

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**

14. April 2025

Erklärung der Eigenleistung

Hiermit erklären wir, dass wir die vorliegende Studienarbeit selbstständig und ohne fremde Hilfe verfasst haben. Wir habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt.

Abstract

Abstrakt

Inhaltsverzeichnis

Abstract	ii
Abstrakt	iii
Abkürzungsverzeichnis	vii
Abbildungsverzeichnis	viii
Tabellenverzeichnis	ix
1 Einleitung	1
1.1 Motivation	1
1.2 Zielsetzung	2
1.3 Ziel der Arbeit	3
2 Theoretische Grundlagen	4
2.1 Single-page applications und Frameworks	4
2.1.1 Entstehung und Motivation von Single-page applications (SPAs)	4
2.1.2 Architektur, Vorteile und Herausforderungen von SPAs	4
2.1.3 Vergleich von React, Angular und Vue	5
2.2 Theoretischer Vergleich: Webanwendungen versus native Apps	5
2.2.1 Architekturmodelle mobiler und Webanwendungen	6
2.2.2 Vergleich von Webanwendungen und nativen Android-Apps	6
2.2.3 Kombinationsstrategien	7
2.3 Benutzerzentriertes Design und UI/UX im Frontend	7
2.3.1 User-Centered Design (UCD)	7
2.3.2 Usability-Heuristiken nach Nielsen	8
2.3.3 Gamification im UI/UX-Kontext	9
2.4 Theoretische Grundlagen des Softwaretestens mit Fokus auf Frontend-Frameworks	9
2.4.1 Ziele und Nutzen von Softwaretests	10
2.4.2 Testarten und deren Abgrenzung	10
2.4.3 Frontend-Testing mit Vue.js	11
2.4.4 Beispielhafte Unit-Test-Spezifikation mit Vue Test Utils	11
2.4.5 Beispielhafter Cypress-Test zur End-to-End-Verifikation	11

2.4.6	Codeabdeckung und Testmetriken	12
2.4.7	Einordnung und Fazit	12
3	Anforderungen	13
3.1	Anforderungsdefinition für das Frontend	13
4	Auswahl der Technologien	15
4.1	Auswahl von Vue.js für die Implementierung	15
4.1.1	Modularität und Komponentenstruktur	15
4.1.2	Reaktivität und Datenbindung	15
4.1.3	Community, Dokumentation und Lernkurve	15
4.1.4	Flexibilität, Integration und Zukunftssicherheit	16
4.1.5	Vergleich: Options API vs. Composition API in Vue.js	16
4.1.6	Einschränkungen und Gegenmaßnahmen	18
4.2	State-Management mit Pinia und Persistenz	19
4.2.1	Einordnung von State Management in SPAs	19
4.2.2	Persistenz mit <code>pinia-plugin-persistedstate</code>	19
4.2.3	Bewertung und Grenzen	19
4.3	Einsatz von Tailwind CSS und Shadcn-Vue	20
4.3.1	Tailwind CSS: Utility-First-Ansatz	20
4.3.2	Shadcn-Vue: Komponentenbibliothek für Vue 3	21
4.3.3	Kombination beider Technologien im Frontend-Projekt	21
4.4	Mobile Kompilierung mit Capacitor	21
4.4.1	Funktionsweise von Capacitor	21
4.4.2	Nutzung nativer APIs	22
4.4.3	Besonderheiten und Herausforderungen	22
4.4.4	Vor- und Nachteile der Verwendung von Capacitor	22
4.4.5	Bewertung für das Projekt	23
4.5	Internationalisierung der Anwendung in Vue.js	23
4.5.1	Motivation und Bedeutung	23
4.5.2	Technologiewahl: <code>vue-i18n</code>	23
4.5.3	Fazit	24
5	Umsetzung	25
5.1	Frontendarchitektur und Datenflüsse im System	25
5.1.1	Komponentenbasierte Struktur und Navigationsmodell	25
5.1.2	Pinia als Vermittlungsinstanz zwischen API und Frontend	26
5.1.3	Datenfluss nach dem Flux-Prinzip	26

5.1.4	Fazit	27
5.2	Benutzerzentriertes Design und UI/UX im Frontend	27
5.2.1	Anwendung von UCD im smarten Bewässerungssystem	27
5.2.2	Verwendete Heuristiken in der Anwendung	28
5.2.3	Responsive Design	28
5.2.4	User Stories und funktionale Umsetzung	29
5.3	Erweiterte Frontend-Techniken	29
5.3.1	Lazy Loading in der Anwendung	29
5.3.2	Frontend-Messung mit Lighthouse	30
5.4	KI-Komponente zur automatisierten Pflanzenklassifikation	30
5.4.1	Modellarchitektur und Trainingsstrategie	30
5.4.2	Regularisierung und Datenkonsolidierung	31
5.4.3	Leistung und Interpretation	32
5.4.4	Rolle der KI-Komponente bei der Pflanzenerstellung	34
5.5	Aufbau einer spezifischen View als Vertreter	34
6	Kritische Reflexion	37
6.1	Reflexion zur Frontend-Umsetzung	37
7	Ausblick	39
7.1	Ausblick Frontend	39
Literaturverzeichnis		40
Anhang		45

Abkürzungsverzeichnis

SPA	Single-page application
REST	Representational State Transfer
SEO	Search-Engine Optimization
SSR	Server-Side-Rendering
DOM	Document Object Model
UCD	User-Centered Design
UI	User Interface
UX	User Experience
POC	Proof of Concept
TDD	Test-Driven Development
KI	Künstliche Intelligenz
SFC	Single File Component
ResNet50	50-layer Residual Network
ResNet	Residual Network

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

1.1 Motivation

1.2 Zielsetzung

1.3 Ziel der Arbeit

2 Theoretische Grundlagen

2.1 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 [1].

2.1.1 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 [1].

2.1.2 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 La-dezeiten 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:

- **Schwächer:** 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 [2].

2.1.3 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 [3].

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 [4].

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 [5]. Die Reaktivierung erfolgt über ein Proxy-basiertes System, das automatische DOM-Updates bei Datenänderungen ermöglicht [6].

2.2 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).

2.2.1 Architekturmodelle 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 [7].
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 [7].
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 [8].

2.2.2 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 [7].

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) [8].

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 [8].

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 [7].

2.2.3 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 [8].

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.

2.3 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. Gera-de 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 [9]. 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 [10]. 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 komplexere 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 [11].

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[11], [12].

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 [11].

2.4 Theoretische Grundlagen des 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.

2.4.1 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 [13]. 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 [14]. 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.

2.4.2 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 [15].

2.4.3 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 [16]. 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 [16].

2.4.4 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})

```

Beispiel 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 [16].

2.4.5 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   })

```

Beispiel 2.2: Beispielhafter Cypress-Test

Ein vollständiger Cypress-Test wurde im Rahmen des Prototyps nicht umgesetzt, kann aber als Vorlage für spätere Implementierungen dienen [17].

2.4.6 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.

2.4.7 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.

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.

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 [18]. 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 [19]. 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 [6]. 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 [20]. Die offizielle Dokumentation von Vue gilt als eine der besten im Bereich der Webframeworks und trägt wesentlich zur schnellen Produktivität bei [18]. 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 [18].

```

1 export default {
2   data() {
3     return {
4       counter: 0
5     }
6   },

```

```

7   methods: {
8     increment() {
9       this.counter++
10    }
11  }
12 }
```

Beispiel 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 [18].

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 [21].

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 [22].

```

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

Beispiel 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 [19].

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 [21].

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 [21].

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 [23].

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 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.

4.2.1 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 [20], [24].

4.2.2 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 [23]. Jeder Store konfiguriert explizit, ob und wiepersistiert wird.

4.2.3 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 Backbone-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.

4.3 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 [25], [26].

4.3.1 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 [25].

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.

4.3.2 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 [26].

Modularer Aufbau Im Gegensatz zu anderen UI-Bibliotheken installiert Shadcn-Vue Komponenten selektiv. Dies führt zu einer schlankerem 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

4.3.3 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 [10], [25].

4.4 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.

4.4.1 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 [27], [28]. 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.

4.4.2 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 [27]. Diese APIs werden durch offizielle `@capacitor`-Plugins bereitgestellt und erlauben den Zugriff auf Systemfunktionen, ohne dass plattformspezifischer Code geschrieben werden muss.

4.4.3 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 [29]. 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 [27].

4.4.4 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 [27]. 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 [28]. Auch ist der Zugriff auf bestimmte

Systemfunktionen eingeschränkt, und die Pflege des nativen Projekts setzt Kenntnisse in Android Studio und dem Android-Entwicklungsprozess voraus.

4.4.5 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.

4.5 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 [30].

4.5.1 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 [30].

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 [31].

4.5.2 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 [31]. 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 [32].

4.5.3 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.

5 Umsetzung

5.1 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.

5.1.1 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 [18].

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.

5.1.2 Pinia als Vermittlungsinstanz zwischen API und Frontend

Zur zentralen Zustandsverwaltung kommt Pinia zum Einsatz, welches das offizielle State-Management-System für Vue 3 darstellt [20], [24]. 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 Backend 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.

5.1.3 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 [33], [34].

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 [33]. 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.

5.1.4 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.

5.2 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.

5.2.1 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 skizzierten. 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.

5.2.2 Verwendete Heuristiken in der Anwendung

Im Rahmen der konkreten Umsetzung wurden vier der zehn Heuristiken gezielt adressiert mit der Oberfläche:

- **Sichtbarkeit des Systemstatus:** Ladeindikatoren und Sensorstatusanzeigen geben kontinuierliches Feedback über Systemvorgänge.
- **Konsistenz und Standards:** Einheitliche Icons und Farbkonzepte gemäß Tailwind-Designprinzipien unterstützen ein kohärentes Erscheinungsbild [25].
- **Fehlervorbeugung:** Formulare verhindern fehlerhafte Eingaben durch Validierung und Feedback in Echtzeit.
- **Hilfe und Dokumentation:** Tooltips und leere Zustandsanzeigen dienen der kontextsensitiven Orientierung.

Darüber hinaus wurde auch die Heuristik **Entsprechung zwischen System und realer Welt** (Heuristik 2) durch das Objektdesign berücksichtigt. Die hierarchische Struktur der Anwendung – von der Wohnung über Zimmer bis zu einzelnen Pflanzen – entspricht einem mentalen Modell aus dem Alltagskontext. Dadurch wird die Orientierung erleichtert und eine intuitive Navigation gefördert.

Die übrigen Heuristiken wurden entweder implizit adressiert (z. B. minimalistisches Design durch Tailwind) oder aufgrund von Priorisierungen im Projektverlauf nicht explizit umgesetzt. Beispielsweise wurde auf Heuristik 9 verzichtet, da es sich nur um ein POC handelt und somit keine fortgeschrittenen User hat.

5.2.3 Responsive Design

Die Anwendung wurde im Hinblick auf verschiedene Gerätegrößen konzipiert. Dabei kamen insbesondere die Tailwind-Breakpoints `sm`, `md`, `lg` und `xl` zum Einsatz [25]. Für mobile Endgeräte wurde eine Bottom-Navigation eingeführt. Horizontales Scrollen auf der Startseite dient der kompakten Darstellung mehrerer Informationskarten.

5.2.4 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 Gruppenmitglied möchte ich einem Raum beitreten, 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.

5.3 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.

5.3.1 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   ]

```

Beispiel 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 [35], [36].

5.3.2 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 [37]:

- **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 worden, da aber keine starken Verzögerungen, bei der Bedienung, identifiziert werden konnten wurde noch keine Optimierung durchgeführt.

5.4 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 [38]. 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.

5.4.1 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 [39].

Der Kernmechanismus zur Lösung dieses Problems sind *Residual-Blöcke*. Statt rohe Ausgaben direkt weiterzureichen, lernen ResNet-Blöcke nur die Abweichung von der Identität[39].

Das ResNet50-Modell besteht aus insgesamt 50 Schichten, die sich aufteilen in:

- eine initiale Convolution-Schicht (7×7 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 [40]. 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 [41].

5.4.2 Regularisierung und Datenkonsolidierung

Zur Verbesserung der Modellrobustheit wurden mehrere datenaugmentierende Verfahren eingesetzt. Dazu zählen insbesondere Mixup [42] und CutMix [43], 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 [44].

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 [45].

5.4.3 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 5.5 auf der nächsten Seite sind Beispiel zu sehen.



Abbildung 5.1: Korrekt erkannt als *Anthurium Andraeanum* mit 99,72%



Abbildung 5.2: Korrekt erkannt als *Fragaria X Ananassa* mit 90,42%

Abbildung oben: Übersicht einiger markanter Testpflanzen.



Abbildung 5.3: Korrekt erkannt als *Lavandula Stoechas* mit 99,49%



Abbildung 5.4: Korrekt erkannt als *Lavandula Angustifolia* mit 98,05%

Abbildung unten: Vergleich der Genauigkeit zweier Lavendel Arten

Abbildung 5.5: Beispiel für die KI-Erkennung

5.4.4 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.

5.5 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 5.6 auf der nächsten Seite zu sehen.

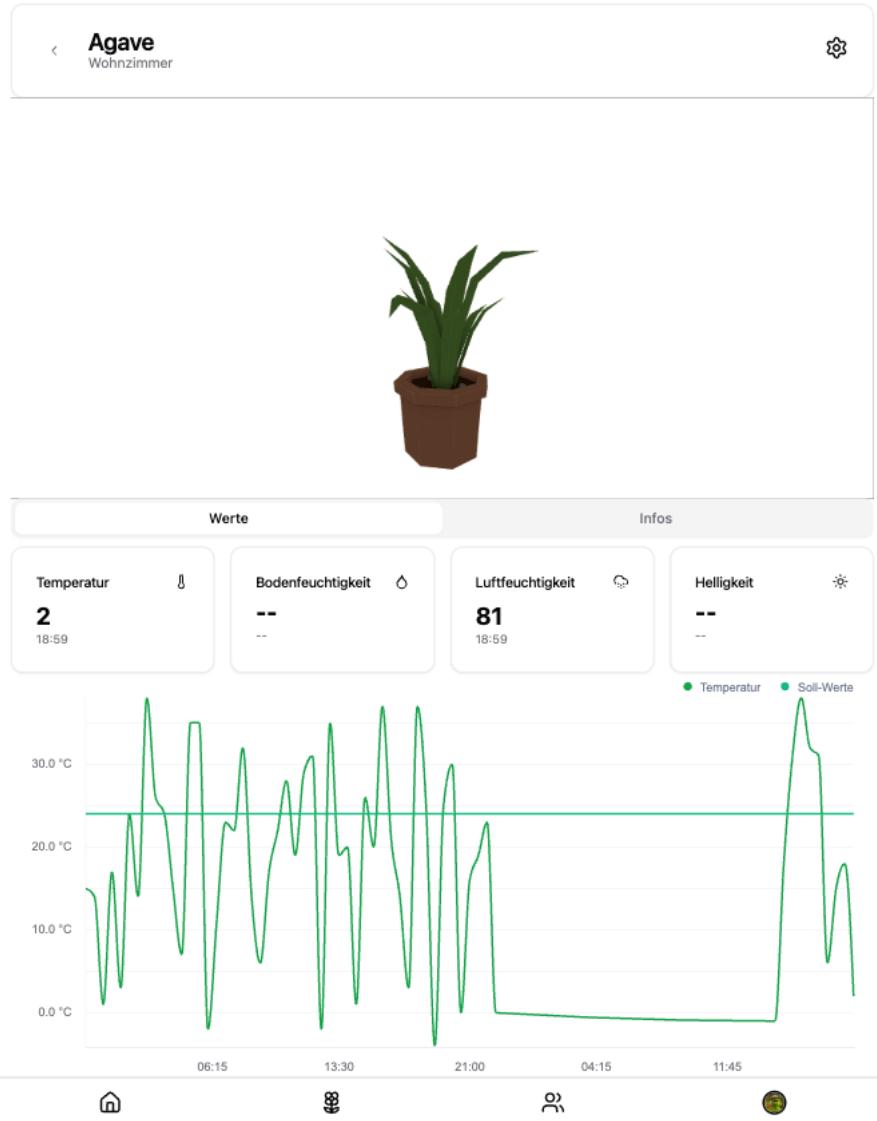


Abbildung 5.6: SinglePlantView in Aktion

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 [12].

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 [46].

Ausarbeitung/Umsetzung

6 Kritische Reflexion

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.

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 Zufunktionalen aus zeitlichen Gründen nicht realisiert, die in einer späteren Entwicklungsphase ergänzt werden können.

Darüber hinaus sind zwei kleinere Bugs bekannt, die zum aktuellen Stand noch nicht behoben wurden:

- Auf Geräten mit Android API Version 35 kann es bei aktiver Drei-Punkte-Navigationsleiste zu einer Überlappung mit der App-eigenen Navigationsleiste kom-

men.

- Nach der Erstellung einer neuen Gruppe werden die darin enthaltenen Räume in der Bearbeitungsansicht nicht sofort angezeigt, sofern kein Seitenwechsel oder manueller Refresh erfolgt.

Diese Einschränkungen haben jedoch keinen kritischen Einfluss auf die Hauptfunktionen und Nutzbarkeit der Anwendung.

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 Verbesserungspunkte bieten darüber hinaus eine wertvolle Grundlage für zukünftige Weiterentwicklungen.

7 Ausblick

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.

Literaturverzeichnis

Literatur

- [1] 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/>.
- [2] *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>.
- [3] Meta Open Source, *React Documentation*, URL: <https://reactjs.org/>, 2025.
- [4] Google, *Angular Documentation*, URL: <https://angular.io/docs>, 2025.
- [5] Vue.js Core Team, *Vue.js - The Progressive JavaScript Framework*, URL: <https://vuejs.org/>, 2016.
- [6] 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>.
- [7] 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).
- [8] 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>.
- [9] 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>.

- [10] 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>.
- [11] 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.
- [12] K. Werbach und D. Hunter, *For the Win: How Game Thinking can Revolutionize your Business*. Jan. 2012.
- [13] G. J. Myers, T. Badgett und C. Sandler, *The Art of Software Testing*, 3rd. Wiley, 2011, ISBN: 978-1118031964.
- [14] P. Ammann und J. Offutt, *Introduction to Software Testing*, 2. Aufl. Cambridge University Press, 2016.
- [15] J. Humble und D. Farley, *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Addison-Wesley, 2010, ISBN: 978-0321601919.
- [16] Vue.js Core Team, *Vue Test Utils Documentation*, <https://test-utils.vuejs.org/>, 2024. (besucht am 12.04.2025).
- [17] C. Team, *Cypress Documentation*, <https://docs.cypress.io/>, 2024. (besucht am 13.04.2025).
- [18] 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).
- [19] Vue.js Core Team, *Composition API: Introduction*, 2020. Adresse: <https://vuejs.org/guide/reusability/composables.html> (besucht am 14.04.2025).
- [20] 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/>.
- [21] V. C. Team, *Composition API FAQ*, 11. März 2025. Adresse: <https://vuejs.org/guide/extras/composition-api-faq.html>.
- [22] V. C. Team, *<script setup>*, 11. März 2025. Adresse: <https://vuejs.org/api/sfc-script-setup.html>.
- [23] 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/>.
- [24] *Vuex*, 12. März 2025. Adresse: <https://vuex.vuejs.org/>.

- [25] *Tailwind CSS – A utility-first CSS framework*, <https://tailwindcss.com>, Accessed 11 Apr 2025.
- [26] *shadcn-vue: Vue port of shadcn/ui*, <https://shadcn-vue.com>, Accessed 08 Apr 2025, 2023.
- [27] M. Singh und G. Shobha, „Comparative Analysis of Hybrid Mobile App Development Frameworks,“ Juli 2021. DOI: 10.35940/ijscse.F3518.0710621.
- [28] 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-1-131cf2e4410>.
- [29] I. Team, *Capacitor Documentation*, 2024. Adresse: <https://capacitorjs.com/docs>.
- [30] F. Stallmann und U. Wegner, *Internationalisierung von E-Commerce-Geschäften*. Springer Vieweg, 2015. DOI: 10.1007/978-3-658-06782-3.
- [31] Kazupon, *Vue I18n Documentation*, 2024. Adresse: <https://vue-i18n.intlify.dev/> (besucht am 14.04.2025).
- [32] 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).
- [33] T. B. David, „Why Flux,“ *Medium*, 28. März 2025. Adresse: <https://medium.com/@Tom1212121/why-flux-2d25ab8a8063>.
- [34] F. Inc., *Flux: Application Architecture for Building User Interfaces*, Zugegriffen: 13. April 2025, 2014. Adresse: <https://facebookarchive.github.io/flux/docs/in-depth-overview>.
- [35] V. Team, *Vite - Next Generation Frontend Tooling*, <https://vitejs.dev/>, 2024. (besucht am 13.04.2025).
- [36] R. Team, *Rollup Documentation*, <https://rollupjs.org/>, 2024. (besucht am 13.04.2025).
- [37] *Einführung in Lighthouse*. Adresse: <https://developer.chrome.com/docs/lighthouse/overview>.
- [38] 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>.

- [39] 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>.
- [40] 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>.
- [41] 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.
- [42] 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>.
- [43] 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>.
- [44] 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>.
- [45] 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>.
- [46] D. A. Norman, *The design of everyday things: Revised and expanded edition*. Basic books, 2013.

Anhang