur Verfügung stellt. Hier können SSID, Passwort und Benutzername eingegeben werden. Nach der Formulareinsendung wird über

`esp_wifi_set_config()` die neue Konfiguration übernommen und ein Verbindungsversuch gestartet. Eine JavaScript-basierte Statusabfrage im Browser informiert über Fortschritt und Fehlerzustände (z. B. keine Verbindung oder Backend nicht erreichbar). Die Benutzereingaben werden in der Systemstruktur gespeichert und sind ab diesem Moment Bestandteil der persistierten Konfiguration.

Geräteregistrierung über den Authentifizierungsdienst:

Nach erfolgreichem Verbindungsauflauf über die im Webinterface eingegebenen WLAN-Daten wird die Registrierung des Geräts ausgelöst. Diese erfolgt über die Funktion `register_device()` im Modul `auth_service.c`. Ziel des Prozesses ist es, den Controller eindeutig beim zentralen Authentifizierungsserver zu identifizieren und verschlüsselte Zugangsdaten für den MQTT-Broker zu erhalten. Die Logik basiert auf einem mehrstufigen Challenge-Response-Verfahren mit HMAC-gesicherter Kommunikation und symmetrischer Fernet-Verschlüsselung (basierend auf AES-128 im CBC-Modus).

Die Registrierung umfasst folgende Schritte:

1. Erstellung eines HMAC-basierten Token-Hashes aus Hardware- und Software-Token.
2. Senden des Hashes zusammen mit dem Benutzernamen an die API `/api/controller/init`.
3. Empfang einer zufälligen Challenge vom Server (Base64-codiert).
4. Berechnung einer HMAC-Response auf Basis der Challenge und des Hardware-Tokens.
5. Antwort über die API `/api/controller/verify`, zusammen mit dem ursprünglichen Token-Hash.
6. Erhalt eines Fernet-codierten Credential-Tokens, das folgende Informationen enthält:
 - MQTT-Benutzername und Passwort,
 - Controller-ID und Modell,
 - Publish- und Subscribe-Topics,
 - Broker-URL, Port und SSL-Informationen.

Zur Validierung der Authentizität des Tokens sendet der Server zusätzlich einen sogenannten `credential_key`, der lokal gegen einen HMAC-Vergleichswert verifiziert wird. Erst wenn diese Validierung erfolgreich ist, beginnt die Entschlüsselung des Tokens.

Manuelle Implementierung der Fernet-Entschlüsselung:

Da im ESP32/ESP-IDF-Ökosystem keine native Unterstützung für Fernet besteht, wurde die gesamte Entschlüsselungslogik auf Byte-Ebene manuell umgesetzt. Diese umfasst:

- Umwandlung von URL-safe Base64 in reguläres Base64 (inkl. Padding-Korrektur),
- Decodierung des Tokens mit `mbedtls_base64_decode()`,
- Extraktion von Version, Timestamp, IV, Ciphertext und HMAC aus dem dekodierten Token,
- HMAC-Validierung des Tokens mit dem empfangenen Schlüssel zur Integritätsprüfung,
- Entschlüsselung des Payloads mit AES-128 im CBC-Modus über `mbedtls_aes_crypt_cbc()`,
- Entfernung und Validierung des PKCS#7-Paddings.

Diese Implementierung stellte hohe Anforderungen an Genauigkeit im Umgang mit Speicherpuffern, Byte-Reihenfolgen und Formatierungsvorgaben. Zur Validierung wurden sämtliche Zwischenstände gegen Python-Referenzwerte getestet. Fehler in Padding-Handling oder IV-Länge führten während der Entwicklung regelmäßig zu Speicherverletzungen oder inkorrekten Dekodierungen. Dadurch wurde die Notwendigkeit einer präzisen und robusten Implementierung deutlich. Die gesamte Logik ist in der Funktion `decrypt_fernet_token()` im Modul `auth_service.c` zusammengefasst.

Speicherung und Abschluss der Registrierung:

Nach erfolgreicher Entschlüsselung werden alle empfangenen Felder aus dem Credential-Payload in die Struktur `system_data_t` überführt. Zusätzlich wird der Registrierungsstatus auf `true` gesetzt. Der gesamte Datenblock wird abschließend über `save_system_data()` dauerhaft im Flash gespeichert. Temporäre Speicherbereiche für MQTT-Daten und URLs werden aus dem Heap freigegeben. Dieser Mechanismus sichert sowohl Speicherstabilität als auch Datenintegrität und bildet die Basis für alle folgenden Kommunikationsprozesse.

Abschluss der Initialisierung durch Deaktivierung des Setup-Modus:

Sobald Registrierung und Netzwerkverbindung erfolgreich abgeschlossen sind, endet eine while Schleife in `app_main()`, da die Funktion `device_init_done` nun `true` returned. Danach wird der Setup-Modus beendet. Hierzu erfolgt im Hauptprogramm `app_main()` ein

expliziter Aufruf von `stop_ap()`. Diese Funktion schaltet den Access Point ab, stoppt den HTTP-Webserver und wechselt den Betriebsmodus von `WIFI_MODE_APSTA` nach `WIFI_MODE_STA`. Dabei wird sichergestellt, dass keine Verbindungsversuche mehr mit dem offenen Netzwerk erfolgen und alle Schnittstellen auf eine abgesicherte Betriebsumgebung migriert werden. Ein typischer Ablauf von `stop_ap()` umfasst:

1. Überprüfung des aktuellen Betriebsmodus,
2. Abschaltung des Webservers mit `httpd_stop()`,
3. Wechsel in den reinen STA-Modus mittels `esp_wifi_set_mode()`,
4. Soft-Restart des Wi-Fi-Treibers zur Sicherstellung eines sauberen Zustands,
5. Reconnect-Versuch zur bekannten SSID bei verlorener Verbindung (optional),
6. LED-Signalisierung durch Stoppen des Blinkens und dauerhaftes Einschalten.

Diese Umstellung ist notwendig, da der SoftAP-Modus als temporäres Konfigurationsmittel konzipiert ist und im Normalbetrieb ein Sicherheitsrisiko darstellen würde.

Abschlussprüfung der Initialisierung:

Nach Durchführung aller Schritte verbleibt das System im verbundenen, registrierten Zustand und ist bereit zur Ausführung der nachgelagerten Subsysteme. Der Status wird über die Funktion `device_init_done()` regelmäßig geprüft. Diese liefert den Wert `true`, sobald sowohl eine gültige IP-Adresse bezogen wurde (`is_sta_connected()`) als auch der `registered`-Status in `system_data_t` auf `true` gesetzt ist.

Erst wenn diese Bedingung erfüllt ist, werden in `app_main()` die weiteren Initialisierungen durchgeführt:

- Zeitsynchronisation mit SNTP über `initialize_sntp()` und `wait_for_time_sync()`,
- Initialisierung der MQTT-Verbindung über `solace_init()`,
- Start der Sensorik und Bewässerungslogik.

Der Initialisierungsprozess ist damit abgeschlossen. Das System befindet sich fortan im überwachten Betriebsmodus und kann vollständig autonom auf externe Kon

5.13.2 Regelbetrieb des ESP

Zentral bei der Umsetzung des automatisierten Bewässerungssystems ist der ESP32-WROOM-32D Mikrocontroller, welcher die Interaktion zwischen Sensorik, Aktorik und Backend-Systemen steuert. Dabei stellt dieser ESP32 eine umfangreichen GPIO-Ausstattung, welche den Anschluss verschiedenster Peripheriekomponenten ermöglicht.

Im Anschluss an die ESP-Initialisierung, die mittels Wifi-Provisioning sowie Authentifizierung gegen den Backend-Service die Konnektivität sicherstellt, beginnt der zyklische Programmablauf, welcher in verschiedene, klar voneinander getrennte Komponenten untergliedert ist. Diese sind modular aufgebaut und folgen einem Event-Loop-Modell, wie es in Echtzeitbetriebssystemen (RTOS) wie FreeRTOS üblich ist. Die zentrale Steuerung erfolgt durch Tasks, welche Sensorwerte erfassen, verarbeiten, und basierend darauf Aktionen wie etwa die Aktivierung der Pumpe auslösen.

Zur Erfassung der Umweltbedingungen kommen drei Sensoren zum Einsatz: ein *GY-302 BH1750* zur Messung der Beleuchtungsstärke, ein *DHT11*-Sensor zur Ermittlung von Temperatur und Luftfeuchtigkeit sowie ein kapazitiver Bodenfeuchtigkeitssensor (*HiLet-go LM393*). Die Ausgabe der gesammelten Sensordaten erfolgt in periodischen Intervallen über MQTT an eine Solace PubSub Instanz, die über ein Docker-Container-System bereitgestellt wird.

Im Folgenden wird jeder Abschnitt des abstrahierten Ablaufplans des ESP-Regelbetriebs, zu sehen in Abbildung 5.10 auf der nächsten Seite, technisch und funktional erläutert.

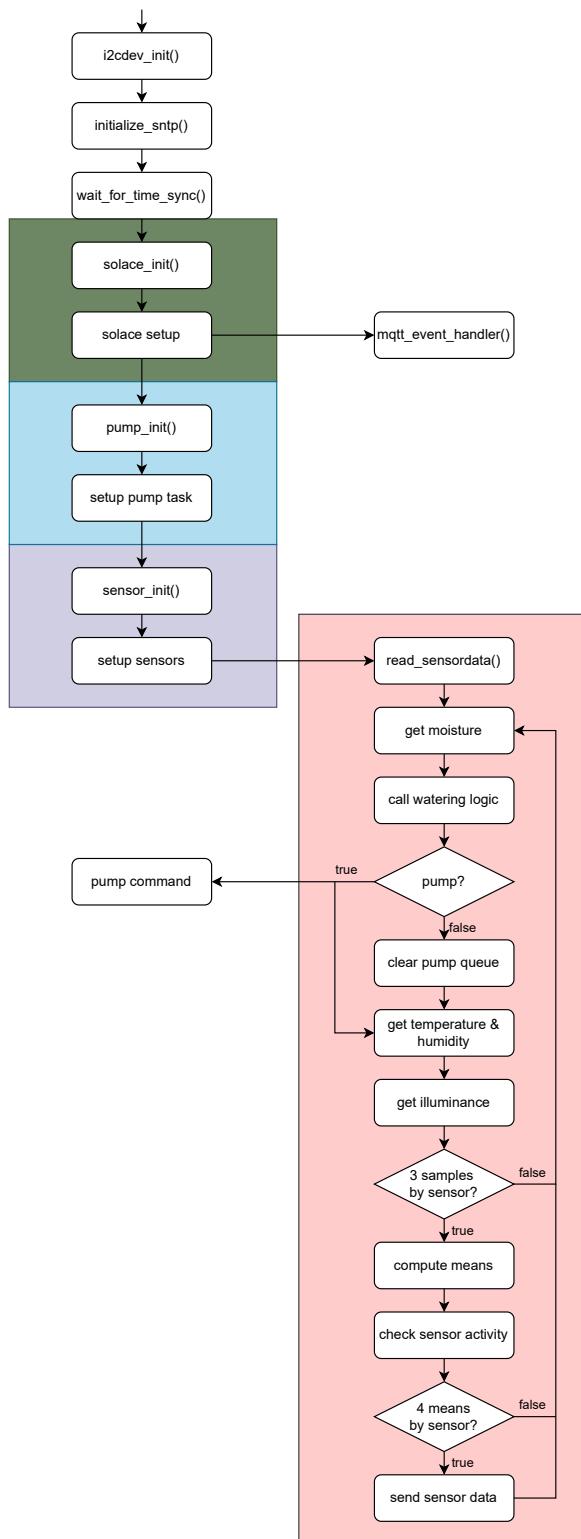


Abbildung 5.10: Ablaufplan Regelbetrieb des ESP

Systemvorbereitungen: I²C-Stack und Zeitdienst

Nach Beendigung der initialen Gerätekonfiguration und Abschaltung des Access Points (`stop_ap()`) beginnt der ESP32 im Regelbetrieb mit einer Reihe vorbereitender Initialisierungen, die für die korrekte Funktion der Sensorik und Kommunikation essenziell sind. Im Ablaufplan (Abbildung 5.10 auf der vorherigen Seite) ist dieser Schritt durch die folgenden Funktionsaufrufe dargestellt:

- `i2cdev_init()`
- `initialize_sntp()`
- `wait_for_time_sync()`

Die erste vorbereitende Maßnahme ist die Initialisierung des I²C-Stacks. Die Funktion `i2cdev_init()` stammt aus einer externen Bibliothek und abstrahiert die Konfiguration des I²C-Busses, welcher für die Kommunikation mit dem BH1750-Lichtsensor erforderlich ist. Der Sensor wird über die standardisierte I²C-Schnittstelle betrieben und benötigt einen initialisierten Treiberstack, bevor Lese- oder Schreibzugriffe erfolgen können. Im Falle eines Fehlers bei der Initialisierung wird der Systemstart abgebrochen und eine entsprechende Fehlermeldung ausgegeben. Diese Maßnahme dient der Robustheit und stellt sicher, dass keine fehlerhaften Sensordaten erzeugt werden, die zu falschen Bewässerungsentscheidungen führen könnten.

Im Anschluss an die I²C-Vorbereitung erfolgt die Synchronisierung der Systemzeit über das Simple Network Time Protocol (SNTP). Die korrekte Zeit ist essenziell, da alle Sensordaten, die später an das Backend übertragen werden, mit Zeitstempeln versehen werden. Diese Zeitstempel sind wiederum entscheidend für die korrekte Interpretation und Verarbeitung im Backend, insbesondere in Systemen, die auf Zeitreihenanalysen oder historischen Vergleichswerten basieren.

Die Funktion `initialize_sntp()` konfiguriert einen SNTP-Client im Polling-Modus und setzt den verwendeten Zeitserver (standardmäßig `pool.ntp.org`). Anschließend wird in der Funktion `wait_for_time_sync()` geprüft, ob eine gültige Systemzeit empfangen wurde. Diese Überprüfung erfolgt durch wiederholtes Abfragen der lokalen Zeitinformationen. Der Ablauf ist so gestaltet, dass der ESP32 so lange blockiert, bis eine Zeit verfügbar ist, deren Jahr größer als 2016 ist – dies dient als Heuristik für eine erfolgreiche Synchronisierung.

Diese Zeitsynchronisation ist ein zentrales Qualitätsmerkmal des Systems, da sie nicht nur zur Zeitstempelung der Messwerte dient, sondern auch für die Vergleichbarkeit mit Zielwerten im Backend sowie für langfristige Protokollierungen notwendig ist. Eine falsch

synchronisierte Uhr würde beispielsweise zu verzerrten Zeitreihen führen und könnte fehlerhafte Rückschlüsse auf das Verhalten der Pflanzen oder das Klimaregime erlauben. Insgesamt dienen die beschriebenen Systemvorbereitungen also sowohl der Sicherstellung einer funktionierenden Peripheriekommunikation (über I²C) als auch der korrekten zeitlichen Einordnung aller Mess- und Steuerdaten (über SNTP). Beide Schritte bilden somit eine essenzielle Grundlage für den nachfolgenden zyklischen Ablauf im Regelbetrieb.

MQTT-Kommunikationsclient

Nach Abschluss der systemnahen Initialisierungen folgt im Regelbetrieb die Anbindung an die Backend-Infrastruktur über das MQTT-Protokoll. Dies erfolgt durch den Aufruf der Funktion `solace_init()`, welche im Ablaufplan (Abbildung 5.10 auf Seite 180) das „solace setup“ sowie den `mqtt_event_handler()` anstößt.

Die MQTT-Kommunikation basiert in diesem System auf dem Publish-Subscribe-Modell und erfolgt über eine Solace PubSub Instanz, welche als Docker-Container bereitgestellt wird. Der ESP32 agiert dabei als Publisher und Subscriber zugleich: Er veröffentlicht periodisch aggregierte Sensordaten und empfängt gleichzeitig neue Zielwerte zur Parametrisierung der Regelstrategie (z. B. Soll-Bodenfeuchtigkeitswerte).

Im Rahmen der `solace_init()`-Funktion werden zunächst die aus dem nichtflüchtigen Speicher geladenen Verbindungsparameter wie Benutzername, Passwort, Client-ID sowie die spezifischen Topics konfiguriert. Anschließend wird der MQTT-Client initialisiert und mit dem Broker verbunden. Die Verbindung wird dabei so konfiguriert, dass keine Session-Daten zurückgesetzt werden (`disable_clean_session = true`), was eine persistente Kommunikationsbeziehung zwischen Controller und Broker ermöglicht.

Nach erfolgreicher Verbindung registriert der Client einen Event-Handler, der unter anderem auf das Eintreffen neuer Datenpakete reagiert. Ein zentrales Element dieser Kommunikation ist die Funktion `send_message()` (zu sehen unter Listing 5.2), welche zum Versand der aggregierten Sensordaten dient. Diese Funktion nimmt eine bereits als JSON-formatierte Zeichenkette entgegen und überträgt diese mithilfe des MQTT-Clients auf das konfigurierte Topic:

```

1 void send_message(const char *message) {
2     if (message != NULL) {
3         esp_mqtt_client_enqueue(client, info.solace_publish_topic, message, 0, 1, 1,
4                                 true);
5     }
}
```

Listing 5.2: Senden der Sensordaten

Die Nutzung der Funktion `esp_mqtt_client_enqueue()` ermöglicht dabei eine nicht-blockierende Übertragung, bei der die Nachricht in die interne Queue des MQTT-Stacks geschrieben wird. Durch das Setzen der QoS-Stufe (Quality of Service) auf 1 wird sichergestellt, dass die Nachricht mindestens einmal zuverlässig zugestellt wird – ein angemessener Kompromiss zwischen Zuverlässigkeit und Latenz für diese Anwendungsklasse.

Umgekehrt werden eingehende Steuerinformationen, die z. B. neue Zielwerte für Bodenfeuchte, Temperatur oder Beleuchtung enthalten, vom Event-Handler verarbeitet, sobald eine MQTT-Nachricht auf dem Subscription-Topic eintrifft. In diesem Fall wird der Nachrichtenspeicher zunächst allokiert und die empfangenen Daten sicher aus dem Event-Buffer kopiert, wie in folgendem Listing 5.3 zu sehen ist:

```

1  char *msg = malloc(event->data_len + 1);
2  if (msg == NULL) {
3      break;
4  }
5  memcpy(msg, event->data, event->data_len);
6  msg[event->data_len] = '\0';
7
8  process_received_json(msg);
9
10 free(msg);

```

Listing 5.3: Empfangen von Solace-Daten

Die manuelle Speicherverwaltung (Allokation und Freigabe) stellt sicher, dass keine Datenverluste auftreten, wenn mehrere MQTT-Ereignisse in schneller Folge eintreffen. Der Zwischenschritt über `memcpy()` ist notwendig, da der Zeiger `event->data` nur innerhalb des Event-Kontexts gültig ist. Nach dem Kopieren wird der Inhalt an die Funktion `process_received_json()` übergeben, welche das JSON-Objekt parst und die enthaltenen Zielwerte in den persistenten Flash-Speicher schreibt. Dadurch werden Sollwerte zur Laufzeit aktualisiert und unmittelbar im weiteren Sensorzyklus wirksam.

Insgesamt übernimmt die MQTT-Client-Initialisierung eine zentrale Rolle im System, da sie die bidirektionale Kommunikation zwischen Hardware-Controller und Backend sicherstellt. Sie erlaubt nicht nur die kontinuierliche Datenerfassung und -übermittlung, sondern auch die Fernparametrierung des Systems durch das Backend. Das asynchrone Design mit Event-Handlers, QoS-gestütztem Datentransport und stabiler Speicherverwaltung entspricht den Anforderungen moderner IoT-Anwendungen hinsichtlich Robustheit, Modularität und Echtzeitfähigkeit.

Des Weiteren wird der verwendete MQTT-Client mit einem automatischen Reconnect-Mechanismus betrieben, um eine hohe Verfügbarkeit der Kommunikation sicherzustellen. Tritt ein Verbindungsabbruch auf, erkennt der zugrunde liegende MQTT-Client der ESP-

IDF diesen Fall und übernimmt selbstständig Wiederverbindungsversuche. Dieses Verhalten ist in Echtzeitsystemen mit instabiler oder intermittierender Netzwerkverbindung besonders vorteilhaft, da es die Notwendigkeit eines externen Fehlerbehandlungsmechanismus minimiert und die Robustheit des Systems erhöht. Ein manueller Eingriff durch das Backend ist in solchen Fällen nicht erforderlich.

Initialisierung der Aktorik (Pumpe)

Die Initialisierung der Pumpensteuerung erfolgt über die Funktion `pump_init()`, welche zwei grundlegende Aufgaben erfüllt: Einerseits wird der für die Relaisansteuerung vorgesehene GPIO-Pin hardwareseitig konfiguriert und initial in einen definierten Zustand versetzt (High = Relais aus), andererseits wird eine dedizierte FreeRTOS-Task mit dem Namen `pump_task` gestartet, die in der Abbildung 5.10 auf Seite 180 unter „setup pump task“ zu erkennen ist. Diese Task ist dauerhaft aktiv und wartet blockierend auf neue Steuerbefehle, welche durch die Sensorik generiert werden.

Die Pumpe selbst wird über ein SRD-05VDC-SL-C Relaismodul gesteuert, das bei Aktivierung den Stromkreis zur Wasserpumpe schließt. Die Softwarelogik verwendet dazu die `gpio_set_level()`-Funktion, um den Relaiskontakt aktiv (Low) oder inaktiv (High) zu schalten. Eine Initialisierung mit eindeutigem Anfangszustand ist dabei essenziell, um ungewolltes Aktivieren der Pumpe beim Systemstart zu verhindern.

Die Task `pump_task` wird über die FreeRTOS-Funktion `xTaskCreatePinnedToCore()` auf Core 1 des ESP32 platziert. Diese Kernzuweisung erfolgt bewusst, um eine Lastverteilung zwischen Kommunikations- und Sensor-/Aktorlogik zu ermöglichen. Da der WLAN-Stack sowie der MQTT-Client auf Core 0 des ESP32 betrieben werden, ergibt sich durch diese Trennung eine höhere Echtzeitfähigkeit der Pumpenlogik und eine geringere Interferenz zwischen den Tasks. Diese Architekturentscheidung trägt zur Stabilität und deterministischen Ausführung der Aktorik bei, insbesondere unter hoher Netzwerklast.

Die in Listing 5.4 auf der nächsten Seite gezeigte Erstellung der `pump_task` wurde mit einer Priorität von 5 gewählt, da sie auf eingehende Steuerbefehle schnell und zuverlässig reagieren muss. Die höhere Priorität stellt sicher, dass Pumpvorgänge bevorzugt behandelt werden. Die Stackgröße von 2048 Bytes ist ausreichend, da die Task nur einfache Befehle verarbeitet und keine speicherintensiven Operationen durchführt.

```

1 BaseType_t result = xTaskCreatePinnedToCore(
2     pump_task,           // Task-Funktion
3     "pump_task",         // Name
4     2048,                // Stackgroesse
5     NULL,                // Parameter
6     5,                   // Prioritaet
7     NULL,                // Handle (nicht verwendet)
8     1                    // Core 1
9 );

```

Listing 5.4: Start der Pump-Task

Neben der Task-Erzeugung wird eine Queue `pumpQueue` erstellt, welche als asynchrone Kommunikationsschnittstelle zwischen Sensorik und Aktorik dient. Über diese Queue werden Befehle der Struktur `pump_params_t`, welche unter anderem die gewünschte Pumpdauer enthalten, an die Aktorik weitergereicht. Dieses Kommunikationsmuster entspricht dem klassischen *Producer-Consumer-Modell*, bei dem die Sensorik als Ereignisquelle (Producer) fungiert und die Aktorik aufgabenbasiert reagiert (Consumer).

Die Queue dient dabei nicht nur der Synchronisation, sondern ermöglicht auch eine temporäre Entkopplung der Aufgaben. Selbst bei kurzen Verzögerungen in der Ausführung der Pumpen-Task kann ein eingehender Befehl zwischengespeichert werden, was zur Entkopplung der Sensordatenverarbeitung von der physikalischen Ausführung beiträgt. Diese Architekturentscheidung verbessert sowohl die Reaktionsfähigkeit als auch die Systemrobustheit.

Die beiden im Ablaufdiagramm (vgl. Abbildung 5.10 auf Seite 180) aufgeführten Schritte `pump_init()` sowie „setup pump task“ stellen also die Grundlage für die Aktorik im Regelbetrieb dar und ermöglichen die automatische Umsetzung von Bewässerungentscheidungen in hardwareseitige Steuerimpulse.

Start der Sensorik-Task

Die kontinuierliche Erfassung und Verarbeitung der Umgebungsdaten bildet das funktionale Zentrum des Regelbetriebs. Dieser Prozess wird durch die FreeRTOS-Task `read_sensordata()` realisiert, welche zyklisch die Werte aller verbauten Sensoren erfasst, bewertet, aggregiert und im Anschluss sowohl lokal verarbeitet als auch über MQTT an das Backend übermittelt. Der Start dieser Task erfolgt in der Funktion `sensor_init()` und wird explizit dem Core 1 des ESP32 zugewiesen.

Wie auch bei der parallel laufenden Pump-Task, dient die gezielte Platzierung auf Core 1 dazu, Konflikte mit systemnahen Funktionen wie WLAN- und MQTT-Kommunikation zu vermeiden, welche bevorzugt auf Core 0 abgewickelt werden. Durch diese Kerntrennung wird eine deterministische und unterbrechungsarme Ausführung der zyklischen Messpro-

zesse gewährleistet.

Die Erstellung der Task erfolgt mit einer Priorität von 2 und einer Stackgröße von **8192 Bytes**. Die moderate Priorität wurde gewählt, da die Sensorik-Task zwar regelmäßig und zuverlässig ausgeführt werden muss, jedoch nicht auf unmittelbare Reaktion angewiesen ist. Eine Priorität von 2 stellt sicher, dass die Task bei ausreichenden Ressourcen regelmäßig eingeplant wird, jedoch gegenüber zeitkritischeren Prozessen – wie etwa der Pumpensteuerung mit Priorität 5 – zurücktritt. Diese Priorisierung entspricht den Anforderungen eines Echtzeitssystems, in dem Datenaufnahme zwar zeitnah, jedoch nicht unbedingt unterbrechungsfrei ablaufen muss.

Die vergleichsweise große Stackgröße von **8192 Bytes** wurde gewählt, um der erhöhten Speicheranforderung durch folgende Faktoren Rechnung zu tragen:

- Speicherung mehrerer Sensorwerte und Mittelwerte in Arrays,
- Verarbeitung mehrerer cJSON-Objekte bei der MQTT-Nachrichtenerstellung,
- Nutzung zusätzlicher Hilfsfunktionen (z. B. Zeitstempelerzeugung, Statusprüfung),
- Absicherung gegen potenzielle Stacküberläufe bei komplexeren JSON-Strukturen.

```

1   xTaskCreatePinnedToCore(
2     &read_sensordata,      // Task-Funktion
3     "read_sensordata",    // Name
4     8192,                // Stackgroesse in Bytes
5     NULL,                // Parameter
6     2,                   // Prioritaet
7     NULL,                // Handle (nicht benoetigt)
8     1                    // Core-ID
9   );

```

Listing 5.5: Start der Sensorik-Task

Im Ablaufdiagramm (vgl. Abbildung 5.10 auf Seite 180) ist der gesamte Funktionsblock der Sensorik-Task im unteren Bereich dargestellt, beginnend mit dem Schritt `read_sensordata()`. Innerhalb der Task werden in einem fortlaufenden Loop nacheinander die Messwerte aller Sensoren erfasst. Dazu zählen:

- die Bodenfeuchte über den kapazitiven Sensor (ADC),
- Temperatur und Luftfeuchte über den DHT11 (digital, GPIO),
- sowie die Beleuchtungsstärke über den BH1750 (I²C).

Die dabei gewonnenen Messwerte werden zunächst in Arrays zwischengespeichert. Sobald pro Sensor drei Einzelmessungen (`SAMPLES = 3`) vorliegen, wird ein arithmetischer Mittelwert gebildet. Dies erfolgt zur Rauschunterdrückung und Glättung von Ausreißern, welche etwa durch Störimpulse oder instabile Umgebungsbedingungen entstehen können. Die entsprechende Funktion `compute_mean()`, zu sehen unter Listing 5.6, summiert dazu alle Einzelwerte und berechnet daraus den Durchschnitt. Der Algorithmus ist bewusst einfach gehalten, um Rechenzeit zu sparen und eine konstante Ausführungszeit zu garantieren:

```

1 int compute_mean(const int values[]) {
2     int sum = 0;
3     for (int i = 0; i < SAMPLES; i++) {
4         sum += values[i];
5     }
6     return sum / SAMPLES;
7 }
```

Listing 5.6: Mittelwertbildung zur Glättung von Sensorwerten

Vor der Weiterverarbeitung wird zusätzlich eine Validierung durchgeführt, ob der jeweilige Sensor als aktiv und funktionsfähig eingestuft werden kann. Dabei wird überprüft, ob alle Messwerte innerhalb eines gültigen Wertebereichs liegen. Dies verhindert, dass fehlerhafte Sensoren (z. B. durch Kabelbruch oder fehlerhafte Kalibrierung) zu ungültigen Steuerentscheidungen führen. Die Funktion `is_sensor_active()` übernimmt diese Plausibilitätsprüfung:

```

1 bool is_sensor_active(const int values[], int sampleCount) {
2     for (int i = 0; i < sampleCount; i++) {
3         if (values[i] <= 0 || values[i] >= 4095) {
4             return false; // Sensorfehler oder Disconnect
5         }
6     }
7     return true;
8 }
```

Listing 5.7: Überprüfung der Sensoraktivität

Sobald vier Mittelwerte (`MEAN_COUNT = 4`) pro Sensor vorliegen, erfolgt die Erzeugung einer strukturierten JSON-Nachricht. Diese umfasst Messwerte, Zeitstempel, Sensorstatusinformationen sowie Controller-Metadaten und wird über die zuvor erläuterte Schnittstelle `send_message()` via MQTT an das Backend gesendet.

Parallel zur Messwerterfassung wird bei jeder neuen Bodenfeuchte-Messung die Bewässerungslogik aktiviert, wie im Ablaufdiagramm unter „call watering logic“ dargestellt. Die

Entscheidungslogik erfolgt durch Aufruf der Funktion `calculate_watering_duration()`, welche im nächsten Abschnitt detailliert beschrieben wird. Führt diese Analyse zu einer positiven Bewässerungsentscheidung, wird ein entsprechender Pump-Befehl über die Funktion `pump_send_command()` an die globale Queue `pumpQueue` übermittelt, welche als Kommunikationsschnittstelle zur `pump_task()` dient und die sichere, asynchrone Übergabe von Steuerbefehlen ermöglicht.

Insgesamt vereint die Sensorik-Task zentrale Aufgaben der Datenerfassung, Signalverarbeitung, Qualitätssicherung, Entscheidungslogik und Backend-Kommunikation. Ihre saubere architektonische Einbindung sowie die klar definierte Schnittstelle zur Aktorik machen sie zu einem robusten und flexiblen Kernbestandteil des Gesamtsystems.

Bewässerungslogik

Die Funktion `calculate_watering_duration()` ist für die Ermittlung der notwendigen Bewässerungsdauer auf Basis der Differenz zwischen gemessener und Ziel-Bodenfeuchte zuständig. Sie lädt die Systemkonfiguration, vergleicht Soll- und Ist-Werte und berechnet daraus die Pumpdauer. Um Über- und Unterbewässerung zu vermeiden, wird die Dauer auf definierte Grenzwerte begrenzt.

Die Entscheidung darüber, ob eine Bewässerung erforderlich ist und in welchem Umfang diese erfolgen soll, wird durch die Funktion `calculate_watering_duration()` getroffen. Dieser Funktionsaufruf ist im Ablaufdiagramm unter dem Knoten „call watering logic“ eingezeichnet und bildet das zentrale Bindeglied zwischen der gemessenen Bodenfeuchte und der Ansteuerung der Pumpenlogik.

Die Funktion wird bei jeder neuen Messung der Bodenfeuchte innerhalb der Sensorik-Task aufgerufen. Sie lädt zunächst die gespeicherten Systemdaten, insbesondere den konfigurierten Zielwert für die Bodenfeuchte (`target_moisture`) aus dem persistenten Speicher. Anschließend wird geprüft, ob der aktuelle Feuchtwert unterhalb des Sollwertes liegt. Nur wenn ein solches Defizit besteht, wird eine Pumpdauer berechnet.

Wie in Listing 5.8 auf der nächsten Seite zu erkennen, basiert die Berechnung der Dauer auf einem linearen Modell: Die Differenz zwischen Ziel- und Istwert wird mit einem konstanten Faktor (`WATERING_K_FACTOR`) multipliziert, der in diesem Fall auf 100 gesetzt wird. Dieses einfache Modell erlaubt eine intuitive Parametrierung und gewährleistet eine proportionale Anpassung der Bewässerungsdauer an das Ausmaß der Trockenheit.

```

1  int calculate_watering_duration(int current_moisture) {
2      load_system_data(&info);
3
4      if (!info.target_moisture) {
5          info.target_moisture = 0;
6      }
7
8      if (current_moisture >= info.target_moisture) {
9          return 0;
10     }
11
12     int deficit = info.target_moisture - current_moisture;
13     int duration = deficit * WATERING_K_FACTOR;
14
15     if (duration < MIN_WATERING_TIME) return 0;
16     if (duration > MAX_WATERING_TIME) duration = MAX_WATERING_TIME;
17
18     return duration;
19 }
```

Listing 5.8: Berechnung der Bewässerungsdauer

Führt die Funktion zu einer Nicht-Null-Dauer, wird das Ergebnis in Form eines `pump_params_t`-Strukturobjekts an die zuvor initialisierte Queue `pumpQueue` übergeben. Die Übergabe erfolgt über die Funktion `pump_send_command()`, wodurch die asynchron laufende Pumpen-Task (vgl. Listing 5.4 auf Seite 185) aktiviert wird. Dieses Systemdesign gewährleistet eine klare Trennung zwischen Entscheidung und Ausführung, was die Modularität, Wartbarkeit und Reaktionssicherheit des Systems erhöht.

Zusammenfassend bildet die Funktion `calculate_watering_duration()` die Kernkomponente der Regelstrategie. Durch die Kombination aus Zielwertvergleich, linearer Be- rechnung und Schutzwertgrenzen wird ein praxisnahes Steuerverhalten realisiert, das zugleich einfach und robust in der Anwendung ist. Ihre Position innerhalb der Sensorik-Task er- laubt eine kontinuierliche Reaktion auf veränderte Umweltbedingungen und macht die Bewässerungsentscheidung zu einem integralen Bestandteil des zyklischen Regelbetriebs.

Testen der Funktionalitäten

Die Validierung der im Rahmen dieser Arbeit implementierten Systemfunktionen erfolgte durch eine Kombination aus lokalen Integrationstests und laufzeitbasiertem Debugging über den seriellen Monitor. Ziel war es, sowohl die Korrektheit der Sensorerfas- sung und Datenverarbeitung als auch die Robustheit der Kommunikations- und Aktorik- Komponenten unter realitätsnahen Bedingungen zu überprüfen.

Ein zentrales Element der Testinfrastruktur war die Verwendung einer containerisierten MQTT-Broker-Umgebung auf Basis von Solace. Der Broker wurde über eine

`docker-compose.yml`-Datei lokal bereitgestellt, welche alle relevanten Ports (u. a. 1883 für MQTT sowie 8080 für das Web-UI) freigibt und essenzielle Umgebungsparameter für die Broker-Verwaltung definiert. Durch diese Virtualisierung konnte eine vollständig kontrollierte und jederzeit rekonstruierbare Testumgebung geschaffen werden, ohne auf externe Dienste angewiesen zu sein.

Die `docker-compose.yml` ermöglichte somit die schnelle Inbetriebnahme und erlaubte über das Web-Interface auch die manuelle Kontrolle von Topic-Inhalten, Subskriptionen sowie eingehenden Sensorwerten. Darüber hinaus konnte die Reaktion des Systems auf eingehende Steuerbefehle (z. B. Zielwertänderungen) in Echtzeit nachverfolgt und ausgewertet werden. Dieses Setup bildete die Grundlage für systematische Integrationstests der gesamten MQTT-Kommunikationskette.

Zur Laufzeitanalyse des embedded Systems wurde intensiv der serielle Monitor (UART-Konsole) genutzt. Über die im Code implementierten Logging-Ausgaben (mittels `ESP_LOGI`, `ESP_LOGE`, `ESP_LOGW`) konnten interne Zustände und Entscheidungen transparent nachvollzogen werden. Repräsentative Lognachrichten betrafen u. a. erfolgreiche Verbindungen zu MQTT-Broker oder Messwerte einzelner Sensoren und deren Mittelwerte.

Insgesamt stellte das kombinierte Vorgehen aus containerisierter Backend-Umgebung und systematischer UART-Analyse eine effektive Methode zur Verifikation aller Kernfunktionalitäten dar. Die Modularität der Softwarearchitektur erleichterte dabei das gezielte Testen einzelner Komponenten, während die Loggingstruktur eine transparente und nachvollziehbare Fehlerdiagnose ermöglichte.

6 Kritische Reflexion

Dieses Kapitel widmet sich der retrospektiven Betrachtung des gesamten Projekts. Dabei werden die Zielerreichung, die Qualität der Teamarbeit, die getroffenen technologischen Entscheidungen sowie der Projektverlauf hinsichtlich Zeitmanagement und Risikoplanung kritisch analysiert.

Zielerreichung und Projekterfolg

Das übergeordnete Ziel, ein automatisiertes, sensorgesteuertes Bewässerungssystem auf Basis eines ESP32-Microcontrollers zu entwickeln, konnte vollständig erreicht werden. Der entwickelte Prototyp ist funktionstüchtig, sammelt zuverlässig Umweltdaten, wertet diese aus und steuert die Bewässerung entsprechend. Die Systemarchitektur ist modular aufgebaut und prinzipiell erweiterbar. Auch die Anbindung an das externe Backend (MQTT-Broker) sowie die Benutzerinteraktion über eine WLAN-Einrichtungsseite funktionieren wie vorgesehen. Insgesamt kann das Projekt daher als technisch erfolgreich bewertet werden.

Zusammenarbeit und Schnittstellen

Im Verlauf des Projekts traten insbesondere in der Anfangsphase Schwierigkeiten in der teaminternen Kommunikation auf. Diese betrafen vor allem die Abstimmung der Schnittstellen zwischen den unterschiedlichen Systemkomponenten – also insbesondere zwischen dem Microcontroller, dem Backend (inkl. Authentifizierung) und der mobilen App. Unklare Verantwortlichkeiten und uneinheitlich dokumentierte Anforderungen führten wiederholt zu Redundanzen im Code und notwendigem Refactoring. Erst durch regelmäßige Absprachen und strukturiertere Koordination konnten diese Probleme im späteren Projektverlauf reduziert werden. Für zukünftige Projekte erscheint eine frühzeitige Definition der Datenschnittstellen sowie eine durchgängige Versionskontrolle unerlässlich.

Projektmanagement und Zeitplanung

Die zeitliche Planung des Projekts erwies sich rückblickend als zu knapp bemessen. Insbesondere unvorhergesehene technische Schwierigkeiten – etwa bei der Integration der Sensordatenverarbeitung oder der Authentifizierungslogik – führten zu Verzögerungen. Ein strukturiertes Risikomanagement wurde nicht im ausreichenden Maße berücksichtigt, was sich negativ auf die Reaktionsfähigkeit bei auftretenden Problemen auswirkte. Auch die parallele Abhängigkeit von verschiedenen Software- und Hardwarekomponenten führte zu Engpässen. Die Etablierung von Meilensteinen und flexiblen Pufferzeiten hätte hier zu einem reibungsloseren Ablauf beitragen können.

Technologische Entscheidungen

Die grundlegenden technologischen Entscheidungen wurden größtenteils fundiert und erfolgreich getroffen. Die Wahl des ESP32-Moduls als zentrales Steuerungselement erwies sich trotz seiner Komplexität als leistungsfähig und zukunftssicher. Die Verwendung des esp-idf-Frameworks ermöglichte zwar eine tiefergehende Kontrolle über das System, brachte jedoch auch einen deutlich höheren Implementierungs- und Debuggingaufwand mit sich als zunächst vermutet. Der Verzicht auf kommerzielle High-End-Sensorik wurde im Sinne des Projektbudgets bewusst in Kauf genommen. Trotz der eingeschränkten Präzision der eingesetzten Sensoren konnten stabile Messergebnisse erzielt werden, die für einen funktionalen Prototyp ausreichend sind.

6.1 Reflexion zur Frontend-Umsetzung

Die Umsetzung des Frontends im Rahmen dieses Projekts kann insgesamt als gelungen und stabil bewertet werden. Die Anwendung ist vollständig funktionsfähig und unterstützt sowohl die deutsche als auch die englische Sprache durch ein konsistentes Internationalisierungskonzept. Zusätzlich bietet das Interface die Auswahl zwischen einem Dark Mode und einem Light Mode, was zur Barrierefreiheit und zum Nutzungskomfort beiträgt.

Besonders hervorzuheben ist das moderne, einheitliche und visuell ansprechende Design, das konsequent auf aktuellen UI/UX-Prinzipien basiert. Durch die Integration von Gamification-Elementen wie individuellen Pflanzen-Avataren wurde die Nutzerbindung zusätzlich gestärkt. Die Verwendung bewährter Best Practices in der Frontend-Architektur sowie die Orientierung am Flux-Prinzip sorgen für einen klar strukturierten Datenfluss und eine effiziente Benutzerinteraktion.

Ein wesentlicher Aspekt der Frontend-Gestaltung war die Gewährleistung eines flüssigen Nutzererlebnisses durch intuitive Navigation und konsistente Layouts. Die modular

aufgebaute Komponentenstruktur ermöglicht eine gute Wartbarkeit und einfache Erweiterbarkeit der Anwendung.

6.1.1 Verbesserungspotential

Trotz der grundsätzlich hohen Qualität bestehen einige Optimierungsmöglichkeiten. Zum einen könnten Performance-Verbesserungen vorgenommen werden, um die Ladezeiten insbesondere bei datenintensiven Ansichten zu verringern. Zum anderen wurden einige Zu-satzfunktionen aus zeitlichen Gründen nicht realisiert, die in einer späteren Entwicklungsphase ergänzt werden können.

Darüber hinaus sind drei kleinere Bugs bekannt, die zum aktuellen Stand noch nicht behoben wurden:

- Auf Geräten ab der Android API Version 35 kann es bei aktiverter Drei-Punkte-Navigationsleiste zu einer Überlappung mit der App-eigenen Navigationsleiste kommen.
- Nach der Erstellung einer neuen Gruppe werden die darin enthaltenen Räume in der Bearbeitungsansicht nicht sofort angezeigt, sofern kein Seitenwechsel oder manueller Refresh erfolgt.
- Man bekommt einen Fehler, wenn man eine Pflanze bearbeitet, die einen fremden Controller hat, weil im Backend diese Änderungen nicht angenommen werden.

Diese Einschränkungen haben jedoch keinen kritischen Einfluss auf die Hauptfunktionen und Nutzbarkeit der Anwendung.

Bei einer Weiter Entwicklung des Frontends und damit auch bei einer realen Nutzung sollten Frontend-Test in Verbindung mit E2E-Tests durchgeführt werden. Dadurch können Funktionen überprüft und Bugs vermieden werden.

6.1.2 Fazit

Die in der Konzeption formulierten Anforderungen an das Frontend wurden weitestgehend erfolgreich umgesetzt. Die Anwendung bietet ein modernes, benutzerfreundliches Interface mit internationaler Ausrichtung und ansprechendem Design. Funktionalität, Nutzerfluss und Wartbarkeit konnten auf hohem Niveau realisiert werden. Die identifizierten Verbesse rungspunkte bieten darüber hinaus eine wertvolle Grundlage für zukünftige Weiterentwicklungen.

6.2 Reflexion der Microcontroller-Programmierung

Die Entwicklung der Steuerungssoftware für den **ESP32-WROOM-32D** stellte einen zentralen technischen Teilapekt dieses Projektes dar. Ziel war es, ein modulares, robustes und erweiterbares Embedded-System zu entwerfen, das Sensordaten erfassen, verarbeiten und zur Steuerung eines Bewässerungsvorgangs nutzen kann. Grundsätzlich konnten alle im Vorfeld definierten funktionalen Anforderungen erfolgreich umgesetzt werden. Dennoch zeigten sich im Verlauf der Entwicklung typische Herausforderungen, begründete Einschränkungen sowie Verbesserungspotentiale, die im Folgenden kritisch reflektiert werden.

6.2.1 Komplexität der Hardwareanbindung

Die Anbindung der verwendeten Sensorik – bestehend aus dem **GY-302 BH1750** Lichtsensor (I^2C), dem **DHT11** Temperatur- und Luftfeuchtigkeitssensor (digitaler GPIO) sowie dem **HiLetgo LM393** Bodenfeuchtesensor (analog, ADC) – erwies sich erwartungsgemäß als verhältnismäßig unkritisch. Alle Sensoren konnten erfolgreich über entsprechende Treiberbibliotheken initialisiert und zyklisch ausgelesen werden. Bei der I^2C -Kommunikation wurden Standardports verwendet, die durch einen `i2c_scanner()` überprüft werden, was die Fehlersuche erleichterte.

Aufgrund des Proof-of-Concept-Charakters des Projektes wurde bewusst auf qualitativ hochwertige Sensoren verzichtet. Stattdessen kamen besonders kostengünstige Module zum Einsatz, was unter anderem in Bezug auf Genauigkeit, Kalibrierbarkeit und Langzeitstabilität gewisse Einschränkungen mit sich bringt. Die Kalibrierung der Bodenfeuchtesensorik erfolgte lediglich grob anhand definierter ADC-Schwellen (DRY/WET), ohne systematische Eichung mit standardisierten Feuchtwerten.

6.2.2 Entwicklung der Steuerungslogik

Die Steuerungslogik wurde als zyklischer Prozess in einer eigenen FreeRTOS-Task implementiert. Sensordaten werden regelmäßig gemessen, gemittelt und anschließend analysiert. Die daraus abgeleiteten Aktionsentscheidungen – konkret die Steuerung der Wasserpumpe – erfolgen auf Basis einer einfachen Regel: Nur wenn der gemessene Mittelwert der Bodenfeuchte unterhalb des Sollwertes liegt, wird ein Bewässerungsimpuls ausgelöst. Diese monokausale Logik berücksichtigt aktuell keine weiteren Umweltfaktoren wie Temperatur oder Lichtintensität, obwohl entsprechende Sensoren vorhanden sind.

Die Dauer des Pumpvorgangs wird anhand eines linearen Defizitmodells berechnet. Diese einfache, nachvollziehbare Methode hat sich in der Umsetzung als stabil erwiesen, könnte aber perspektivisch durch multivariate oder lernfähige Entscheidungsmodelle ersetzt

werden.

6.2.3 Fehleranalyse und Debugging

Während der Entwicklung traten vor allem Probleme im Bereich der Synchronisierung, des Taskmanagements und der Speicherverwaltung auf. Besonders beim Einsatz mehrerer FreeRTOS-Tasks (Sensor-Task, Pumpen-Task, MQTT-Kommunikation) war auf die korrekte Priorisierung und den Stackverbrauch zu achten.

Tools wie `uxTaskGetStackHighWaterMark()` wurden gezielt eingesetzt, um Stackoverflows frühzeitig zu identifizieren.

Ein weiteres Fehlerpotenzial zeigte sich im Zusammenhang mit der WiFi-Verbindung: Wird die Verbindung unterbrochen, so können aktuell keine Sensordaten zwischengespeichert werden. Es fehlt eine clientseitige Pufferlogik, die bei Wiederherstellung der Verbindung eine Übertragung nachholt. Auch eine manuelle Auslösung einer Messung durch Nutzerinteraktion ist derzeit nicht implementiert.

Zudem wurde bewusst auf die Aktivierung von Flash Encryption verzichtet, um die Nachvollziehbarkeit und Nachprüfbarkeit des Programmcodes zu gewährleisten. Flash Encryption ist eine Sicherheitsfunktion, die den im Flash-Speicher abgelegten Programmspeicherinhalt verschlüsselt und somit vor unbefugtem Auslesen und Reverse Engineering schützt. Dies stellt im produktiven Einsatz ein relevantes Sicherheitsrisiko dar, jedoch bleibt damit die Möglichkeit des uneingeschränken Beschreibens für die Entwicklung erhalten.

6.2.4 Codequalität und Modularität

Der Code wurde in einzelne, klar strukturierte Module unterteilt (`sensor_manager`, `pump_manager`, `device_manager` etc.), wodurch eine gute Lesbarkeit und Erweiterbarkeit gewährleistet ist. Die Verwendung von Header-Dateien sowie des `esp_log`-Systems zur Laufzeitdiagnose trug wesentlich zur Wartbarkeit bei. Die Verwendung von Queues zur Steuerung asynchroner Ereignisse (z. B. Pumpenbefehl) entspricht bewährten Embedded-Designprinzipien.

Einige geplante Features wie die Anzeige aktueller Sensordaten auf einem lokalen Display oder ein OTA-Update-Mechanismus (Over-the-Air) wurden bislang nicht umgesetzt, sind jedoch in der Systemarchitektur antizipiert und könnten in zukünftigen Erweiterungen ergänzt werden.

6.2.5 Lernfortschritt und technologische Erkenntnisse

Die Arbeit am Microcontroller-Modul bot wertvolle praktische Erfahrungen in der Low-Level-Programmierung, insbesondere im Bereich:

- FreeRTOS Task-Management und Inter-Task-Kommunikation (Queues)
- GPIO-Steuerung und Sensorauswertung (ADC, I²C, Digital)
- Datenverarbeitung (z. B. Mittelwertbildung, Fehlererkennung)
- Kommunikationsprotokolle (MQTT, JSON)
- Persistente Datenspeicherung über NVS
- Wifi-Provisionierung
- Web-Server-Hosting

Zudem konnte ein vertieftes Verständnis für typische Probleme und Designentscheidungen in Embedded-Systemen entwickelt werden. Besonders hervorzuheben ist das Zusammenspiel aus hardwarenaher Programmierung, Netzwerkintegration und Systemverantwortung im Gesamtkontext eines realen Anwendungsszenarios.

6.3 Kritische Refelexion Auth-Service

Die oben geschilderte Implementierung des Auth-Service reflektiert mehrere sicherheitsrelevante und architekturelle Entscheidungen:

1. HMAC-basierte PSK-Verifikation: Die Wahl eines Challenge-Response-Verfahrens mit HMAC stellt sicher, dass ein Gerätegeheimnis nicht direkt übertragen wird, sondern nur indirekt bewiesen. Der Auth-Service speichert Tokens gesalzen (mittels TOKEN_SECRET) als Hash in seiner Config, was grundsätzlich ein Sicherheitsvorteil ist. Allerdings verbleibt das Token zusätzlich im Klartext in der Konfigurationsdatei, was im Widerspruch zu bewährten Sicherheitsprinzipien steht – es wäre ausreichend, nur den Hash zu persistieren und das Token nach Ausgabe zu verwerfen[160].
2. Kopplung mit Systemdatenbank: Die referenzielle Konsistenz durch Verknüpfung von Controller und User in der Datenbank unterstützt zentrale Anforderungspunkte. Jedoch erhöht sich dadurch auch die Kopplung einzelner Komponenten, was bei Änderungen im Datenbankschema zu potenziellen Wartungsaufwänden führen kann.

3. Automatisierte Broker-Provisionierung: Die dynamische Benutzer- und ACL-Erstellung über die Solace SEMPv2 API bietet hohe Flexibilität und Skalierbarkeit. Die Implementierung reagiert korrekt auf typische Statuscodes wie HTTP 409, bleibt jedoch ohne Transaktionsmechanismen anfällig für inkonsistente Zustände, falls einzelne Schritte des Provisionierungsprozesses scheitern. Eine explizite Fehlerbehandlung für Teilausfälle wäre hier sinnvoll.
4. Sichere Auslieferung von Secrets: Die Übertragung der Broker-Zugangsdaten erfolgt verschlüsselt und signiert, wodurch ein zusätzlicher Schutzlayer entsteht. Dennoch erfolgt die Übertragung des temporären Session Keys im Klartext (wenn auch signiert), was gegenüber klassischen Key-Exchange-Protokollen eine reduzierte Sicherheit darstellt. Angesichts des bestehenden PSK stellt dies jedoch eine pragmatische, wenn auch nicht optimale Lösung dar.
5. Fehlerbehandlung und Logging: Die granulare Fehlerdifferenzierung und das konstante Logging tragen zur Transparenz und Diagnosefähigkeit bei. In produktiven Szenarien könnte jedoch ein zentralisiertes Log-Management hilfreich sein, um sicherheitsrelevante Ereignisse auswertbar zu halten.

Zusammenfassend realisiert der Auth-Service die in den Anforderungen definierte Geräteauthentifizierung durch einen robusten, mehrschrittigen Prozess. Die Verwendung etablierter Kryptographie (HMAC-SHA256, AES-basierte Fernet-Verschlüsselung) in Kombination mit einer modularen Architektur erfüllt die Sicherheits- und Skalierbarkeitsanforderungen. Einzelne sicherheitsrelevante Details, wie die Speicherung von Tokens im Klartext und fehlende Transaktionslogik bei der Broker-Provisionierung, sollten jedoch überarbeitet werden.

6.4 Kritische Reflexion Mailservice

Einige besondere Aspekte und Entscheidungen des Mail-Service:

1. Einsatz von FastAPI und Async DB: Die Wahl von FastAPI und asyncpg ermöglicht performante, nicht-blockierende Datenbankabfragen. Die Verwendung einer lifespan-Funktion zur Initialisierung der Datenbankverbindung ist effizient. Dennoch fehlt eine Absicherung gegen potenzielle Verbindungsverluste zur Laufzeit – ein automatischer Reconnect-Mechanismus wäre für höhere Robustheit wünschenswert.
2. Preshared Service Key (PSK) für /verify: Der PSK-Mechanismus schützt den sensiblen Verifikationsendpunkt wirkungsvoll gegen unautorisierte Aufrufe. In der gege-

benen Architektur ist der statische Schlüssel praktikabel, für größere Systeme oder Produktionsumgebungen wäre jedoch ein zeitlich begrenzter, rotierender Schlüssel oder ein dedizierter Auth-Service vorzuziehen.

3. Token-Generierung und -Speicherung: Die Token-Erzeugung mit `secrets.token_urlsafe` erfüllt kryptografische Anforderungen. Die ausschließliche Speicherung im RAM (Python Dictionary) vermeidet Persistenzprobleme, führt jedoch zu Verlust bei Dienstneustart. Dies kann in Szenarien mit hoher Verifikationslatenz problematisch sein. Eine persistente Speicherung in der Nutzerdatenbank wäre hier langfristig robuster.
4. Aktivierung des Benutzerkontos: Die Kapselung der Account-Aktivierung im Service ist gelungen. Jedoch gibt es keine Validierung, ob das Token bereits abgelaufen oder mehrfach genutzt wurde. Eine Ergänzung um zeitbasierte Gültigkeit oder Einmaligkeit wäre zur Erhöhung der Sicherheit empfehlenswert.
5. Versand per SMTP: Die Entscheidung, den Mailversand über SMTP und smtplib selbst zu realisieren, ist nachvollziehbar, aber mit gewissen Risiken verbunden. Die Nutzung eines persönlichen Gmail-Kontos sowie die fehlende Wiederholungslogik bei Versandfehlern (z.B. temporärer SMTP-Ausfall) schränken die Zuverlässigkeit und Sicherheit ein. Für produktive Umgebungen wäre ein externer Maildienst mit stabiler API und Fehlerbehandlung vorzuziehen.
6. Feedback an den Nutzer: Die Rückmeldung nach erfolgreicher Verifikation erfolgt korrekt, ist jedoch funktional und gestalterisch minimalistisch gehalten. Eine klarere Benutzerführung – etwa durch Weiterleitung in die Hauptanwendung – könnte die Nutzererfahrung verbessern.

Durch die Entkopplung des E-Mail-Versands in einen eigenen Microservice bleibt die Verantwortung klar getrennt. Die Umsetzung ist funktional und erfüllt die Grundanforderungen. Verbesserungsbedarf besteht insbesondere bei der Tokenpersistenz und SMTP-Fehlertoleranz, um den Dienst in produktiven Szenarien zuverlässig betreiben zu können.

6.5 Kritische Reflexion Database Writer

Eine entscheidende Stärke dieser Implementierung ist die Nutzung der persistenten Solace-Queue. Dadurch wird das Prinzip der garantierten Zustellung (Guaranteed Delivery) zuverlässig umgesetzt. Dennoch gibt es einige Punkte, die kritisch hinterfragt werden können:

1. Die Inserts der Messwerte basieren auf einem at-least-once-Verarbeitungsmuster. Während dies grundsätzlich sinnvoll ist, fehlen dedizierte Mechanismen zur Dublettenvermeidung auf Werteebene. Das System verlässt sich auf die Einzigartigkeit der `vid`, wodurch bei Redelivery identische Messwerte mehrfach gespeichert werden können. In produktiven Szenarien könnte dies zu Datenverzerrungen führen. Eine explizite Deduplikationsstrategie – etwa über kombinierte Primärschlüssel oder Hashing der Payload – wäre wünschenswert.
2. Die Idempotenz bei der Sensoranlage wird nur durch die sequenzielle Abarbeitung im Single-Thread sichergestellt. Dies funktioniert im aktuellen Setup, skaliert jedoch nicht ohne weiteres auf parallele Consumer. Eine atomare Datenbankoperation (z.B. `INSERT ON CONFLICT DO NOTHING`) wäre hier robuster.
3. Die Statusverwaltung des Sensors (z.B. Wechsel zu 'error') ist funktional implementiert, jedoch in der Logik sehr spezifisch auf einen 5-Minuten-Zeitraum beschränkt. Eine flexiblere, konfigurierbare Lösung – z.B. ein Watchdog-Modul mit Schwellenwerten – könnte das System anpassungsfähiger machen.
4. Bei Ausfällen der Datenbank wird keine explizite Retry-Strategie verfolgt. Das aktuelle Verhalten – Rückgabe von `None` und Verzicht auf Ack – ist korrekt und verhindert Datenverlust. Jedoch erfolgt keine gezielte Behandlung der hängengebliebenen Nachrichten, etwa durch automatisiertes Requeueing oder Logging für spätere Analyse. Eine gezielte Fehlerbehandlung (z.B. persistente Dead-Letter-Queue) könnte hier mehr Transparenz und Kontrolle schaffen.
5. Die Logging-Strategie mit leicht menschenlesbarer Struktur erleichtert die Entwicklung und Analyse. Für eine produktive Umgebung fehlen jedoch strukturierte Logs (z.B. JSON), die von externen Tools (wie ELK oder Grafana Loki) verarbeitet werden könnten. Zudem fehlt eine Differenzierung nach Komponenten, um z.B. zwischen Netzwerkfehlern und Datenbankproblemen besser zu unterscheiden.

Im Rahmen dieser Arbeit zeigt sich, dass der Service zuverlässig und fehlertolerant arbeitet. Die grundlegenden Prinzipien – insbesondere die Trennung von Empfang und Persistenz durch Messaging – sind sinnvoll umgesetzt. Für den produktiven Betrieb wären allerdings Verbesserungen bei Idempotenz, Fehlerbehandlung und Logging erforderlich, um Skalierbarkeit und Betriebssicherheit langfristig zu gewährleisten.

6.6 Kritische Reflexion Setpoint-API

Der Setpoint-Service ist relativ einfach gestrickt, dennoch gibt es einige bemerkenswerte Punkte:

1. Flask mit Swagger: Ähnlich dem Auth-Service wurde hier Flask verwendet, ergänzt um Flasgger für die API-Dokumentation. Somit kann auch dieser Service seine Schnittstelle (Parameter, Responses) im Swagger-UI darstellen. Die Entscheidung für Flask (anstelle von FastAPI) erscheint aus Konsistenzgründen nachvollziehbar, birgt jedoch Einschränkungen hinsichtlich asynchroner Verarbeitung. Zwar spielt dies im aktuellen Anwendungsfall keine Rolle, langfristig wäre eine einheitliche, moderne Technologieplattform vorteilhaft.
2. Broker-Konnektivität: Der Service initialisiert beim Start eine Verbindung zum Solace-Broker und erzeugt einen persistenten Publisher. Sollte der Broker zu diesem Zeitpunkt nicht verfügbar sein, wird durch die Retry-Schleife ein erneuter Verbindungsversuch unternommen. Allerdings fehlt eine Statusüberwachung der Verbindung zur Laufzeit. Ein unerwarteter Verbindungsabbruch könnte dazu führen, dass der Service Anfragen annimmt, aber keine Nachrichten mehr versenden kann. Dies müsste entweder aktiv überwacht oder durch geeignete Fehlerbehandlung im Code abgesichert werden.
3. Payload-Struktur und Flexibilität: Die aktuelle Struktur erwartet, dass Controller- und Sensor-ID direkt übergeben werden. Die angedeutete Weiterentwicklung, diese IDs aus der Datenbank zu ermitteln, wurde bislang nicht umgesetzt. Dies führt zu einer potenziellen Fehlerquelle, wenn fehlerhafte Zuordnungen gemacht werden. Eine stärkere Entkopplung und Validierung anhand der DB wäre hier ein Qualitätsgewinn – insbesondere im Hinblick auf die korrekte Adressierung von Geräten.
4. Security und Authentifizierung: Der Service verfügt über keinerlei Authentifizierungsmechanismus. In einer offenen Netzwerkumgebung stellt dies ein erhebliches Risiko dar: Jeder, der über die nötigen IDs verfügt, könnte Steuerbefehle an beliebige Controller senden. Auch wenn davon ausgegangen wird, dass eine vorgelagerte Authentifizierung existiert, sollte der Service zumindest ein optionales Authentifizierungsverfahren unterstützen (z.B. API-Key, JWT). Das Fehlen einer solchen Sicherung steht im Widerspruch zur sonst hohen Sicherheitsorientierung des Gesamtsystems.
5. Themenpersistenz: Die vom Auth-Service gesetzten ACLs verhindern erfolgreich, dass Nachrichten an unautorisierte Controller übermittelt werden. Dies schützt

zwar den Empfangsweg, verhindert jedoch nicht, dass falsche Sollwerte vom falschen Benutzer initiiert werden. Zudem basiert das Zustellmodell ausschließlich auf MQTT-Mechanismen. Die Möglichkeit zur Nutzung von Queues zur Bestätigung oder Rückmeldung wurde bislang nicht implementiert. Das könnte bei sicherheitskritischen Steuerbefehlen ein relevantes Defizit sein.

6. Fehlerbehandlung und Benutzerfeedback: Die Rückmeldungen an den Aufrufer sind funktional, aber minimalistisch. Im Fehlerfall wird lediglich eine Exception geworfen, die zu einem HTTP 500 führt. Eine genauere Differenzierung von Fehlertypen (z.B. ungültige IDs, Broker nicht erreichbar) wäre für Client-Systeme hilfreich. Auch ein Retry-Konzept bei temporären Fehlern könnte die Zuverlässigkeit verbessern.
7. Relevanz zum Gesamtsystem: Die Setpoint API realisiert einen der in der Einleitung geforderten Interface-Services. Sie ist relativ unabhängig von den anderen – außer dass sie natürlich von der existierenden Broker-Infrastruktur und der Auth vorher eingerichteten ACL profitiert. Ohne Auth-Service hätte der Controller kein Broker-Login oder keine Subscription. Hier sieht man schön die Arbeitsteilung: Der Auth-Service bereitet das System so vor, dass Controller Messwerte liefern und Sollwerte empfangen dürfen. Die Setpoint API nutzt diese vorbereitete Bahn, um auf einfache Weise Nachrichten an Geräte zu schicken, ohne jeden Controller einzeln zu kennen (außer über die ID).

Das gewählte REST-zu-MQTT-Muster ist gängig in IoT-Architekturen: Es erlaubt z.B. einer Webanwendung per HTTP (wofür es viele Tools/SDKs gibt) mit einem IoT-Gerät zu kommunizieren, das nur MQTT spricht, über einen Vermittlungsbroker. Dennoch zeigen sich bei der aktuellen Implementierung mehrere Schwächen im Hinblick auf Authentifizierung, Validierung und Fehlertoleranz. Diese sollten im weiteren Verlauf adressiert werden, um die Sicherheit und Robustheit der Kommunikation zu gewährleisten.

6.7 Kritische Reflexion Solace-Init

Solace Init stellt sicher, dass die notwendige Broker-Umgebung programmgesteuert hergestellt wird, ohne manuelle Schritte in der Solace-Admin-Oberfläche. Dies ist aus mehreren Gründen vorteilhaft:

1. Automatisierung: In containerisierten Deployments können so neue Umgebungen hochgefahren werden und sind sofort lauffähig, weil die benötigten Infrastruktur-Komponenten (Warteschlangen) automatisch erzeugt werden. Der Infrastructure-as-Code-Gedanke wird hier konsequent umgesetzt – die JSON-Datei fungiert als

zentrale Konfigurationsquelle. Eine gewisse Einschränkung besteht jedoch darin, dass die Warteschlange und Subscriptions ausschließlich zum Initialzeitpunkt erzeugt werden. Eine dynamische Reaktion auf Laufzeitänderungen (z.B. neue Topics) ist im aktuellen Design nicht vorgesehen.

2. Konsistenz: Es wird sichergestellt, dass z.B. die `sensor_data` Queue wirklich existiert, bevor andere Services (Database Writer) damit arbeiten. Die Verwendung von `Compose depends_on` allein reicht nicht aus, um dies zu garantieren. Solace Init füllt diese Lücke, indem es nach einer festen Wartezeit die Einrichtung vornimmt. Diese Wartezeit ist jedoch statisch gewählt und könnte bei langsamen Systemstarts zu einem Rennen führen – hier wäre eine robuste Verfügbarkeitsprüfung des Brokers vorzuziehen.
3. Wiederholbarkeit: Durch die Abfrage der HTTP-Antwort ($409 \rightarrow$ „existiert bereits“) ist das Skript idempotent. Man kann es gefahrlos erneut laufen lassen, etwa nach einem Broker-Neustart, ohne Duplikate zu erzeugen. Dies erleichtert Updates – z.B. kann ein neues Topic zur JSON-Datei hinzugefügt und das Skript erneut ausgeführt werden. Einschränkend bleibt jedoch festzuhalten, dass die Lösung keine Validierung bestehender Konfigurationen vornimmt: Änderungen (z.B. an ACLs) werden nicht überprüft oder aktualisiert, sondern nur ergänzt. Eine differenziertere Änderungsstrategie wäre für langfristige Wartbarkeit hilfreich.

Der Code von Solace Init ist bewusst einfach gehalten. Er verzichtet z.B. auf einen ständigen Daemon – er braucht nur einmal zu Beginn laufen. Ein kleiner Nachteil dieser Einfachheit: Wenn nachträglich (zur Laufzeit) neue Controller angelegt werden, könnten diese evtl. zusätzliche Topics erfordern. Allerdings deckt unser Modell das dynamisch über ACLs ab, nicht über Queues – neue Controller publizieren auf vorhandenes Muster `sensora/v1/send/<id>`, was durch den Wildcard `>` der bestehenden Subscription bereits erfasst ist. Somit muss man Solace Init nur erneut laufen lassen, wenn man das generelle Muster ändern würde. Dennoch wäre eine fortlaufende Integration mit der Controller-Verwaltung denkbar, etwa durch ein Triggern von Solace Init bei Anlage neuer Controller, um zukünftige Erweiterungen vollständig abzubilden.

7 Ausblick

Mit dem erfolgreichen Abschluss des vorliegenden Projekts konnte ein funktionierender Prototyp für ein automatisiertes Bewässerungssystem entwickelt werden, das zentrale Anforderungen hinsichtlich Datenerfassung, Steuerung und Kommunikation erfüllt. Gleichzeitig bietet das System vielfältige Ansatzpunkte für eine Weiterentwicklung und Professionalisierung in technischer, organisatorischer und konzeptioneller Hinsicht.

Potenziale für Systemerweiterungen

Die modulare Architektur des Systems ermöglicht eine schrittweise Erweiterung und Anpassung. Denkbar sind beispielsweise die Einbindung zusätzlicher Sensorik (z. B. Boden-pH, CO₂, Regen), die Ansteuerung mehrerer Pumpen in unterschiedlichen Zonen oder die Einbindung von Aktoren wie Ventile oder Ventilatoren. Auch eine Integration in bestehende Gartenmanagement- oder Smart-Home-Systeme eröffnet neue Anwendungsszenarien.

Optimierung der Systemintegration

Eine zentrale Herausforderung in der aktuellen Projektphase bestand in der Koordination zwischen Microcontroller, Backend und mobiler App. Künftig sollte die Systemintegration stärker auf standardisierte Protokolle und Datenformate setzen, um die Wartbarkeit und Austauschbarkeit einzelner Komponenten zu erhöhen. Insbesondere die Etablierung klar definierter Schnittstellen sowie eine kontinuierliche Integration (*CI*) könnten die Zusammenarbeit und Weiterentwicklung erleichtern.

Produktisierung und Praxistauglichkeit

Die Überführung des entwickelten Prototyps in ein marktfähiges Produkt würde eine Reihe zusätzlicher Anforderungen mit sich bringen, etwa hinsichtlich Gehäusedesign, Stromversorgung (z. B. Akku, Solarpanel), Energieoptimierung und Benutzerfreundlichkeit. Auch die Robustheit gegenüber Umwelteinflüssen und die Einhaltung regulatorischer Anforderungen müssten bei einer Produktisierung geprüft und berücksichtigt werden.

Nutzerzentrierung und Usability

Während der Fokus des vorliegenden Projekts primär auf der technischen Realisierung lag, wäre eine stärkere Einbindung der Perspektive potenzieller Nutzer in weiteren Projektphasen wünschenswert. Dazu zählen beispielsweise eine intuitive Benutzeroberfläche in der App, eine transparente Darstellung der Messwerte sowie einfache Möglichkeiten zur Konfiguration und Fehlerdiagnose. Auch Aspekte wie Mehrsprachigkeit, Barrierefreiheit oder eine smarte Benachrichtigungslogik könnten den Anwendungskomfort erheblich steigern.

Langfristige Perspektiven und Forschungspotenzial

Das Projekt bietet über den unmittelbaren Anwendungsfall hinaus auch spannende Anknüpfungspunkte für weiterführende Forschung. So könnten beispielsweise adaptive Systeme zur Ressourcenschonung in der Landwirtschaft, autonome Sensornetzwerke oder KI-gestützte Umweltanalysen auf der vorliegenden Arbeit aufbauen. Auch die Verbindung mit externen Wetterdiensten, Satellitendaten oder cloudbasierten Agrarsystemen bietet interessante Perspektiven.

7.1 Ausblick Frontend

Im weiteren Verlauf der Entwicklung bestehen vielfältige Potenziale zur funktionalen und gestalterischen Erweiterung des Frontends. Ein naheliegender Ansatzpunkt ist die Vertiefung der Gamification-Strategien. Erste Ideen, etwa visuelle Wetter-Overlays wie eine Sonne bei Trockenheit oder eine Regenwolke bei Staunässe oberhalb des Pflanzen-Avatars, wurden bereits erprobt, konnten jedoch aus Zeitgründen nicht final implementiert werden. Diese Erweiterung würde die emotionale Bindung zur virtuellen Pflanze weiter steigern und eine intuitivere Wahrnehmung ihres Zustandes ermöglichen.

Auch im Bereich der Datenvisualisierung besteht weiteres Entwicklungspotenzial. Die aktuelle Version der Anwendung zeigt Messwerte innerhalb eines Zeitfensters von 24 Stunden an. Die zugrundeliegenden Methoden und Datenmodelle erlauben jedoch eine verlängerte Betrachtung, beispielsweise über Tage oder Wochen hinweg. Eine benutzerfreundliche Auswahlkomponente für den gewünschten Zeitraum könnte hier eine wertvolle Ergänzung darstellen, um Langzeitveränderungen besser nachvollziehen zu können.

Ein weiterer spannender Entwicklungsschritt betrifft die Integration aktiver Steuerungsmechanismen für die Pflanzenbewässerung. Mit entsprechender Konfiguration und der Anbindung wäre es möglich, die Applikation um eine Live-Bewässerungsfunktion zu erweitern. Auch automatisierte Abläufe, etwa zur Simulation von Tageszyklen oder zur

reaktiven Anpassung an Umweltdaten, könnten in einem erweiterten System realisiert werden.

Diese möglichen Erweiterungen unterstreichen das technische und konzeptionelle Potenzial der aktuellen Frontend-Architektur und liefern wertvolle Ansatzpunkte für eine fortlaufende Weiterentwicklung.

7.2 Ausblick Microcontroller

Die erfolgreiche Realisierung des funktionalen Prototyps bildet eine solide Grundlage für zukünftige Erweiterungen und Verbesserungen. Auf technischer wie funktionaler Ebene bestehen vielfältige Ansatzpunkte zur Weiterentwicklung des Systems, die im Folgenden skizziert werden.

7.2.1 Erweiterung der Steuerungslogik

Derzeit basiert die Bewässerungsentscheidung ausschließlich auf dem gemessenen Bodenfeuchtwert. Für eine genauere Bedarfsanalyse wäre die Einbeziehung zusätzlicher Umweltparameter wie Temperatur, Luftfeuchtigkeit oder Lichtintensität wünschenswert. Auch die Kombination mehrerer Sensorwerte durch gewichtete Entscheidungsmodelle oder regelbasierte Steuerungen bietet sich an. Mittel- bis langfristig wäre der Einsatz datengetriebener Modelle (z. B. über einfache Machine-Learning-Algorithmen) denkbar, um situationsabhängige Optimierungen der Bewässerungsstrategie zu ermöglichen.

7.2.2 Pufferung und Ausfallsicherheit

Ein zentrales Ziel für kommende Iterationen besteht in der Erhöhung der Robustheit bei Netzwerkausfällen. Aktuell werden Sensordaten nur live übertragen; bei Verbindungsverlust zur MQTT-Infrastruktur gehen diese Daten unwiederbringlich verloren. Eine lokale Zwischenspeicherung der Sensordaten (z. B. in einem Ringpuffer) und eine nachträgliche Übertragung nach Wiederherstellung der Verbindung wäre ein entscheidender Schritt zur Verbesserung der Datenintegrität und Systemverlässlichkeit.

7.2.3 Sicherheit und Datenschutz

Die derzeitige Entwicklungsumgebung verzichtet bewusst auf die Aktivierung von Flash-Verschlüsselung und weiteren Sicherheitsfeatures, um die Nachvollziehbarkeit des Codes für Test- und Bewertungszwecke zu ermöglichen. Im produktiven Einsatz stellt dies jedoch ein potenzielles Sicherheitsrisiko dar. Zukünftig sollte die Flash Encryption aktiviert

werden, um den Schutz sensibler Daten – insbesondere Netzwerk- und Zugangsdaten – zu gewährleisten. Auch Mechanismen zur sicheren Authentifizierung, etwa über Token-Validierung und TLS-Verbindungen, sollten weiter ausgebaut werden.

7.2.4 Erweiterung der Nutzerinteraktion

Einige nützliche Funktionen im Bereich der Nutzerinteraktion wurden im aktuellen Prototyp noch nicht realisiert, könnten jedoch mit überschaubarem Aufwand nachgerüstet werden:

- **Manuelle Auslösung von Messvorgängen:** Eine Möglichkeit zur gezielten Datenerhebung durch den Nutzer, etwa über die Web-App oder einen physischen Button, wäre für Test- und Diagnosezwecke sinnvoll.
- **Anzeige der aktuellen Sensordaten:** Die Ausgabe der Messwerte auf einem kleinen Display (z. B. OLED, I²C) direkt am Gerät würde die Transparenz und Nutzbarkeit im Alltag zusätzlich erhöhen.
- **Userdaten-Übergabe:** Der bereits implementierte Web-Server könnte hinsichtlich seiner Funktionalität sowie dem UI-Design verbessert werden, um die Benutzerfreundlichkeit zu erhöhen.

7.2.5 Updatefähigkeit und Wartung

Zukünftige Versionen sollten die Möglichkeit eines **Over-the-Air (OTA)**-Updates beinhalten. Dies erlaubt die Wartung und Weiterentwicklung des Systems ohne physischen Zugriff auf das Gerät. Auch ein OTA-Reset zur Wiederherstellung eines definierten Werkszustandes wäre denkbar und würde insbesondere in produktiven Szenarien die Wartbarkeit erheblich verbessern.

7.2.6 Sensorik und Hardwarequalität

Im Rahmen des vorliegenden Prototyps wurde auf kostengünstige Sensorik zurückgegriffen, was sich in einer begrenzten Präzision und Langzeitstabilität bemerkbar machen kann. Für eine Serien- oder Feldanwendung wäre die Evaluation alternativer höherwertiger Sensormodelle mit besserer Genauigkeit, Kalibrierfähigkeit und Schutzklasse ratsam. Zudem sollten Energieverbrauch, Witterungsbeständigkeit und elektromagnetische Verträglichkeit der eingesetzten Komponenten systematisch verbessert werden.

7.2.7 Systemintegration und Skalierbarkeit

Langfristig sollte die Architektur des Systems so erweitert werden, dass mehrere Bewässerungseinheiten parallel betrieben und zentral verwaltet werden können. Auch die Integration in Smart-Home-Ökosysteme (z. B. über Home Assistant oder Matter-Protokolle) bietet ein großes Innovationspotenzial. Zur Unterstützung einer solchen Skalierung ist eine weitere Modularisierung des Codes sowie die Nutzung standardisierter Kommunikationsschnittstellen erforderlich.

Literatur

- [1] Mieterzeitung, „Grünes Zuhause,“ 2022, Abgerufen am 14. April 2025. Adresse: <https://www.mieterzeitung.de/ausgabe-1-2022/statistiken.html> (besucht am 14.04.2025).
- [2] A. Löbke. „Marktbericht: Pandemie treibt Nachfrage nach Grün.“ Abgerufen am 14. April 2025. (2022), Adresse: <https://www.gabot.de/ansicht/marketbericht-pandemie-treibt-nachfrage-nach-gruen-414635> (besucht am 14.04.2025).
- [3] feey AG. „Pflanzenblog - Juni 2021.“ Abgerufen am 14. April 2025. (2023), Adresse: <https://www.statista.com/statistics/1300299/how-many-indoor-plants-us-gardeners-kill-annually/> (besucht am 14.04.2025).
- [4] H. Advisor. „Gardening Expectations vs. Reality Survey.“ Abgerufen am 14. April 2025. (2021), Adresse: <https://www.homeadvisor.com/r/gardening-expectations-vs-reality-survey/> (besucht am 14.04.2025).
- [5] Statista. „How many indoor plants non-expert gardeners in the United States kill in a year.“ Abgerufen am 14. April 2025. (2021), Adresse: <https://www.statista.com/statistics/1300299/how-many-indoor-plants-us-gardeners-kill-annually> (besucht am 14.05.2025).
- [6] K. Mayers. „Houseplant Statistics in 2024 (incl. Covid & Millennials).“ Abgerufen am 14. April 2025. (2022), Adresse: <https://gardenpals.com/houseplant-statistics/> (besucht am 14.05.2025).
- [7] E. D. Schmid. „Deutschland, deine Gärten: Spannende Zahlen und Fakten.“ Abgerufen am 14. April 2025. (2024), Adresse: <https://wohnglueck.de/artikel/gaerten-in-deutschland-zahlen-36505> (besucht am 14.05.2025).
- [8] Gardena. „GARDENA Studienpräsentation 2022: Nachhaltige Lösungen und technologischer Fortschritt im Garten helfen gegen den Klimawandel.“ Abgerufen am 14. April 2025. (2022), Adresse: <https://www.media-gardena.com/news-gardena-studienpraesentation-2022-nachhaltige-loesungen-und-technologischer->

- fortschritt-im-garten-helfen-gegen-den-klimawandel?id=150284 (besucht am 14. 04. 2025).
- [9] B. Cardenas, M. Dukes und G. Miller, „Sensor-Based Automation of Irrigation on Bermudagrass during Dry Weather Conditions,“ *Journal of Irrigation and Drainage Engineering-asce - J IRRIG DRAIN ENG-ASCE*, Jg. 136, März 2010. DOI: 10.1061/(ASCE)IR.1943-4774.0000153.
 - [10] Statista. „Germany: Urbanization from 2013 to 2023.“ Abgerufen am 14. April 2025. (2025), Adresse: <https://www.statista.com/statistics/455825/urbanization-in-germany> (besucht am 14. 04. 2025).
 - [11] markemarketsandmarkets. „European Smart Homes Market Size, Share & Growth.“ Abgerufen am 14. April 2025. (2025), Adresse: <https://www.marketsandmarkets.com/Market-Reports/european-smart-homes-market-1290.html> (besucht am 14. 04. 2025).
 - [12] C. Drumond. „Das Agile Manifest.“ de, Atlassian. (), Adresse: <https://www.atlassian.com/de/agile manifesto> (besucht am 09. 08. 2024).
 - [13] e. a. Kent Beck Mike Beedle. „Manifest für Agile Softwareentwicklung.“ de. (2001), Adresse: <https://agilemanifesto.org/iso/de/manifesto.html> (besucht am 09. 08. 2024).
 - [14] H. Wolf und W.-G. Bleek, *Agile Softwareentwicklung - Werte, Konzepte und Methoden*. Heidelberg: dpunkt.verlag, 2011, ISBN: 978-3-864-91021-0.
 - [15] C. BasuMallick, „What Is a Single-Page Application? Architecture, Benefits, and Challenges,“ *Spiceworks Tech Articles*, 2022, Accessed 11 Apr 2025. Adresse: <https://www.spiceworks.com/tech/devops/articles/what-is-single-page-application/>.
 - [16] *Top 21 Vue.js Best Practices & Security Tips for 2025*, Bacancy Technology Blog, Accessed 02 Apr 2025, 2023. Adresse: <https://www.bacancytechnology.com/blog/vue-js-best-practices>.
 - [17] Meta Open Source, *React Documentation*, URL: <https://reactjs.org/>, 2025.
 - [18] Google, *Angular Documentation*, URL: <https://angular.io/docs>, 2025.
 - [19] Vue.js Core Team, *Vue.js - The Progressive JavaScript Framework*, URL: <https://vuejs.org/>, 2016.
 - [20] Vue.js Core Team, *Reactivity in Depth – Vue.js 2 Documentation*, Accessed 11 Apr 2025, 2016. Adresse: <https://v2.vuejs.org/v2/guide/reactivity.html>.

- [21] Interaction Design Foundation, „What is User Centered Design (UCD)? – Definition and Phases,“ 2025, Accessed 07 Apr 2025. Adresse: <https://www.interaction-design.org/literature/topics/user-centered-design>.
- [22] F. Guimarães, *Nielsen's Heuristics: 10 Usability Principles to Improve UI Design*, Aela.io Blog, Accessed 05 Apr 2025, 2021. Adresse: <https://www.aela.io/en/blog/all/10-usability-heuristics-ui-design>.
- [23] S. Deterding, D. Dixon, R. Khaled und L. Nacke, „From Game Design Elements to Gamefulness: Defining Gamification,“ Bd. 11, Sep. 2011, S. 9–15. DOI: [10.1145/2181037.2181040](https://doi.org/10.1145/2181037.2181040).
- [24] K. Werbach und D. Hunter, *For the Win: How Game Thinking can Revolutionize your Business*. Jan. 2012.
- [25] A. Charland und B. Leroux, „Mobile application development: web vs. native,“ *Communications of the ACM*, Jg. 54, Nr. 5, S. 49–53, 2011. DOI: [10.1145/1941487.1941504](https://doi.org/10.1145/1941487.1941504).
- [26] M. Mahendra und B. Anggorojati, „Evaluating the performance of Android based Cross-Platform App Development Frameworks,“ in *Proceedings of the 6th International Conference on Communication and Information Processing*, Ser. ICCIP '20, Tokyo, Japan: Association for Computing Machinery, 2021, S. 32–37, ISBN: 9781450388092. DOI: [10.1145/3442555.3442561](https://doi.org/10.1145/3442555.3442561). Adresse: <https://doi.org/10.1145/3442555.3442561>.
- [27] G. J. Myers, T. Badgett und C. Sandler, *The Art of Software Testing*, 3rd. Wiley, 2011, ISBN: 978-1118031964.
- [28] P. Ammann und J. Offutt, *Introduction to Software Testing*, 2. Aufl. Cambridge University Press, 2016.
- [29] J. Humble und D. Farley, *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Addison-Wesley, 2010, ISBN: 978-0321601919.
- [30] Vue.js Core Team, *Vue Test Utils Documentation*, <https://test-utils.vuejs.org/>, 2024. (besucht am 12.04.2025).
- [31] C. Team, *Cypress Documentation*, <https://docs.cypress.io/>, 2024. (besucht am 13.04.2025).
- [32] R. D. John Au-Yeung. „Best practices for REST API design.“ (2. März 2020), Adresse: <https://stackoverflow.blog/2020/03/02/best-practices-for-rest-api-design/> (besucht am 19.08.2024).

- [33] M. Massee, *REST API Design Rulebook - Designing Consistent RESTful Web Service Interfaces*. Sebastopol: O'Reilly Media, Inc.", 2011, ISBN: 978-1-449-31990-8.
- [34] P. T. Eugster, P. A. Felber, R. Guerraoui und A.-M. Kermarrec, „The Many Faces of Publish/Subscribe,“ *ACM Computing Surveys*, Jg. 35, Nr. 2, S. 114–131, 2003. DOI: 10.1145/857076.857078.
- [35] Solace, *Supported Open APIs and Protocols*, Zugriff am 13. April 2025, 2024. Adresse: <https://docs.solace.com/Open-APIs-Protocols/open-apis.htm>.
- [36] Solace, *Direct and Guaranteed Messaging Overview*, Zugriff am 13. April 2025, 2024. Adresse: <https://docs.solace.com/Messaging/Messaging-Concepts.htm>.
- [37] Solace, *Topic Subscription Syntax and Wildcards*, Zugriff am 13. April 2025, 2024. Adresse: <https://docs.solace.com/PubSub-Basics/Wildcard-Charaters-Topic-Subs.htm>.
- [38] Solace, *Using MQTT with Solace PubSub+*, Zugriff am 13. April 2025, 2024. Adresse: <https://docs.solace.com/Features/Core-Features/MQTT.htm>.
- [39] Solace, *Solace Dapr MQTT Support*, Zugriff am 13. April 2025, 2024. Adresse: <https://docs.solace.com/Integrations/Dapr/Dapr-MQTT.htm>.
- [40] Solace, *About Access Control Lists (ACLs)*, Zugriff am 13. April 2025, 2024. Adresse: <https://docs.solace.com/Configuring-and-Managing/Configuring-ACLs.htm>.
- [41] A. Silica, *Securing MQTT with TLS and PSK on IoT Devices*, Zugriff am 13. April 2025, 2022. Adresse: <https://www.avnet.com/wps/portal/silica/resources/whitepapers/securing-mqtt-psk>.
- [42] Solace, *SEMP v2 Overview*, Zugriff am 13. April 2025, 2024. Adresse: <https://docs.solace.com/SEMP/Using-SEMP.htm>.
- [43] J. Kreps, N. Narkhede und J. Rao, „Kafka: A Distributed Messaging System for Log Processing,“ *NetDB*, 2011, Verfügbar unter: <https://kafka.apache.org/08/documentation.html>.
- [44] H. Krawczyk, M. Bellare und R. Canetti, *HMAC: Keyed-Hashing for Message Authentication*, RFC 2104, 1997. Adresse: <https://datatracker.ietf.org/doc/html/rfc2104>.
- [45] N. I. of Standards und T. (NIST), *The Keyed-Hash Message Authentication Code (HMAC)*, FIPS Publication 198-1, 2008. Adresse: <https://csrc.nist.gov/publications/detail/fips/198/1/final>.

- [46] D. M'Raihi, S. Machani, M. Pei und J. Rydell, *HOTP: An HMAC-Based One-Time Password Algorithm*, RFC 4226, 2005. Adresse: <https://datatracker.ietf.org/doc/html/rfc4226>.
- [47] H. Krawczyk, „The order of encryption and authentication for protecting communications (or: how secure is SSL?)“ *Advances in Cryptology — CRYPTO '01*, 2001. DOI: [10.1007/3-540-44647-8_8](https://doi.org/10.1007/3-540-44647-8_8).
- [48] Cryptography.io, *Fernet (symmetric encryption)*, Zugriff am 13. April 2025, 2024. Adresse: <https://cryptography.io/en/latest/fernet/>.
- [49] F. Authors, *Fernet Token Format Specification*, Zugriff am 13. April 2025, 2014. Adresse: <https://github.com/fernet/spec/blob/master/Spec.md>.
- [50] Totalphase. „Microcontroller vs Microprocessor - What are the Differences?“ Abgerufen am 14. April 2025. (2019), Adresse: <https://www.totalphase.com/blog/2019/12/microcontroller-vs-microprocessor-what-are-the-differences/> (besucht am 14.04.2025).
- [51] Curatepartners. „Understanding Low Power Modes.“ Abgerufen am 14. April 2025. (2024), Adresse: <https://curatepartners.com/blogs/skills-tools-platforms/understanding-low-power-modes-enhancing-energy-efficiency-in-modern-electronics> (besucht am 14.04.2025).
- [52] P. D. C. Siemers. „Echtzeit: Grundlagen von Echtzeitsystemen.“ Abgerufen am 14. April 2025. (2017), Adresse: <https://www.embedded-software-engineering.de/echtzeit-grundlagen-von-echtzeitsystemen-a-5897f8abe8f52370ee04a724b23339f8/> (besucht am 14.04.2025).
- [53] B. M. Al-Hashimi, *System-on-a-Chip*. Institution of Engineering und Technology, 2006, ISBN: 978-0-86341-552-4.
- [54] IETF. „The Internet of Things.“ Abgerufen am 14. April 2025. (), Adresse: <https://www.ietf.org/technologies/iot/> (besucht am 14.04.2025).
- [55] jawaharlal-nehru-technological-university-hyderabad. „Internet of Things.“ Abgerufen am 14. April 2025. (), Adresse: <https://www.studocu.com/in/document/jawaharlal-nehru-technological-university-hyderabad/internet-of-things/iotcpa-question-bank/114791708> (besucht am 14.04.2025).
- [56] I. Craggs. „MQTT Vs CoAP for IoT.“ Abgerufen am 14. April 2025. (2022), Adresse: <https://www.hivemq.com/blog/mqtt-vs-coap-for-iot/> (besucht am 14.04.2025).

- [57] N. Fernando, S. Shrestha, S. W. Loke und K. Lee, „On Edge-Fog-Cloud Collaboration and Reaping Its Benefits: A Heterogeneous Multi-Tier Edge Computing Architecture,“ *Future Internet*, Jg. 17, Nr. 1, 2025. doi: 10.3390/fi17010022.
- [58] cloudflare. „Was ist IoT-Sicherheit?“ Abgerufen am 14. April 2025. (), Adresse: <https://www.cloudflare.com/de-de/learning/security/glossary/iot-security/> (besucht am 14.04.2025).
- [59] proofpoint. „Was ist IoT-Security?“ Abgerufen am 14. April 2025. (), Adresse: <https://www.proofpoint.com/de/threat-reference/iot-security> (besucht am 14.04.2025).
- [60] moocit. „IoT Security: The Ultimate Guide.“ Abgerufen am 14. April 2025. (2024), Adresse: https://moocit.de/index.php/Technik_-_Steuern_und_Regeln (besucht am 14.04.2025).
- [61] haustechnik. „Der Unterschied zwischen Steuerung und Regelung.“ Abgerufen am 14. April 2025. (2024), Adresse: <https://www.haustechnikverstehen.de/der-unterschied-zwischen-steuerung-und-regelung/> (besucht am 14.04.2025).
- [62] Wikipedia. „Proportional integral derivative controller.“ Abgerufen am 14. April 2025. (2025), Adresse: https://en.wikipedia.org/wiki/Proportional%E2%80%93integral%E2%80%93derivative_controller (besucht am 14.04.2025).
- [63] Wikipedia. „Regeler.“ Abgerufen am 14. April 2025. (2025), Adresse: <https://de.wikipedia.org/wiki/Regler> (besucht am 14.04.2025).
- [64] Vue.js Core Team, *Vue.js Guide*, URL: <https://vuejs.org/guide/introduction.html>, 2024. Adresse: <https://vuejs.org/guide/introduction.html> (besucht am 14.04.2025).
- [65] Vue.js Core Team, *Composition API: Introduction*, 2020. Adresse: <https://vuejs.org/guide/reusability/composables.html> (besucht am 14.04.2025).
- [66] C. Allotey, „Pinia vs Vuex – Why Pinia wins,“ *Vue.js Developers Newsletter*, 2023, Accessed 10 Apr 2025. Adresse: <https://vuejsdevelopers.com/2023/04/11/pinia-vs-vuex--why-pinia-wins/>.
- [67] V. C. Team, *Composition API FAQ*, 11. März 2025. Adresse: <https://vuejs.org/guide/extras/composition-api-faq.html>.
- [68] V. C. Team, *<script setup>*, 11. März 2025. Adresse: <https://vuejs.org/api/sfc-script-setup.html>.
- [69] D. Nwadiogbu, *Refresh-Proof your Pinia Stores*, Vue Mastery Blog, Accessed 10 Apr 2025, 2023. Adresse: <https://www.vuemastery.com/blog/refresh-proof-your-pinia-stores/>.

- [70] *Tailwind CSS – A utility-first CSS framework*, <https://tailwindcss.com>, Accessed 11 Apr 2025.
- [71] *shadcn-vue: Vue port of shadcn/ui*, <https://shadcn-vue.com>, Accessed 08 Apr 2025, 2023.
- [72] F. Stallmann und U. Wegner, *Internationalisierung von E-Commerce-Geschäften*. Springer Vieweg, 2015. DOI: 10.1007/978-3-658-06782-3.
- [73] Kazupon, *Vue I18n Documentation*, 2024. Adresse: <https://vue-i18n.intlify.dev/> (besucht am 14. 04. 2025).
- [74] I. Krukowski, *Vue 3 i18n: Building a multi-language app with locale switcher*, 2024. Adresse: <https://lokalise.com/blog/vue-i18n/> (besucht am 14. 04. 2025).
- [75] *Vuex*, 12. März 2025. Adresse: <https://vuex.vuejs.org/>.
- [76] M. Singh und G. Shobha, „Comparative Analysis of Hybrid Mobile App Development Frameworks,“ Juli 2021. DOI: 10.35940/ijscse.F3518.0710621.
- [77] S. Giordano, „Ionic/Capacitor for Mobile App Development: Why You Should Consider Alternatives (Part 1),“ *Medium (@simonegiordano)*, 2024, Accessed 06 Apr 2025. Adresse: <https://simonegiordano.medium.com/ionic-capacitor-for-mobile-app-development-why-you-should-consider-alternatives-part-1-131cfe2e4410>.
- [78] I. Team, *Capacitor Documentation*, 2024. Adresse: <https://capacitorjs.com/docs>.
- [79] V. Prokopiou. „Web api benchmarking: Go (Fiber) vs Rust (actix-web).“ en. (19. Dez. 2021), Adresse: <https://youtu.be/MY30u5ehwUQ> (besucht am 13. 04. 2025).
- [80] JetBrains. „Developer Ecosystem - Go.“ (2023), Adresse: <https://www.jetbrains.com/lp/devcosystem-2023/go/> (besucht am 16. 08. 2024).
- [81] Kuree, *The Go Programming Language Report*. GitHub. Adresse: <https://kuree.gitbooks.io/the-go-programming-language-report/content/index.html>.
- [82] R. Pike. „Go at Google: Language Design in the Service of Software Engineering.“ (), Adresse: <https://go.dev/talks/2012/splash.article> (besucht am 16. 08. 2024).
- [83] A. Merrick. „Go Developer Survey 2023 Q1 Results.“ (11. Mai 2023), Adresse: <https://go.dev/blog/survey2023-q1-results> (besucht am 16. 08. 2024).
- [84] +. w. mark-i-m. „Rust Compiler Development Guide.“ en. Version 85c796488d2648a83a234ba821a (14. Apr. 2025), Adresse: <https://rustc-dev-guide.rust-lang.org/> (besucht am 14. 04. 2025).

- [85] C. N. Steve Klabnik, *Die Programmiersprache Rust*, 2. Auflage. Herbert Reiter, 16. März 2024, ISBN: 978-3-98871-003-1. Adresse: <https://rust-lang-de.github.io/rustbook-de/title-page.html>.
- [86] N. Matsakis. „Non-lexical lifetimes (NLL) fully stable.“ en. (5. Aug. 2022), Adresse: <https://blog.rust-lang.org/2022/08/05/nll-by-default.html> (besucht am 14.04.2025).
- [87] B. Wikipedia. „Rust (Programmiersprache).“ de. (3. Mai 2024), Adresse: [https://de.wikipedia.org/wiki/Rust_\(Programmiersprache\)](https://de.wikipedia.org/wiki/Rust_(Programmiersprache)) (besucht am 14.04.2025).
- [88] Bhavya. „Clash of the compiled: Golang vs Rust.“ en. (30. Nov. 2021), Adresse: <https://bhavya-saraf.medium.com/clash-of-the-compiled-golang-vs-rust-62595cb90377> (besucht am 14.04.2025).
- [89] D. puri. „Rust Ownership vs Garbage Collector: A Detailed Comparison with Code.“ en. (1. Jan. 2025), Adresse: https://dev.to/devratapuri/day-2-rust-ownership-vs-garbage-collector-a-detailed-comparison-with-code-52ad?utm_source=chatgpt.com (besucht am 14.04.2025).
- [90] P. C. Team. „Rust and Go performance comparison: Which one is better for high-performance apps?“ en. (5. Mai 2021), Adresse: https://proxify.io/articles/rust-and-go-performance-comparison?utm_source=chatgpt.com (besucht am 14.04.2025).
- [91] Abhinav. „Golang vs. Rust: The Ultimate Showdown for Performance and Safety.“ en. (29. März 2025), Adresse: <https://medium.com/%40abhinavv.singh/golang-vs-rust-the-ultimate-showdown-for-performance-and-safety-aa8b1336f512> (besucht am 14.04.2025).
- [92] dami. „Rust vs Go? Which Should You Learn in 2025.“ en. (27. Aug. 2024), Adresse: https://dev.to/thatcoolguy/rust-vs-go-which-should-you-choose-in-2024-50k5?utm_source=chatgpt.com (besucht am 14.04.2025).
- [93] A. Obregon. „Journey to Fearless Concurrency — Rust vs. Go.“ en. (6. Aug. 2023), Adresse: <https://medium.com/%40AlexanderObregon/journey-to-fearless-concurrency-rust-vs-go-31d49255d6b6> (besucht am 14.04.2025).
- [94] M. Kofler, *Datenbanksysteme - das umfassende Lehrbuch*. Bonn: Rheinwerk Verlag, 2024, ISBN: 978-3-367-10015-6.
- [95] L. Fröhlich, *PostgreSQL - Praxisbuch für Administratoren und Entwickler*. M: Carl Hanser Verlag GmbH Co KG, 2022, ISBN: 978-3-446-47315-7.

- [96] IamXander. „Storage Engine API.“ en. Version 35db3811d8f749edd5b79ba910adcbe1ceb54cc4. (6. Apr. 2024), Adresse: <https://github.com/mongodb/mongo/blob/master/src/mongo/db/storage/README.md> (besucht am 13.08.2024).
- [97] themattman. „Execution Internals.“ en. Version 17a11f90374c2f1f0a8f5d7844819c13721591b8. (8. Aug. 2024), Adresse: <https://github.com/mongodb/mongo/blob/master/src/mongo/db/catalog/README.md> (besucht am 13.08.2024).
- [98] P. Velikhov. „BSON - Specification Version 1.1.“ (), Adresse: <https://bsonspec.org/spec.html> (besucht am 13.08.2024).
- [99] A. Chauhan, „A Review on Various Aspects of MongoDB Databases,“ *International Journal of Engineering Research & Technology (IJERT)*, Jg. 08, Nr. 05, Mai 2019, ISSN: 2278-0181. Adresse: <https://www.ijert.org/a-review-on-various-aspects-of-mongodb-databases> (besucht am 13.08.2024).
- [100] P. S. Foundation, *Flask Web Framework*, Zugriff: April 2025, 2024. Adresse: <https://flask.palletsprojects.com>.
- [101] H. Krawczyk, M. Bellare und R. Canetti, „HMAC: Keyed-Hashing for Message Authentication,“ *RFC 2104, IETF*, 1997. Adresse: <https://tools.ietf.org/html/rfc2104>.
- [102] M. Foundation, *SMTP and Email Security Best Practices*, Zugriff: April 2025, 2023. Adresse: <https://infosec.mozilla.org/guidelines/email/>.
- [103] E. Foundation, *paho-mqtt: Python MQTT Client Library*, Zugriff: April 2025, 2024. Adresse: <https://www.eclipse.org/paho/index.php?page=clients/python/index.php>.
- [104] O. M. Standard, *MQTT Version 3.1.1 Specification*, Zugriff: April 2025, 2014. Adresse: <https://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html>.
- [105] F. Project, *Flasgger: Swagger UI for Flask APIs*, Zugriff: April 2025, 2023. Adresse: <https://github.com/flasgger/flasgger>.
- [106] S. Corporation, *Solace PubSub+ Overview: The Advanced Event Broker*, Zugriff: April 2025, 2024. Adresse: <https://solace.com/products/event-broker/overview/>.
- [107] E. Technologies, *MQTT vs. Kafka: Which is Better for IoT?* Zugriff: April 2025, 2023. Adresse: <https://www.emqx.com/en/blog/mqtt-vs-kafka-iot>.
- [108] S. Corporation, *Getting Started with Solace Messaging in Python*, Zugriff: April 2025, 2023. Adresse: <https://docs.solace.com/Features/python-api.htm>.

- [109] JSON.org, *Introducing JSON: A Lightweight Data-Interchange Format*, Zugriff: April 2025, 2022. Adresse: <https://www.json.org/json-en.html>.
- [110] Y. Goyal, „Comparative Study of Microcontroller: ARDUINO UNO, RASPBERRY PI 4, ESP 32,“ *International Journal for Research in Applied Science & Engineering Technology*, Jg. 12, Nr. 7, S. 588–592, 2024. doi: 10.22214/ijraset.2024.63598.
- [111] espressif. „ESP32-WROOM-32D and ESP32-WROOM-32U Datasheet.“ Abgerufen am 15. April 2025. (2025), Adresse: https://www.espressif.com/sites/default/files/documentation/esp32-wroom-32d_esp32-wroom-32u_datasheet_en.pdf (besucht am 15.04.2025).
- [112] M. T. Inc. „ATmega4808/4809 Data Sheet.“ Abgerufen am 15. April 2025. (2021), Adresse: <https://www.onlinocomponents.com/en/datasheet/atmega4809au-52119322/> (besucht am 15.04.2025).
- [113] Z. Didi und I. El Azami, „IoT, Comparative Study Between the Use of Arduino Uno, Esp32, and Raspberry pi in Greenhouses,“ in *Digital Technologies and Applications*, S. Motahhir und B. Bossoufi, Hrsg., Cham: Springer International Publishing, 2022, S. 718–726, ISBN: 978-3-031-02447-4.
- [114] R. P. (Ltd. „Raspberry Pi 4 Model B Datasheet.“ Abgerufen am 15. April 2025. (2024), Adresse: <https://datasheets.raspberrypi.com/rpi4/raspberry-pi-4-datasheet.pdf> (besucht am 15.04.2025).
- [115] R. P. (Ltd. „Raspberry Pi Pico W Datasheet.“ Abgerufen am 15. April 2025. (2024), Adresse: <https://datasheets.raspberrypi.com/picow/pico-w-datasheet.pdf> (besucht am 15.04.2025).
- [116] O. Lab. „ESP32 vs. Arduino vs. Raspberry Pi Pico: Was ist besser?“ Abgerufen am 15. April 2025. (2024), Adresse: <https://openelab.io/de/blogs/learn/esp32-vs-arduino-vs-raspberry-pi-pico-which-is-better?srsltid=AfmB0opiSmQCJDHVula3KxkMyY1Ba02SJEFFdnJZa3n7CBXCnnyUL6mN> (besucht am 15.04.2025).
- [117] I. Shields. „How to use ESP32 NVS memory to store data permanently.“ Abgerufen am 15. April 2025. (2023), Adresse: <https://www.industrialshields.com/blog/arduino-industrial-1/how-to-use-esp32-nvs-memory-to-store-data-permanently-550> (besucht am 15.04.2025).
- [118] P. E. Academy. „A Practical Guide to ESP32 Deep Sleep Modes.“ Abgerufen am 15. April 2025. (), Adresse: <https://www.programmingelectronics.com/esp32-deep-sleep-mode> (besucht am 15.04.2025).

- [119] DigiKey. „ESP32-WROOM-32-N4.“ Abgerufen am 15. April 2025. (2025), Adresse: https://www.digikey.de/en/products/detail/espressif-systems/ESP32-WROOM-32-N4/8544298?srsltid=AfmB0opq_74Fcavp64CDj91WpsWYXsbfWneae9bHzyxsZiKKPap (besucht am 15.04.2025).
- [120] A. Store. „Arduino Nano Every.“ Abgerufen am 15. April 2025. (2025), Adresse: https://store.arduino.cc/en-de/products/arduino-nano-every?srsltid=AfmB0oo3nq0p1Nq71Z4c09na8fh1I-N1E829_YLhGob5q99nfr8bXcif (besucht am 15.04.2025).
- [121] Wetterott. „Raspberry Pi 4 Modell B 4Gb RAM- 64Bit 1.5GHz Quad-Core ARM-Cortex-A72.“ Abgerufen am 15. April 2025. (2025), Adresse: https://shop.wetterott.com/Raspberry-Pi-4-Modell-B-4Gb-RAM-64Bit-15GHz-Quad-Core-ARM-Cortex-A72_1 (besucht am 15.04.2025).
- [122] R. Barnes. „Introducing Raspberry Pi Zero W.“ Abgerufen am 15. April 2025. (2017), Adresse: <https://magazine.raspberrypi.com/articles/pi-zero-w> (besucht am 15.04.2025).
- [123] ars Technica. „Raspberry Pi availability is visibly improving after years of shortages.“ Abgerufen am 15. April 2025. (2025), Adresse: <https://arstechnica.com/gadgets/2023/08/some-shops-will-let-you-buy-more-than-one-raspberry-pi-at-a-time-again/> (besucht am 15.04.2025).
- [124] E. Systems, *ESP-IDF Programming Guide*, Zugriff am 15.04.2025, Espressif Systems, 2024. Adresse: <https://docs.espressif.com/projects/esp-idf>.
- [125] Arduino und Espressif, *ESP32 Arduino Core GitHub Repository*, Zugriff am 15.04.2025, 2024. Adresse: <https://github.com/espressif/arduino-esp32>.
- [126] M. Community, *MicroPython für ESP32 – Dokumentation*, Zugriff am 15.04.2025, 2024. Adresse: <https://docs.micropython.org/en/latest/esp32/quickref.html>.
- [127] P. Labs, *PlatformIO Core Documentation*, Zugriff am 15.04.2025, 2024. Adresse: <https://docs.platformio.org>.
- [128] R. T. E. Ltd., *FreeRTOS Kernel Documentation*, Zugriff am 15.04.2025, 2024. Adresse: https://www.freertos.org/Documentation/RTOS_book.html.
- [129] E. Systems, *ESP32 Technical Reference Manual*, Zugriff am 15.04.2025, 2024. Adresse: https://www.espressif.com/sites/default/files/documentation/esp32_technical_reference_manual_en.pdf.
- [130] Adafruit Industries, *DHT11 Temperature and Humidity Sensor Guide*, Zugriff am 15.04.2025, 2023. Adresse: <https://learn.adafruit.com/dht>.

- [131] Bosch Sensortec, *BME280 Combined Humidity and Pressure Sensor - Datasheet*, Zugriff am 15.04.2025, 2023. Adresse: <https://www.bosch-sensortec.com/media/boschsensortec/downloads/datasheets/bst-bme280-ds002.pdf>.
- [132] Aosong Electronics Co., Ltd., *DHT11 Temperature and Humidity Sensor - Datasheet (DFR0067)*, Zugriff am 15.04.2025, 2023. Adresse: <https://www-aosong.com>.
- [133] Handson Technology, *BH1750 Light Sensor Module – Datasheet*, Zugriff am 15.04.2025, 2021. Adresse: <https://www.handsontec.com/dataspecs/sensor/BH1750%20Light%20Sensor.pdf>.
- [134] Components101, *Soil Moisture Sensor Module – Pinout, Features and Specifications*, Zugriff am 15.04.2025, 2020. Adresse: <https://components101.com/modules/soil-moisture-sensor-module>.
- [135] Texas Instruments, *LM393, LM293, LM2903 – Dual Comparators – Technical Datasheet*, Zugriff am 15.04.2025, 2025. Adresse: <https://www.ti.com/product/LM393>.
- [136] Songle Relay, *SRD-05VDC-SL-C – Relay Module Datasheet*, Zugriff am 15.04.2025, 2015. Adresse: <https://www.circuitbasics.com/wp-content/uploads/2015/11/SRD-05VDC-SL-C-Datasheet.pdf>.
- [137] Opto 22, *Solid-State Relays (SSRs) Data Sheet – Z Series*, Zugriff am 15.04.2025, 2024. Adresse: https://documents.opto22.com/0859_Solid_State_Relays_data_sheet.pdf.
- [138] Wolle. „Der MOSFET als Schalter.“ Zugriff am 15.04.2025. (2022), Adresse: <https://wolles-elektronikkiste.de/der-mosfet-als-schalter>.
- [139] S. Newman, *Building Microservices: Designing Fine-grained Systems*. O'Reilly, 2021, ISBN: 9781492034025. Adresse: <https://books.google.de/books?id=MTClswEACAAJ>.
- [140] N. Dragoni, S. Giallorenzo, A. L. Lafuente et al., „Microservices: Yesterday, Today, and Tomorrow,“ in *Present and Ulterior Software Engineering*, M. Mazzara und B. Meyer, Hrsg. Cham: Springer International Publishing, 2017, S. 195–216, ISBN: 978-3-319-67425-4. DOI: 10.1007/978-3-319-67425-4_12. Adresse: https://doi.org/10.1007/978-3-319-67425-4_12.
- [141] T. B. David, „Why Flux,“ *Medium*, 28. März 2025. Adresse: <https://medium.com/@Tom1212121/why-flux-2d25ab8a8063>.

- [142] F. Inc., *Flux: Application Architecture for Building User Interfaces*, Zugriffen: 13. April 2025, 2014. Adresse: <https://facebookarchive.github.io/flux/docs/in-depth-overview>.
- [143] V. Team, *Vite - Next Generation Frontend Tooling*, <https://vitejs.dev/>, 2024. (besucht am 13.04.2025).
- [144] R. Team, *Rollup Documentation*, <https://rollupjs.org/>, 2024. (besucht am 13.04.2025).
- [145] *Einführung in Lighthouse*. Adresse: <https://developer.chrome.com/docs/lighthouse/overview>.
- [146] D. A. Norman, *The design of everyday things: Revised and expanded edition*. Basic books, 2013.
- [147] A. Affouard, H. Goëau, P. Bonnet, J.-C. Lombardo und A. Joly, „Pl@ntNet app in the era of deep learning,“ in *International Conference on Learning Representations*, 2017. Adresse: <https://api.semanticscholar.org/CorpusID:52212860>.
- [148] K. He, X. Zhang, S. Ren und J. Sun, *Deep Residual Learning for Image Recognition*, 2015. arXiv: 1512.03385 [cs.CV]. Adresse: <https://arxiv.org/abs/1512.03385>.
- [149] S. Kornblith, J. Shlens und Q. V. Le, *Do Better ImageNet Models Transfer Better?* 2019. arXiv: 1805.08974 [cs.CV]. Adresse: <https://arxiv.org/abs/1805.08974>.
- [150] S. J. Pan und Q. Yang, „A Survey on Transfer Learning,“ *IEEE Transactions on Knowledge and Data Engineering*, Jg. 22, Nr. 10, S. 1345–1359, 2010. DOI: 10.1109/TKDE.2009.191.
- [151] H. Zhang, M. Cisse, Y. N. Dauphin und D. Lopez-Paz, *mixup: Beyond Empirical Risk Minimization*, 2018. arXiv: 1710.09412 [cs.LG]. Adresse: <https://arxiv.org/abs/1710.09412>.
- [152] S. Yun, D. Han, S. J. Oh, S. Chun, J. Choe und Y. Yoo, *CutMix: Regularization Strategy to Train Strong Classifiers with Localizable Features*, 2019. arXiv: 1905.04899 [cs.CV]. Adresse: <https://arxiv.org/abs/1905.04899>.
- [153] G. V. Horn, O. M. Aodha, Y. Song et al., *The iNaturalist Species Classification and Detection Dataset*, 2018. arXiv: 1707.06642 [cs.CV]. Adresse: <https://arxiv.org/abs/1707.06642>.

- [154] M. Buda, A. Maki und M. A. Mazurowski, „A systematic study of the class imbalance problem in convolutional neural networks,“ *Neural Networks*, Jg. 106, S. 249–259, Okt. 2018, ISSN: 0893-6080. DOI: 10.1016/j.neunet.2018.07.011. Adresse: <http://dx.doi.org/10.1016/j.neunet.2018.07.011>.
- [155] P. S. Foundation, *secrets — Generate secure random numbers for managing secrets*, Accessed April 14, 2025, 2024. Adresse: <https://docs.python.org/3/library/secrets.html>.
- [156] M. Krawczyk, H. Krawczyk und H. Shacham, *HMAC: Keyed-Hashing for Message Authentication (RFC 2104)*, Accessed April 14, 2025, 1997. Adresse: <https://www.rfc-editor.org/rfc/rfc2104>.
- [157] P. C. Project, *Fernet (symmetric encryption) specification*, Accessed April 14, 2025, 2024. Adresse: <https://cryptography.io/en/latest/fernet/>.
- [158] S. Ramírez, *FastAPI - The modern Python web framework*, Accessed April 14, 2025, 2024. Adresse: <https://fastapi.tiangolo.com/>.
- [159] S. Corporation, *Solace PubSub+ SEMP API v2 Documentation*, Accessed April 14, 2025, 2024. Adresse: <https://docs.solace.com/API-Developer-Online-Ref-Documentation/swagger-ui/config/index.html>.
- [160] O. Foundation, *Pre-Shared Key (PSK) Authentication in IoT Systems*, Accessed April 14, 2025, 2023. Adresse: <https://owasp.org/www-project-internet-of-things/>.

Autorenübersicht:

Kapitel	Kapitelüberschrift	Autor	# Seiten
	abstract	-	
1	Einleitung	Timon	-
1.1	Motivation	Timon	3
1.2	Zielsetzung	Timon	1
2	Theoretische Grundlagen	-	
2.1	Scrum und das Agile Manifest	Max	3
2.2	Single-page applications und Frameworks	Justus	2
2.3	Benutzerzentriertes Design und UI/UX im Frontend	Justus	2
2.4	Theoretischer Vergleich: Webanwendungen versus native Apps	Justus	2
2.5	Softwaretestens mit Fokus auf Frontend-Frameworks	Justus	3
2.6	REST APIs: Grundlagen und Best Practices	Max	3
2.7	Designprinzipien und -muster	Max	3
2.8	Solace PubSub+ Event Broker	Fynn	7.5
2.9	Sicherheitsrelevante Technologien im Auth-Service	Fynn	7.5
2.9	Microcontroller	Timon	4
2.10	IOT/Smarthome	Timon	5
2.11	Regelungstechnik in der Software	Timon	3
3	Anforderungen	-	
3.1	Anforderungsdefinition für das Frontend	Justus	1,5
3.2	Anforderungen an die	Fynn	5

	entwickelten Schnittstellen-Services im Projekt Sensora		
3.3	Anforderungen an das Backend	Max	2
3.4	Anforderungen an die Datenbank	Max	1
3.5	Anforderungen an das IOT Gerät	Lukas	5
3.6	Anforderungen an die Peripheriegeräte	Lukas	4
4	Auswahl der Technologien	-	
4.1	Auswahl von Vue.js für die Implementierung	Justus	4
4.2	CSS-Tailwind - Shadcn-vue	Justus	1,5
4.3	Internationalisierung der Anwendung in Vue.js	Justus	1
4.4	State-Management mit Pinia und Persistenz	Justus	1,5
4.5	Mobile Kompilierung mit Capacitor	Justus	2
4.6	Wahl der Programmiersprache	Max	7
4.7	Datenbankentscheidungen	Max	11
4.8	Auswahl der Technologien für die entwickelten Schnittstellen-Services	Fynn	10.5
4.9	Auswahl Microcontroller	Timon	22
4.10	Framework, Programmiersprache & IDE Microcontroller	Lukas	4,5
5	Umsetzung	-	
5.1	Servicearchitektur	Justus	2,5
5.2	Frontendarchitektur und Datenflüsse im System	Justus	2,5
5.3	Benutzerzentriertes Design und UI/UX im Frontend	Justus	3

5.4	Erweiterte Frontend-Techniken	Justus	1
5.5	Aufbau einer spezifischen View als Vertreter	Justus	3
5.6	KI-Komponente zur automatisierten Pflanzenklassifikation	Justus	4
5.7	Beschreibung des Datenbankaufbaus	Max	3
5.8	Auth-Service: Geräteregistrierung und HMAC-Authentifizierung	Fynn	4.5
5.9	Mail-Service: E-Mail-Verifikation von Benutzerkonten	Fynn	2
5.10	Database Writer: MQTT-Datenpersistierung in PostgreSQL	Fynn	4.5
5.11	Setpoint API: Sollwert-Vorgabe via REST und MQTT	Fynn	2
5.12	Solace Init: Automatisierte Broker-Konfiguration	Fynn	2
5.13	Programmierung des ESP	Timon	1
5.13.1	Initialisierung des ESP	Timon	8
5.13.2	Regelbetrieb des ESP	Lukas	12
6	Kritische Reflektion	-	
6.1	Reflexion zur Frontend-Umsetzung	Justus	1,5
6.2	Microcontroller	Lukas	4
6.3	Kritische Reflexion Auth-Service	Fynn	1
6.4	Kritische Reflexion Mailservice	Fynn	1
6.5	Kritische Reflexion Database Writer	Fynn	1
6.6	Kritische Reflexion Setpoint-API	Fynn	1.5
6.7	Kritische Reflexion Solace	Fynn	1.5
7	Ausblick	Lukas	1

7.1	Ausblick Frontend	Justus	1
7.2	Microcontroller	Lukas	2

Seiten nach Autor

Autor	Max	Justus	Fynn	Lukas	Timon
Seiten Gesamt	32	38,5	51	32,5	48

Hauptverantwortlichkeiten:

- Frontend: Justus
- Backend: Max
- MS-Mail: Fynn
- MS-Authservice/ Databasewriter: Fynn
- Mikrocontroller: Timon und Lukas
- KI-Komponente: Timon, Lukas, Justus