

---

# Verbesserung der Screenreader-Unterstützung in Vuetify: Eine komponentenbasierte Analyse und Optimierung der Accessibility-Implementierung nach WCAG 2.1

Bachelorarbeit zur Erlangung des akademischen Grades  
*Bachelor of Science*  
im Studiengang Medieninformatik  
an der Fakultät für Informatik und Ingenieurwissenschaften  
der Technischen Hochschule Köln

vorgelegt von: Meike Jungilligens  
Matrikel-Nr.: 11153187  
Adresse: Schloßstraße 58  
40477 Düsseldorf  
[meike.jungilligens@smail.th-koeln.de](mailto:meike.jungilligens@smail.th-koeln.de)

eingereicht bei: Prof. Christian Noss  
Zweitgutachter: Prof. Dr. Hoai Viet Nguyen

Düsseldorf, 01.02.2026

## **Kurzfassung / *Abstract***

Digitale Barrierefreiheit ist eine zentrale Voraussetzung für die gleichberechtigte Teilhabe am Web. Mit dem Inkrafttreten des Barrierefreiheitsstärkungsgesetzes (BFSG) im Jahr 2025 sind entsprechende Anforderungen in Deutschland erstmals auch für große Teile des privaten Sektors verbindlich geworden. Screenreader spielen dabei eine wesentliche Rolle, da sie Menschen mit Sehbehinderungen den Zugang zu webbasierten Inhalten ermöglichen. Gleichzeitig werden in der Webentwicklung zunehmend komponentenbasierte UI-Frameworks eingesetzt, deren interne Implementierung maßgeblich über die tatsächliche Barrierefreiheit der resultierenden Benutzeroberflächen entscheidet.

Diese Arbeit analysiert die Screenreader-Kompatibilität ausgewählter Komponenten des Vue-basierten UI-Frameworks Vuetify (Version 3) und untersucht, durch welche Maßnahmen bestehende Barrieren reduziert werden können. Der Fokus liegt auf der Unterstützung von Screenreadern im Sinne der WCAG 2.1. Als Untersuchungsgegenstand wurden vier Komponenten ausgewählt, die anhand offener Accessibility-Issues im Vuetify-Repository identifiziert wurden.

Methodisch wird aufbauend auf einer Analyse bestehender Accessibility-Patterns im Vuetify-Quellcode ein Entscheidungsbaum entwickelt, der als strukturierte Strategie für die Auswahl zwischen nativer HTML-Semantik und ARIA-basierten Implementierungen dient. Die identifizierten Defizite werden gezielt im Quellcode behoben und anhand identischer Testverfahren erneut überprüft.

Die Ergebnisse zeigen, dass sich die Screenreader-Kompatibilität der betrachteten Komponenten durch gezielte Anpassungen deutlich verbessern lässt. Alle zuvor nicht erfüllten Screenreader-relevanten WCAG-Kriterien konnten nach der Optimierung erfüllt werden. Die entwickelten Änderungen wurden in Form von Pull Requests in den Open-Source-Entwicklungsprozess von Vuetify eingebracht und verdeutlichen, dass Barrierefreiheit in komponentenbasierten Frameworks eine systematische Analyse und bewusste Implementierungsentscheidungen erfordert.

Schlüsselwörter: Digitale Barrierefreiheit, Screenreader, WCAG 2.1, Vuetify, ARIA, komponentenbasierte Webentwicklung

# Inhaltsverzeichnis

<b>Tabellenverzeichnis</b>	<b>III</b>
<b>Abbildungsverzeichnis</b>	<b>IV</b>
<b>Glossary</b>	<b>V</b>
<b>Abkürzungsverzeichnis</b>	<b>VI</b>
<b>1. Einleitung</b>	<b>1</b>
<b>2. Theoretischer Hintergrund</b>	<b>3</b>
2.1. Einführung in digitale Barrierefreiheit . . . . .	3
2.2. Rolle von Screenreadern in der Barrierefreiheit . . . . .	4
2.3. Einführung in Vuetify als Untersuchungsgegenstand . . . . .	6
<b>3. Methodik</b>	<b>8</b>
3.1. Komponentenauswahl . . . . .	8
3.2. Komponentenanalyse . . . . .	10
3.3. Implementierung und Verifikation . . . . .	12
3.3.1. Analyse bestehender Accessibility-Patterns in Vuetify . . . . .	13
3.3.2. Entwicklung einer Implementierungsstrategie . . . . .	16
3.3.3. Praktische Umsetzung der Optimierungen . . . . .	19
3.3.4. Analyse und Verifikation der Optimierungen . . . . .	20
3.3.5. Einbindung in den Open-Source-Entwicklungsprozess . . . . .	21
<b>4. Diskussion</b>	<b>22</b>
<b>5. Fazit und Ausblick</b>	<b>24</b>
<b>Literaturverzeichnis</b>	<b>26</b>
<b>A. Anhang</b>	<b>28</b>
A.1. Übersicht der a11y-Issues aus dem Vuetify-Repository . . . . .	28
A.2. Testdokumentation vor der Optimierung . . . . .	28
A.3. Testdokumentation nach der Optimierung . . . . .	30

## **Tabellenverzeichnis**

3.1. Klassifizierung der offenen a11y-Issues (Stand 08.12.2025) in Vuetify . . . . .	9
3.2. Übersicht der ausgewählten a11y-Issues im Vuetify-Repository . . . . .	11
3.3. Klassifikation der in Vuetify beobachteten Implementierungsansätze für Screenreader-Kompatibilität . . . . .	14

# Abbildungsverzeichnis

3.1. Dokumentation der Testergebnisse für die Komponente <b>VDataTable</b> . . . . .	12
3.2. Entscheidungsbaum für die Implementierungsstrategie . . . . .	18
3.3. Vergleich der Testergebnisse für die Komponente <b>VDataTable</b> vor und nach der Optimierung . . . . .	21
A.1. Dokumentation der Testergebnisse für die Komponente <b>VAutocomplete</b> . . . . .	28
A.2. Dokumentation der Testergebnisse für die Komponente <b>VDataTable</b> . . . . .	29
A.3. Dokumentation der Testergebnisse für die Komponente <b>VTimePicker</b> . . . . .	29
A.4. Dokumentation der Testergebnisse für die Komponente <b>VTretreeview</b> . . . . .	30
A.5. Dokumentation der Testergebnisse für die Komponente <b>VAutocomplete</b> nach der Optimierung . . . . .	30
A.6. Dokumentation der Testergebnisse für die Komponente <b>VDataTable</b> nach der Optimierung . . . . .	31
A.7. Dokumentation der Testergebnisse für die Komponente <b>VTimePicker</b> nach der Optimierung . . . . .	31
A.8. Dokumentation der Testergebnisse für die Komponente <b>VTretreeview</b> nach der Optimierung . . . . .	32

# Glossar

**a11y** Abkürzung für „Accessibility“, bezeichnet die barrierefreie Gestaltung von Software und Webanwendungen (MDN Web Docs, 2025a). 8

**Braillezeile** Taktiles Ausgabegerät, das Textinformationen eines Computers in Brailleschrift darstellt (Help Tech, 2026). 4

**Emits** Mechanismus in komponentenbasierten Frameworks wie Vue, mit dem eine Komponente Ereignisse an ihre Elternkomponenten auslöst. 7

**ESM** Standardisiertes Modulsystem von JavaScript zur Strukturierung von Importen und Exporten. 7

**Kompatibilität** (hier bezogen auf Screenreader) Eigenschaft einer Benutzeroberfläche oder einer Komponente, von Screenreadern korrekt interpretiert und ausgegeben zu werden. 1, 2, 4, 8, 10, 13, 15, 19, 20, 22

**SASS** CSS-Präprozessor, der zusätzliche Sprachfunktionen wie Variablen und Verschachtelung bereitstellt. 7

**Screenreader** Assistive Software, die Bildschirminhalte in gesprochene Sprache oder Brailleschrift umwandelt (MDN Web Docs, 2025a). 1, 2, 4, 5, 8, 10, 12, 13, 15, 19, 20, 22

**Slot** Platzhalter in komponentenbasierten UI-Frameworks, über den Inhalte von außen in eine Komponente eingefügt werden können. 7

**TSX** TypeScript-Datei, die JSX-Syntax enthält und häufig zur Implementierung von React- oder Vue-Komponenten verwendet wird. 7

## Abkürzungsverzeichnis

**APG** ARIA Authoring Practices Guide. 6, 16, 17, 19, 20

**ARIA** Accessible Rich Internet Applications. 4–6, 12, 13, 15, 16, 19, 22, 23

**BFSG** Barrierefreiheitsstärkungsgesetz. 1, 3

**BGG** Behingertengleichstellungsgesetz. 3

**BITV** Barrierefreie Informationstechnik-Verordnung. 3

**HTML** Hypertext Markup Language. 4–6, 13, 15, 16, 22, 23

**W3C** World Wide Web Consortium. 3, 4

**WAI** Web Accessibility Initiative. 3–6, 16, 19

**WCAG** Web Content Accessibility Guidelines. 3, 10, 12, 22

# 1. Einleitung

Das Web ist mittlerweile ein zentraler Bestandteil gesellschaftlicher Teilhabe. Somit ist es wichtig, dieses Umfeld für jegliche Personengruppen inklusiv und hürdenlos zu gestalten. In diesem Kontext hat das Thema Barrierefreiheit in den letzten Jahren deutlich an Bedeutung gewonnen und ist seit dem letzten Jahr auch für den privaten Sektor gesetzlich verankert: Am 28. Juni 2025 ist das Barrierefreiheitsstärkungsgesetz (BFSG) in Kraft getreten, welches im deutschen Rechtsraum verbindliche Anforderungen an das Maß an Barrierefreiheit stellt, welches Unternehmen bei ihren digitalen Auftritten aufweisen müssen (BFSG, 2021). Diese Verpflichtung sorgt für viele Fragen und Hilfesuchung der Unternehmen, und das Feld der Barrierefreiheitsmessung von Webseiten oder Unterstützung wie Auditing durch Programme oder Dienstleister hat entsprechend zugenommen.

Digitale Barrierefreiheit bedeutet, dass auch Menschen mit körperlichen oder geistigen Beeinträchtigungen eine hürdenlose Erfahrung bei der Bedienung von und Interaktion mit Webseiten haben. Eine zentrale Rolle spielen dabei unter anderem sogenannte Screenreader, welche die visuellen und textuellen Inhalte einer Webseite auf eine alternative Weise darstellen. Die Ausgabe der Inhalte erfolgt auditiv oder sensorisch über Braille-Texte (American Foundation for the Blind, 2025). Es wird angenommen, dass allein in Deutschland geschätzt rund 1,2 Millionen Menschen mit einer Form von Blindheit oder Sehbehinderung leben (REHADAT-Statistik, 2025), denen das Miterleben von sonst rein visuellen Internetinhalten durch solche Schnittstellen ermöglicht werden kann.

Woran es jedoch mangelt sind Untersuchungen der Zusammenwirkung moderner Webentwicklungsmethoden und der Barrierefreiheit. In der Webentwicklung werden vermehrt Frameworks, vor allem auf JavaScript-Basis, genutzt, die den Entwicklungsprozess vereinfachen, in dem sie unter anderem Grundstrukturen und -komponenten bereitstellen. Eines dieser Frameworks ist Vuetify: Auf Basis von Vue.js und dem Material Design von Google (Vuetify, 2024b) werden Entwicklern fertige funktionale Komponenten zur direkten Nutzung bereitgestellt. Eine Analyse in dieser Arbeit vorangegangenen Praxisprojekt<sup>1</sup> zeigte jedoch, dass diese Komponenten nicht von Grund auf barrierefrei sind. Teilweise müssen durch den Entwickler eigene Anpassungen u.a. bezüglich Screenreader-Kompatibilität oder Kontrasten durchgeführt werden, damit

---

<sup>1</sup><https://github.com/mjung2605/pp2025-barrierefreiheit-vuetify>

diese die gesetzlichen Anforderungen erfüllen. Dies steht im Kontrast zu dem grundlegenden Anspruch moderner UI-Frameworks, Entwicklungsprozesse zu vereinfachen und zu beschleunigen, indem wiederverwendbare, konsistente Komponenten bereitgestellt werden. Obwohl Vuetify den Fokus auf die Umsetzung des Material Designs legt, entsteht bei Entwicklern häufig die implizite Erwartung, dass diese vorgefertigten Komponenten eine solide Basis für die Einhaltung gängiger Qualitätsanforderungen bieten.

Vor diesem Hintergrund und aufbauend auf dem Praxisprojekt steht im Fokus der vorliegenden Arbeit die Optimierung der Screenreader-Kompatibilität ausgewählter Vuetify-Komponenten. Die Arbeit soll einen praktischen Beitrag zu Vuetify als Open-Source-Projekt in Form von Pull Requests leisten, und somit dazu beitragen, dass Entwickler künftig von barriereärmeren Komponenten Gebrauch machen können. „Optimierung“ bezeichnet hier nicht die vollständige Neuentwicklung der betrachteten Komponenten, sondern gezielte Anpassungen bestehender Implementierungen mit dem Ziel, identifizierte Barrieren für Screenreader-Nutzende zu reduzieren. Eine erfolgreiche Optimierung wird anhand der Testmethodik aus dem Praxisprojekt gemessen, die sowohl vor als auch nach der Implementierung angewendet wird. Dadurch wird sichergestellt, dass Verbesserungen der Screenreader-Kompatibilität nachvollziehbar nachgewiesen werden können. Somit verfolgt die Arbeit ein theoretisches sowie darauf aufbauend ein praktisches Ziel. Die zugrundeliegende Forschungsfrage lautet wie folgt:

„Wie barrierefrei sind ausgewählte Vuetify-Komponenten in Bezug auf ihre Screenreader-Kompatibilität und durch welche Maßnahmen kann diese verbessert werden?“

Das Thema der Arbeit und der entsprechende Forschungs- und Praxiskontext, in den es eingebettet ist, wird zunächst in dem theoretischen Teil (Kapitel 2) der Arbeit erläutert. Dieser stellt grundlegende Begriffe und Konzepte aus den Themenfeldern (digitale) Barrierefreiheit, Screenreader sowie die Bedeutung von ARIA-Rollen oder semantischen HTML-Elementen in Bezug auf Screenreader-Kompatibilität dar und stellt die grundlegende Code-Struktur der Vuetify-Komponenten vor. Der Theorieteil dient als Einführung und Grundlage für den darauffolgenden praktischen Teil. Der praktische Methodikteil (Kapitel 3) gliedert sich gemäß der Forschungsfragen in mehrere Hauptteile auf: Der Konzipierung der Implementation und der praktischen Umsetzung im Quellcode der Vuetify-Komponenten. Es werden die methodische Durchführung und die Ergebnisse aufgezeigt. Die anschließende Diskussion dient der wissenschaftlichen Einordnung und Bewertung der Ergebnisse. Den Abschluss der Arbeit bildet das Beantworten der Forschungsfragen sowie ein Ausblick auf weitere Forschungsmöglichkeiten in der Domäne.

Düsseldorf, Januar 2025

## **2. Theoretischer Hintergrund**

### **2.1. Einführung in digitale Barrierefreiheit**

Die rechtliche Verankerung der digitalen Barrierefreiheit besteht nicht erst seit dem BFSG. Schon Anfang der 2000er Jahre begann das Behingertengleichstellungsgesetz (BGG), den öffentlichen Sektor zu verpflichten, ihre Produkte barrierefrei anzubieten (BGG, 2002). Konkrete Anforderungen entsprungen der dabei auf dem BGG basierenden Barrierefreie Informationstechnik-Verordnung (BITV) (BITV 2.0, 2021), die sich an den Web Content Accessibility Guidelines (WCAG) orientierten (W3C, 2025d). Mit den Aktualisierungen der WCAG ging auch eine Überarbeitung der Gesetze und Verordnungen einher. Vor 20 Jahren hat das Thema Barrierefreiheit im digitalen Raum auch seinen Weg in die EU-Richtlinien gefunden, wo sich ebenfalls auf die WCAG als internationaler Standard berufen wurde (EU 2016/2102, 2016; EU 2019/882, 2019). Mittlerweile beziehen sich EU- sowie nationale Gesetzesgebungen auf die EU-Norm 301 549, die auf die WCAG 2.1 aus dem Jahr 2018 verweist („Accessibility requirements for ICT products and services“, 2021). Obwohl schon neue WCAG-Versionen existieren, sind diese zum gegebenen Zeitpunkt nicht rechtlich relevant.

Die WCAG entstammen dem 1994 gegründeten World Wide Web Consortium (W3C), dessen Web Accessibility Initiative (WAI) verschiedene Standards, Empfehlungen und Best Practices im Umfeld der Barrierefreiheit entwickelt (W3C, 2025a). In den Richtlinien werden Kriterien aufgestellt, die in die vier Prinzipien der Barrierefreiheit gegliedert sind: Wahrnehmbar, Bedienbar, Verständlich und Robust (auch bekannt unter dem englischen Akronym POUR). Die einzelnen Kriterien entsprechen jeweils einer aus drei Konformitätsstufen (A, AA, AAA). Alle Kriterien einer Stufe müssen bestanden sein, damit ein Produkt als konform auf dieser Stufe gilt. Die Stufen finden sich auch in den Gesetzesgebungen wieder: Webseiten privater Unternehmen müssen nach dem BFSG die Konformitätsstufe AA erreichen, Webseiten des öffentlichen Sektors nach der BITV die Stufe AAA. Die Stufen sind gleichzusetzen mit dem Ausmaß an erreichter Barrierefreiheit, weswegen nach einer möglichst hohen Stufe gestrebt werden sollte.

## 2.2. Rolle von Screenreadern in der Barrierefreiheit

Menschen mit unterschiedlichen Einschränkungen stehen unterschiedliche assistive Technologien zur Verfügung, die ihre Teilhabe am digitalen kulturellen Austausch erleichtern. Abhängig von der jeweiligen Beeinträchtigung kommen dabei unter anderem Sprachausgaben, Braillezeilen oder alternative Ausgabemethoden zum Einsatz. Für Personen mit visuellen sowie audiovisuellen Behinderungen spielen Screenreader beim Navigieren der digitalen Welt eine zentrale Rolle. Diese analysieren die vom Betriebssystem oder der Anwendung bereitgestellte Benutzeroberfläche und geben deren Inhalte in auditiver Form (Sprachausgabe) oder über eine Braillezeile (taktile Ausgabe) wieder. Screenreader interpretieren zusätzlich Eingaben des Nutzers (z. B. Tastatureingaben zur Navigation durch Bedienelemente), sie sind allerdings selbst keine Eingabegeräte, sondern Vermittler zwischen Eingabehardware und Systemausgabe.

Screenreader geben dabei nicht nur die Textanteile der grafischen Nutzeroberfläche aus, sondern auch Informationen über nicht-textuelle Inhalte wie Informationen zu Bildern oder die Rollen von Interaktionselementen. Um die optimale Funktion des Screenreaders zu gewährleisten, muss die jeweilige Webseite diese Informationen programmatisch bereitstellen. Dazu gehören nicht nur Alternativtexte für Bilder, sondern auch eine klare Strukturierung des Markups, damit Elemente wie Listen, Links oder Formularelemente durch die assistive Technologie richtig erkannt, interpretiert und ausgegeben werden können.

Eine erste (und immer noch bewährte) Methode, die Rollen und Werte programmatisch lesbar zu machen, entstammt einer Arbeitsgruppe der WAI: Die Accessible Rich Internet Applications (ARIA)-Rollen aus dem Jahr 2008 stellen eine semantische Erweiterung der Markup-Sprache Hypertext Markup Language (HTML) dar, die es Entwicklern ermöglicht, Identitäten, Rollen und Beziehungen der Elemente im Markup zu deklarieren und diese so für einen Screenreader erkennbar zu machen (W3C, 2025c). Einen weiteren bedeutenden Entwicklungsschritt markierte im Jahr 2014 die Veröffentlichung der HTML5-Spezifikation durch das W3C (W3C, 2014). Zentral für Screenreader-Nutzende war dabei das Einführen von semantischen HTML, welches die Praxis beschreibt, HTML-Elemente gemäß ihrer inhaltlichen und strukturellen Bedeutung einzusetzen, anstatt lediglich Container zu verwenden. Diese Strukturen ermöglichen assistiven Technologien den Zugang zu den logischen Beziehungen und Rollen einzelner Seiterelemente. ARIA ergänzt semantisches HTML heute gezielt dort, wo Standard-Elemente keine ausreichende Bedeutungsbeschreibung liefern.

ARIA und semantisches HTML schließen sich als Methoden zur Verbesserung der Screenreader-Kompatibilität nicht gegenseitig aus, vielmehr ergänzen sie sich. Während semantisches HTML grundlegende Bedeutungen und Rollen über native Elemente

bereitstellt, wurde ARIA entwickelt, um diese Semantik gezielt zu erweitern, insbesondere in Fällen, in denen die Markup-Sprache selbst keine geeigneten Ausdrucksmittel bietet. Dies ist beispielsweise dann der Fall, wenn interaktive Benutzeroberflächen oder komplexe Widgets umgesetzt werden, für die keine nativen HTML-Elemente existieren, etwa Baumstrukturen (*tree views*) oder komplexe Menüs.

Das WAI-ARIA-Dokument definiert hierzu drei zentrale Kategorien: Rollen (*roles*), Eigenschaften (*properties*) und Zustände (*states*). Rollen beschreiben die grundlegende Funktion eines Elements innerhalb der Benutzeroberfläche und sind in der Regel statisch, da sie die semantische Identität eines Elements festlegen (z. B. `button`, `navigation` oder `treeitem`). Die Attribute (*properties* und *states*) hingegen sind dynamisch und dienen dazu, aktuelle Eigenschaften oder Zustände eines Elements zu kommunizieren. Während Properties eher dauerhafte Merkmale beschreiben (z.B. `aria-multiselectable`), repräsentieren States veränderliche Zustände wie `aria-expanded` oder `aria-disabled`. Diese Informationen ermöglichen es assistiven Technologien, Änderungen der Benutzeroberfläche korrekt zu erfassen und an Nutzende weiterzugeben.

ARIA ist dabei ausdrücklich als Ergänzung und nicht als Ersatz für native HTML-Semantik konzipiert. Die WAI empfiehlt, stets zunächst native HTML-Elemente zu verwenden, sofern diese eine vergleichbare Bedeutung und Funktionalität bereitzustellen (W3C, 2026b). Native Elemente bieten den Vorteil, dass Browser und assistive Technologien bereits standardisierte Tastaturnavigation und Zustandsverwaltung implementieren. ARIA sollte daher nur dort eingesetzt werden, wo die verwendete Auszeichnungssprache keine ausreichenden semantischen Mittel bereitstellt oder wo bestehende Inhalte nachträglich barriereärmer gestaltet werden müssen.

Ein zentraler Grundsatz im Umgang mit ARIA lautet dabei: „A role is a promise.“ (W3C, 2026b). Das Setzen einer ARIA-Rolle verpflichtet Entwickler dazu, sämtliche mit dieser Rolle verbundenen erwarteten Interaktionsmöglichkeiten selbst zu implementieren. Im Gegensatz zu nativen HTML-Elementen erzeugen ARIA-Rollen weder automatische Tastaturbedienbarkeit noch standardisierte Verhaltensweisen. Ein unvollständiger oder inkorrekteter Einsatz von ARIA kann somit zu einer fehlerhaften Repräsentation der Benutzeroberfläche führen und die Nutzung für Screenreader-Nutzende erheblich erschweren.

Vor diesem Hintergrund gilt der in den Authoring Practices formulierte Leitsatz: „No ARIA is better than bad ARIA.“ (W3C, 2026b). Semantisches HTML stellt in vielen Fällen die robustere Grundlage für barrierefreie Webanwendungen dar. Dennoch bleibt ARIA eine bewährte Methode, um neuartige oder komplexe Widgets zugänglich zu machen, die den Funktionsumfang nativer HTML-Elemente übersteigen. Die Herausforderung besteht somit nicht im reinen Einsatz von ARIA, sondern in dessen

verantwortungsvoller Anwendung. Um Entwickler bei der korrekten Anwendung zu unterstützen, existiert mit dem ARIA Authoring Practices Guide (APG) der WAI (W3C, 2026a) etablierte ARIA-Patterns, die für komplexe Widgets nicht nur empfohlene Rollen und Attribute, sondern auch die zu implementierenden Tastaturinteraktionen darstellen.

### 2.3. Einführung in Vuetify als Untersuchungsgegenstand

Während semantisches HTML und ARIA die technischen Grundlagen für barrierefreie Webinhalte schaffen, stellt die praktische Umsetzung in modernen Webprojekten häufig eine zusätzliche Herausforderung dar. Insbesondere der Einsatz von Komponentenframeworks wie Vuetify, Bootstrap oder Material UI beeinflusst die Barrierefreiheit maßgeblich, da sie die zugrunde liegende HTML-Struktur und das Verhalten vieler Interface-Elemente abstrahieren.

Komponentenbasierte Frameworks bieten vorgefertigte UI-Bausteine, die auf modernen Frontend-Technologien (meist JavaScript-basiert) beruhen. Sie erleichtern die Entwicklung konsistenter Benutzeroberflächen, übernehmen jedoch zugleich Verantwortung für deren Korrektheit. Eine barriearame Implementierung hängt daher stark davon ab, in welchem Maß die Komponenten standardkonformes HTML, korrekte ARIA-Attribute und Tastaturnavigation unterstützen.

Das Framework Vuetify ist ein auf Vue.js basierendes Open-Source-Projekt, das 2016 entstanden ist und sich am Material-Design-System von Google orientiert (Vuetify, 2024b). Es stellt eine umfassende Sammlung an UI-Komponenten bereit und verfolgt den Anspruch, responsive und performante Anwendungen zu ermöglichen. Vuetify veröffentlicht mehrmals pro Monat kleinere Releases, die meist der Fehlerbehebung dienen. Am 30.12.2025 wurde die Alpha-Version von Vuetify 4 veröffentlicht (Vuetify, 2026), wobei der initiale Release von Vuetify 3 zu dem Zeitpunkt drei Jahre und zwei Monate zurücklag. Die in dieser Arbeit betrachtete und verwendete Version ist Vuetify 3.11.6.

Technisch ist Vuetify als Monorepo organisiert, das heißt, alle Teilkäpfe (z. B. Komponentenbibliothek, Dokumentation, Themes) werden innerhalb eines gemeinsamen Repositorys verwaltet. Dieses Vorgehen erleichtert die konsistente Versionierung und gemeinsame Weiterentwicklung der Module. Vuetify ist mit ca. 40900 Github-Stars das größte Vue-basierte Komponentenframework und verfügt neben dem Core-Team über eine aktive Community, die durch das Erstellen von Issues oder Pull Requests die Weiterentwicklung unterstützt. Beiträge (Contributions) erfolgen über Pull-Requests, die einer bestimmten Struktur unterliegen, und im besten Fall mit einem vorhandenen Issue verknüpft werden (Vuetify, 2025b).

Trotz dieses Engagements variiert der Grad der umgesetzten Barrierefreiheit einzelner Komponenten. Im dieser Arbeit vorausgegangenen Praxisprojekt traten bereits in grundlegenden Komponenten wie Buttons oder Textfeldern Defizite bezüglich ihrer Barrierefreiheit auf. Daher ist eine kritische Überprüfung der Accessibility-Implementierung in komponentenbasierten Frameworks wie Vuetify notwendig, insbesondere wenn sie in öffentlichen oder rechtlich barrierefreien Anwendungen eingesetzt werden sollen.

Da es in dieser Arbeit um den internen Aufbau der Vuetify-Komponenten geht, wird dieser im Folgenden genauer betrachtet. Der Fokus bei dieser Auseinandersetzung liegt dabei auf den Komponenten. Andere Teile des Quellcodes, wie globales Styling oder Direktiven sind für die vorliegende Arbeit aufgrund der Screenreader-Fokussierung nicht relevant.

Jede der 76 Vuetify-Komponenten befinden sich in einem eigenen Ordner, der die Komponentenlogik (als TSX-Datei) und -Styles (als SASS-Datei) bündelt. Gegebenenfalls befinden sich zusätzliche Tests oder Hilfslogik in weiteren Unterordnern. Eine einzelne TSX-Datei, die eine Komponente oder einen Teil davon beschreibt, lässt sich (nach ESM Import/Export Syntax) in feste Imports und Exports unterteilen. Importiert werden kann je nach Bedarf Folgendes:

- Styles oder Directives, die eine rein visuelle Funktion haben,
- Teil- oder Hilfskomponenten, die im Rendering der Komponente Verwendung finden,
- Composables und Utilities, die Hilfsfunktionen wie Konvertierer oder PropFactories bereitstellen,
- Types, die der sicheren Typisierung von z.B. den Slots der Komponente dienen.

Jede Vuetify-Komponente stellt eine klar definierte öffentliche Schnittstelle bereit, die sich im Wesentlichen aus Properties (Props), optionalen Slots sowie der eigentlichen Komponenteninstanz zusammensetzt. Die Props werden in der Regel über eine zentrale `propsFactory()` definiert, wodurch konsistente Typisierung und Wiederverwendbarkeit ermöglicht werden. Die Komponente selbst wird anschließend mit Metadaten wie Name, Props und optionalen Emits-Definitionen versehen. Die interne Logik ist in der `setup`-Funktion gekapselt, in der reaktive Zustände und Composables initialisiert werden. Das Rendering erfolgt über `useRender()`, wobei native HTML-Elemente oder weitere Vuetify-Komponenten verwendet werden.

Diese Struktur wird in Kapitel 3.2 erneut aufgegriffen und gilt als Basis für dieses, da dort die vorhandene Barrierefreiheitsimplementierungsstrategie im Quellcode der Vuetify-Komponenten analysiert wird, um daraus nach Möglichkeit eine Implementierungsstrategie abzuleiten.

## **3. Methodik**

Die Durchführung der Arbeit gliedert sich in drei Phasen: Komponentenauswahl, Analyse und Implementierung. Die Komponentenauswahl (Kapitel 3.1) dient der Fokussetzung auf eine feste Anzahl an Komponenten, die in der Arbeit in den daraufliegenden Kapiteln analysiert und optimiert werden sollen. Die Analyse (Kapitel 3.2) orientiert sich dabei an dem im Praxisprojekt entwickelten Testschema. Die Implementierung (Kapitel 3.3) stellt den größten Teil des praktischen Teils der Arbeit dar und beschäftigt sich mit dem Herausarbeiten einer Implementierungsstrategie mit anschließender Umsetzung im Vuetify-Quellcode.

### **3.1. Komponentenauswahl**

Um die Untersuchungsgegenstände festzulegen, musste vorab eine Auswahl der zu betrachteten Komponenten getroffen werden. Grundlage hierfür waren Nutzdaten aus dem Vuetify-Repository: Es gibt 33 offene Issues mit dem Label „a11y“ (Stand 08.12.2025), die in einem ersten Schritt genauer betrachtet und ausgewertet wurden. Ziel ist es hierbei, Komponenten mit bestehenden Problemen bei der Screenreader-Kompatibilität zu erfassen. Das Verwenden der Issues aus dem offiziellen Repo als Datengrundlage ist hierbei dadurch motiviert, dass aktuell auftretende Issues aufgezeigt werden, die in den meisten Fällen mit einem Reproduktionslink ausgestattet sind, worauf im Kapitel 3.2 gut aufgebaut werden kann.

Jedes Issue bezieht sich auf eine konkrete fehlerhafte Komponente. Zielbild dieses ersten Schrittes ist eine reduzierte Auswahl von Komponenten, bei denen Barrieren in der Screenreader-Kompatibilität bestehen, dessen Optimierung das Werk dieser Arbeit darstellt. Um dies zu erreichen werden die vorliegenden Issues unter bestimmten Gesichtspunkten vorsortiert. Die in Tabelle 3.1 aufgeführten Ausschlusskriterien ergeben sich aus technischen Einschränkungen seitens der Autorin (Kriterium 3), dem Anwendungsbereich des Projekts (Kriterium 1 und 4) oder dem (Nicht-)Vorhandensein eines nachvollziehbaren Problems (Kriterium 2 und 5).

Die in Kriterium 1 erwähnte „Screenreader-Relevanz“ bezieht sich hier auf zwei Teilfunktionalitäten: Sowohl die Funktion der Screenreader an sich, also korrektes Vorlesen von Labels, Namen und Bezügen, als auch die Tastaturnavigation. Im Nutzungskontext

Nr.	Kennzeichnung in Tabelle	Bedeutung	Begründung
1	<code>not sr issue</code>	Das a11y-Problem bezieht sich auf ein WCAG-Kriterium, das nicht Screenreader-relevant ist (z. B. Kontrast).	Die vorliegende Arbeit begrenzt sich auf die Screenreadernutzung.
2	<code>not reproducible</code>	Das Issue kann in dem verlinkten „Playground“ nicht reproduziert oder nachvollzogen werden.	Das Reproduzieren ist Grundlage für das erneute Testen und Beheben des Problems.
3	<code>wrong os</code>	Das Issue tritt in einem anderen Betriebssystem oder auf einem anderen Gerät auf (z. B. Screenreader für macOS) und kann auf dem vorhandenen System (Windows-PC) nicht reproduziert werden.	Das Reproduzieren ist Grundlage für das erneute Testen und Beheben des Problems.
4	<code>issue on sr side</code>	Das Issue scheint durch den Screenreader selbst verursacht zu sein und nicht durch die Vuetify-Komponente.	Die vorliegende Arbeit beschränkt sich auf das Beheben von Vuetify-internen Problemen.
5	<code>fix pr already linked</code>	Eine lösende Pull Request ist bereits im Issue verlinkt.	Das Problem wird bereits bearbeitet oder ist bereits gelöst.

Tabelle 3.1.: Klassifizierung der offenen a11y-Issues (Stand 08.12.2025) in Vuetify

von Screenreadern fungiert die Tastatur (oder alternative Eingabemodalitäten) als Input und der Screenreader selbst als Output; für die Screenreader-Kompatibilität ist also beides relevant. Issues, die beschreiben, dass eins oder beide dieser Funktionalitäten nicht implementiert sind, gelten hier also als relevant für die Arbeit und werden durch Kriterium 1 nicht ausgeschlossen. Die vollständige Tabelle der Issues mit der Einordnung, ob sie in der weiteren Arbeit reflektiert werden oder nicht (nach Tabelle 3.1) befindet sich aufgrund ihrer Länge im Anhang (s. A.1).

Reduziert wird die Liste der Issues durch die Ausschlusskriterien auf ungelöste, reproduzierbare Probleme der Komponenten bezüglich ihrer Screenreader-Kompatibilität. Dies gewährleistet, dass die in Kapitel 1 gestellten Forschungsfragen bearbeitet und beantwortet werden können, und die Implementation der Optimierung sowie auch besonders das Testen technisch durchführbar sind. Die Ausschlusskriterien reduzieren die Liste der Issues von 33 auf sechs Issues (aufgeführt in Tabelle 3.2), wobei die Komponenten **VTreeview** und **VDataTable** mehrfach genannt wurden. Es sind in der Arbeit also vier Komponenten zu betrachten: **VTreeview**, **VDataTable**, **VAutocomplete** und **VTimePicker**.

## 3.2. Komponentenanalyse

Die Komponenten wurden im nächsten Schritt erneut getestet. Das dient einerseits dazu, sicherzustellen, dass die Fehler auch in der aktuellen Version von Vuetify noch bestehen und gelöst werden müssen. Außerdem erlaubt ein Testen nach dem Schema des Praxisprojekts eine Formalisierung der Issues und eine Herstellung des Bezugs auf die WCAG 2.1. Somit wird auch eine Vergleichsgrundlage für das Evaluieren der optimierten Komponenten am Ende der Arbeit gelegt.

Der Prozess der Auswahl der zu prüfenden Kriterien pro Komponente ist komplexer als im Praxisprojekt. Das ist mit der wachsenden Komplexität der Komponenten und der damit einhergehenden zunehmenden Anzahl prüfbarer Kriterien zu begründen, da die Komponenten im Gegensatz zum Praxisprojekt nicht atomar sind. Die Fokussierung auf Screenreader-relevante Kriterien reduziert die Kriterienauswahl gegenüber dem Praxisprojekt, in dem die Barrierefreiheit der Komponenten in vollem Umfang betrachtet wurde, wiederum wieder. Wie in Kapitel 3.1 erwähnt, setzt sich Screenreader-Kompatibilität aus zwei Aspekten zusammen: Die Ausgabe des Screenreaders (technisch: die Bereitstellung entsprechender Informationen durch Auszeichnung mit bspw. ARIA) sowie die Möglichkeit zur Interaktion mit Tastatur. Bei der Auswahl der zu prüfenden Kriterien wurde sich also wie im Praxisprojekt auf die WCAG 2.1 bezogen, und dabei ein Fokus auf Kriterien bezüglich Erkennbarkeit durch Software (hier: Screenreader) sowie Tastaturbedienbarkeit gelegt.

Issue ID	Titel	URL
21889	[Bug Report][3.9.4] Accessibility issue with v-tree-view	<a href="https://github.com/vuetifyjs/vuetify/issues/21889">https://github.com/vuetifyjs/vuetify/issues/21889</a>
21885	[Feature Request] Add aria-disabled to disabled hours/minutes in time picker	<a href="https://github.com/vuetifyjs/vuetify/issues/21885">https://github.com/vuetifyjs/vuetify/issues/21885</a>
21585	[Feature Request] Add Aria-labels to VTreeview Buttons	<a href="https://github.com/vuetifyjs/vuetify/issues/21585">https://github.com/vuetifyjs/vuetify/issues/21585</a>
20974	[Bug Report][3.7.11] Screen reader is not announcing the sorting information for the column headers under Nutrition section. [Data Table]	<a href="https://github.com/vuetifyjs/vuetify/issues/20974">https://github.com/vuetifyjs/vuetify/issues/20974</a>
20843	[Bug Report][3.7.3] Aria-Controls Attribute Not Present on Expanded V-Autocomplete	<a href="https://github.com/vuetifyjs/vuetify/issues/20843">https://github.com/vuetifyjs/vuetify/issues/20843</a>
13119	[Feature Request] Data Table – Accessible column headers	<a href="https://github.com/vuetifyjs/vuetify/issues/13119">https://github.com/vuetifyjs/vuetify/issues/13119</a>
10796	[Feature Request] v-treeview: Keyboard support	<a href="https://github.com/vuetifyjs/vuetify/issues/10796">https://github.com/vuetifyjs/vuetify/issues/10796</a>

Tabelle 3.2.: Übersicht der ausgewählten a11y-Issues im Vuetify-Repository

Für das Dokumentieren der Tests wurden zunächst Bewertungstabellen angelegt, die die für die Komponente relevanten WCAG-Kriterien aufführt, und die Testmethode sowie Platz für die Ergebnisse vorsieht. Die in den Issues genannten Probleme wurden ebenfalls in Bezug zu den ARIA gestellt und die betroffenen Kriterien fettgedruckt, was hier anhand Abbildung 3.1 verdeutlicht werden soll. Beispielsweise besagt Issue 20974 (s. Tabelle 3.2), dass Informationen zur Sortierung in der **VDataTable** nicht vom Screenreader vorgelesen werden, was dem WCAG-Kriterium 4.1.2 entspricht. Zu Dokumentationszwecken werden durch Issues reflektierte Defizite in der entsprechenden Tabelle fettgedruckt (s. Abb. 3.1). Dieser Rückbezug auf die Issues bestätigt, dass die schon belegten Probleme auch in den Tests und der Optimierung reflektiert werden. Die vollständigen Tabellen befinden sich im Anhang (s. A.2).

WCAG-Richtlinie	Kriterium	Testmethode	Bewertung	Anmerkung
1.3.1 (A) Info & Beziehungen	Struktur- und DOM-Beziehungen müssen erkennbar sein	auto/tool		
1.3.2 (A) Bedeutungstragende Reihenfolge	Reihenfolge muss softwarebestimmbar sein	auto/tool		
<b>2.1.1 (A) Keyboard</b>	Komponente muss vollständig mit Tastatur bedienbar sein	man		sorting buttons not accessible
2.4.3 (A) Fokusreihenfolge	Fokus bewegt sich logisch und vorhersehbar	man		
<b>4.1.2 (A) Name, Rolle, Wert</b>	Rolle, Name, Zustand müssen durch Software erkennbar sein	auto/tool		sorting buttons not accessible <a href="#">miro</a>

Abbildung 3.1.: Dokumentation der Testergebnisse für die Komponente **VDataTable**

Die Ergebnisse der Tests variierten zwischen einer bis zu vier nicht bestandenen Testkriterien. Die häufigsten Defizite traten dabei im „Lesbarkeit“-Bereich auf, ungefähr halb so viele in der Tastaturbedienbarkeit. Im folgenden Kapitel wird der Ursprung dieser Fehler im Quellcode erörtert und darauf aufbauend eine allgemeine Strategie für das Beseitigen dieser entwickelt. Die hier entstandenen Testergebnisse finden in Kapitel 3.3.4 erneut Verwendung, wo durch erneutes Testen sichergestellt wird, dass die Komponenten optimiert werden konnten.

### 3.3. Implementierung und Verifikation

Die Implementierungsphase beschäftigt sich mit der konkreten Optimierung der ausgewählten Komponenten und resultiert in dem für dieses Projekt zentralen Werk: den Pull Requests in das Vuetify-Repository. Grundlage für die Implementierung sind die im

voherigen Kapitel herausgearbeiteten Defizite in der Screenreader-Kompatibilität der betrachteten Komponenten. Wie in Kapitel 2 erläutert, sind mit nativem HTML und ARIA-Rollen zwei grundlegende Methoden gegeben, um Screenreader-Kompatibilität herzustellen oder zu verbessern. Dabei gilt der sogenannte HTML-first-Ansatz als Paradigma (W3C, 2026b), dem zufolge native HTML-Elemente bevorzugt eingesetzt werden sollen, sofern sie die benötigte Semantik und Funktionalität der Komponente mitbringen.

Die folgenden Unterkapitel beschäftigen sich zunächst mit der Analyse bestehender Accessibility-Patterns im Vuetify-Quellcode mit dem Ziel, daraus eine Implementierungsstrategie abzuleiten. Anschließend daran werden deren Anwendung, Verifikation und Einbindung in den Open-Source-Entwicklungsprozess beschrieben.

### 3.3.1. Analyse bestehender Accessibility-Patterns in Vuetify

Um eine gezielte Optimierung der Screenreader-Kompatibilität vornehmen zu können, ist zunächst ein Verständnis über Vuetifys bestehende Implementierungsansätze erforderlich. In Kapitel 2.3 wurde bereits die grundlegende Struktur des Komponenten-Quellcodes dargelegt. Aufbauend darauf werden nun die entsprechenden Stellen im Quellcode untersucht, an denen Screenreader-Kompatibilität bereits implementiert ist, vorrangig durch native HTML-Elemente und ARIA-Rollen sowie durch die manuelle Implementierung von Tastaturinteraktionsmöglichkeiten.

Die Analyse erfolgt komponentenübergreifend und betrachtet sowohl Haupt- als auch Teilkomponenten. Ziel ist es, vorhandene Implementierungsmuster zu identifizieren, um im Idealfall auf eine bestehende Strategie zurückgreifen zu können. Hierzu wurden die Orte der Implementierung von HTML-Elementen, ARIA-Rollen und Tastaturinteraktionen im Code dokumentiert.

Das Ergebnis dieser Analyse zeigt jedoch, dass Vuetify keine einheitliche Strategie zur Umsetzung von Screenreader-Kompatibilität verfolgt. Stattdessen lassen sich verschiedene Ansätze beobachten (s. Tabelle 3.3), darunter die Verwendung nativer HTML-Elemente, der Einsatz von ARIA-Rollen und -Attributaten sowie hybride Lösungen, bei denen semantische Informationen über Props oder dynamisch gesetzte Attribute ergänzt werden. Insgesamt ist eine deutlich häufigere Verwendung von ARIA-Attributaten im Vergleich zu nativen HTML-Elementen zu beobachten.

Die Tendenz zur Verwendung von ARIA lässt sich unter anderem durch den Anspruch an Wiederverwendbarkeit und visuelle Anpassbarkeit erklären, der in komponentenbasierten Frameworks wie Vuetify eine zentrale Rolle spielt und durch den Einsatz nativer HTML-Elemente in bestimmten Fällen eingeschränkt werden kann. Native

Nr.	Ansatz	Beobachtete Häufigkeit	Charakteristik
1	Native HTML-Semantik	13 Komponenten	Verwendung semantisch geeigneter nativer HTML-Elemente mit impliziter Screenreader-Accessibility, entspricht dem HTML-first-Ansatz
2	Dynamische HTML-Tags	5 Komponenten	Semantische HTML-Tags werden über konfigurierbare Props dynamisch gesetzt, gewährleistet Flexibilität und Wiederverwendbarkeit
3	ARIA-basierte Semantik	42 Komponenten	Semantik und Rolle werden explizit über ARIA-Rollen und -Attribute definiert, hohe Flexibilität mit erhöhter Implementierungsverantwortung

Tabelle 3.3.: Klassifikation der in Vuetify beobachteten Implementierungsansätze für Screenreader-Kompatibilität

HTML-Elemente bringen eine fest definierte Semantik sowie ein vordefiniertes Interaktionsmodell mit, die sich teilweise nicht mit den konfigurativen Anforderungen wiederverwendbarer Komponenten vereinbaren lassen. Dies führt dazu, dass semantisch passende HTML-Elemente zwar theoretisch vorhanden sind, in der Praxis im Framework-Kontext jedoch Einschränkungen verursachen würden.

Ein konkretes Beispiel hierfür ist die Implementierung von *Slider*- oder *RangeSlider*-Komponenten: Vuetify verwendet hier keine nativen `<input type="range">`-Elemente, sondern `<div>`-Container mit ergänzenden beschreibenden ARIA-Attributen. Dieser Ansatz erlaubt mehrere Griffe (die sogenannten *Slider-Thumbs*), dynamische Wertebereiche und flexible visuelle Anpassungen, die mit nativen `<input type="range">`-Elementen nicht realisierbar wären. Dieses Vorgehen erlaubt es, die Interaktionen, Fokussierung und visuelle Darstellung flexibel zu gestalten, stellt jedoch gleichzeitig eine Abweichung vom klassischen HTML-first-Ansatz dar. Die Abwägung der richtigen Implementierungsstrategie wird in Kapitel 3.3.2 weiter ausgeführt.

Auffällig ist zusätzlich, dass ARIA-Rollen und -Attribute häufig innerhalb der `useRender()`-Funktion direkt am gerenderten Tag der Komponente gesetzt werden. Diese Vorgehensweise ähnelt technisch der nativen Verwendung von ARIA in reinem HTML, bei der entsprechende Informationen ebenfalls als Attribute am Element hinterlegt werden. Für die im Rahmen dieser Arbeit vorgenommenen Implementierungen wurde dieser Ort daher ebenfalls gewählt, um sich an bestehende Konventionen im Vuetify-Quellcode anzulehnen und die Konsistenz der Accessibility-Implementierung zu erhöhen.

Tastaturinteraktionen wurden in nur zehn von den 76 Komponenten manuell implementiert, dabei vorrangig in Komponenten mit Dialog- oder dialogähnlichen Funktionen, also solchen, die Pop-Ups oder Overlays nutzen. Grund für die manuelle Implementierung ist, dass die relevanten Elemente in diesen Komponenten nicht in einer logischen Reihenfolge im DOM angeordnet sind. Dadurch lässt sich der Fokusfluss nicht automatisch über die standardmäßige Tab-Reihenfolge ableiten, was eine manuelle Umsetzung nötig macht.

Die insgesamt eher heterogene Umsetzung der Implementierung von Screenreader-Kompatibilität innerhalb von Vuetify, besonders in der Verwendung von ARIA-Auszeichnung erschwert eine direkte Übertragung bestehender Muster auf die hier betrachteten Komponenten. Vor diesem Hintergrund wird im folgenden Unterkapitel eine eigene, abstrahierte Implementierungsstrategie entwickelt, die als Entscheidungsgrundlage für die Optimierung der ausgewählten Komponenten dient.

### 3.3.2. Entwicklung einer Implementierungsstrategie

Die im vorherigen Unterkapitel identifizierte Heterogenität der bestehenden Implementierungen macht deutlich, dass für die gezielte Optimierung ausgewählter Vuetify-Komponenten eine eigenständige, systematische Entscheidungsgrundlage erforderlich ist. Ziel der hier entwickelten Implementierungsstrategie ist es, für Komponenten und deren Teilkomponenten nachvollziehbare und konsistente Entscheidungen bezüglich des Einsatzes nativer HTML-Elemente, ARIA-Rollen und -Attribute sowie gegebenenfalls notwendiger Ergänzungen der Tastaturinteraktion zu ermöglichen.

Zu diesem Zweck werden aus den zentralen ARIA-Dokumenten die relevanten Entscheidungsfragen in einem Entscheidungsbaum (Abbildung 3.2) gebündelt, aus dem sich pro Komponente eine geeignete Implementierungsstrategie ableiten lässt. Der Entscheidungsbaum stellt dabei einen Soll-Zustand dar und ist nicht an die in dieser Arbeit betrachteten Komponenten gekoppelt, sondern auch auf die Entwicklung neuer Komponenten übertragbar.

Der erste Entscheidungsknoten prüft, ob für die jeweilige Funktionalität ein semantisch geeignetes natives HTML-Element existiert. Die Platzierung dieser Entscheidung an erster Stelle reflektiert den oben benannten HTML-first-Ansatz. Ist ein entsprechendes Element vorhanden, wird in einem weiteren Schritt bewertet, ob dieses die funktionalen Anforderungen der Komponente vollständig abdeckt. Als „funktionale Anforderung“ gelten hierbei insbesondere Anforderungen an das Interaktionsverhalten der Komponente, etwa hinsichtlich Tastaturbedienbarkeit, Fokusmanagement, Abbildung semantischer Beziehungen zwischen Teilkomponenten sowie gestalterischer oder struktureller Flexibilität. Kann die gewünschte Interaktion allein durch natives HTML abgebildet werden, wird die Komponente ohne zusätzliche ARIA-Rollen implementiert. Reicht die native Semantik hingegen nicht aus, ist eine Erweiterung durch ARIA-Rolle und -Attribute möglich. Dieser Pfad bildet die in den APG beschriebenen Ausnahmeregeln ab, nach denen eine Erweiterung nativer HTML-Elemente durch ARIA zulässig ist, sofern dies aus gestalterischen, technischen oder strukturellen Gründen erforderlich ist (W3C, 2026c).

Existiert kein geeignetes natives HTML-Element, wird im rechten Entscheidungspfad geprüft, ob für die gewünschte Funktionalität eine passende ARIA-Rolle gemäß der WAI-ARIA-Spezifikation vorhanden ist. In diesem Fall erfolgt die Implementierung als ARIA-Widget unter Berücksichtigung der jeweils definierten Rollen, Zustände und Eigenschaften. Ist auch dies nicht möglich, sieht der Entscheidungsbaum als letzte Option die Entwicklung eines eigenen ARIA-Widgets vor, das sich aus semantisch sinnvollen Rollen und Attributen zusammensetzt.

Ein weiteres Merkmal des Entscheidungsbaums ist seine Rekursivität. Komplexe

Widgets werden in eigenständige Teilkomponenten untergliedert, die jeweils separat bewertet werden. Für jede dieser Teilkomponenten wird der Entscheidungsprozess erneut angewendet, wodurch auch bei komplexeren oder verschachtelt aufgebauten Komponenten eine konsistente Semantik sichergestellt werden kann.

Der Entscheidungsbaum definiert keinen konkreten Implementierungsleitfaden für einzelne Komponenten, sondern stellt Entscheidungsrahmen dar, dessen Ursprung in den relevanten Primärquellen liegt. Für die tatsächliche Umsetzung ist ergänzend die jeweils relevante Spezifikation (im Baum mit einem Buch-Icon vermerkt) sowie, falls vorhanden, der APG der entsprechenden Komponente heranzuziehen. Unabhängig vom gewählten Implementierungspfad sollte das Verfahren durch Tests jeder Komponente mit einem Screenreader abgeschlossen werden, um das reale Interaktionsmuster überprüfen zu können.

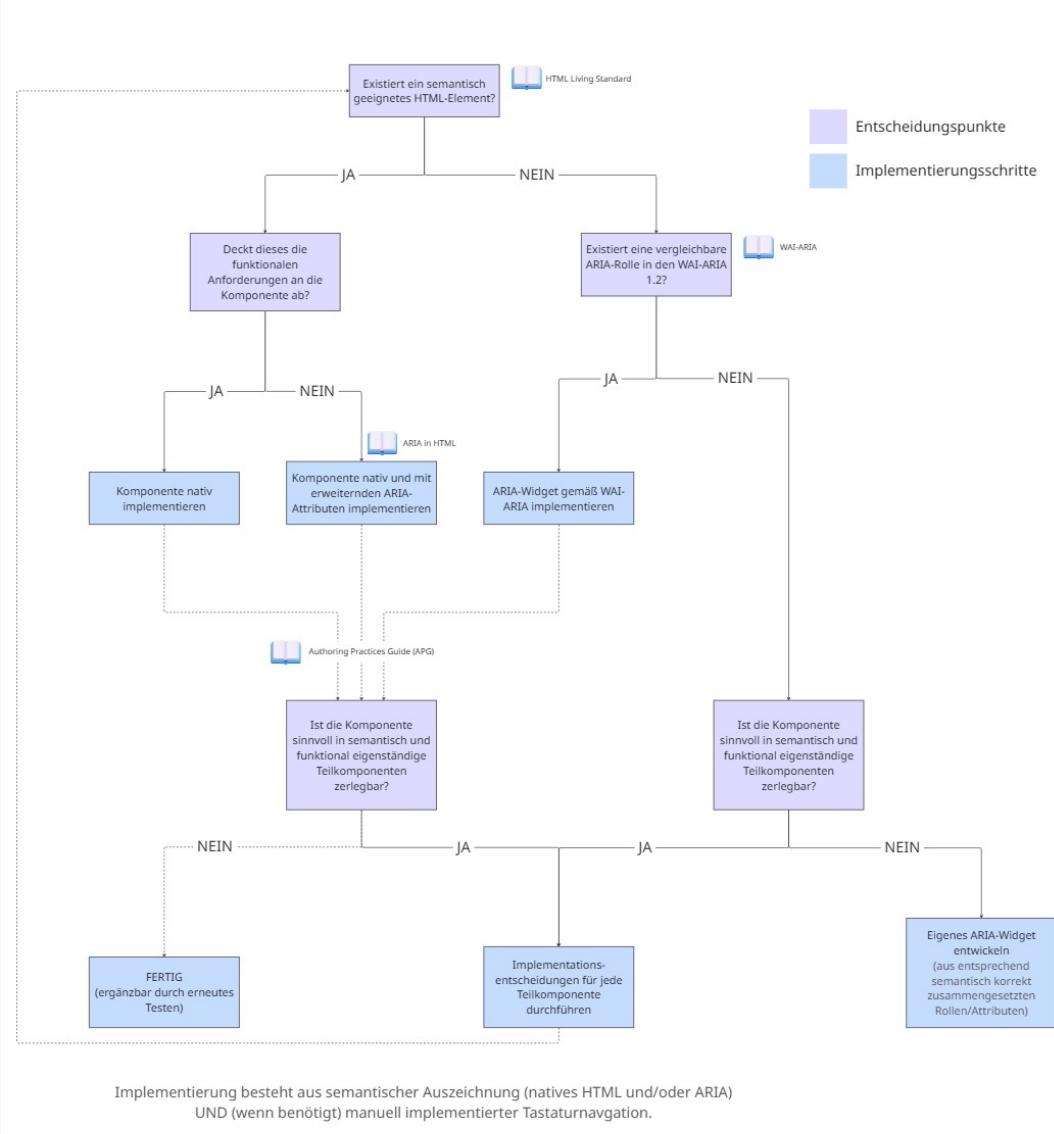


Abbildung 3.2.: Entscheidungsbaum für die Implementierungsstrategie

### 3.3.3. Praktische Umsetzung der Optimierungen

Die im vorherigen Unterkapitel entwickelte Implementierungsstrategie stellt keinen konkreten Implementierungsleitfaden dar. Stattdessen dient der Entscheidungsbaum als strukturierte Orientierung, um für jede betrachtete Komponente eine begründete Richtung für die Umsetzung der Screenreader-Kompatibilität abzuleiten. Die eigentliche Implementierung erfolgt kontextabhängig und muss an das Issue, die interne Struktur der jeweiligen Komponente sowie bestehende Konventionen im Vuetify-Quellcode angepasst werden.

Der Übergang vom Entscheidungsbaum zur konkreten Code-Implementierung erfolgt daher in mehreren Schritten:

- Analyse des Issues (Kap. 3.1) und der Tests (Kap. 3.2) mit dem Ziel, die konkrete Ursache der eingeschränkten Screenreader-Kompatibilität zu identifizieren und die dafür verantwortliche Haupt- oder Teilkomponente einzuschränken. Diese Fokussierung dient der Reduktion der Komplexität und vermeidet unnötige Eingriffe in unbeteiligte Komponenten.
- Bewertung der identifizierten (Teil-)Komponente entlang des Entscheidungsbaums, um zu bestimmen, ob eine native HTML-Semantik vorliegt und ob diese die funktionalen Anforderungen der Komponente aus Sicht der Screenreader vollständig abdeckt.
- Abgleich des Ist-Zustands der Komponente mit dem Soll-Zustand gemäß der vorhandenen Spezifikation (z.B. APG oder WAI-ARIA) mit Ableitung der notwendigen Anpassungen.
- Umsetzung der identifizierten Anpassungen im Vuetify-Quellcode, zum Beispiel durch das gezielte Setzen oder Ergänzen von ARIA-Attributen an bestehenden Elementen.

Dieses Vorgehen lässt sich exemplarisch an der Implementierung der Issues (20194 und 13119, s. Tabelle 3.2) zur fehlenden Ausgabe von Sortierinformationen in der **VDataTable** verdeutlichen. Dieses Problem betrifft nicht die gesamte Tabellenstruktur, sondern konkret ihre Kopfzeile, welche intern in der Teilkomponente **VDataTableHeaders** implementiert ist.

Die Anwendung des Entscheidungsbaums auf diese Teilkomponente führt zu dem Pfad „natives HTML mit ARIA-Ergänzung“, da die Tabellenstruktur bereits über semantisch geeignete HTML-Elemente wie `<thead>` und `<th>` realisiert ist, die funktionalen Anforderungen der Sortierinteraktion jedoch nicht vollständig abdecken.

Der anschließende Abgleich mit dem vorhandenen APG für sortierbare Tabellen zeigt, dass der aktuelle Sortierzustand über das Attribut `aria-sort` auszuzeichnen ist. Da dieser Zustand im bestehenden Vuetify-Code zwar intern verwaltet, jedoch nicht semantisch ausgezeichnet wird, ergibt sich eine Abweichung zwischen Soll- und Ist-Zustand. Die Implementierung besteht folglich aus einer Ergänzung der bestehenden Struktur durch dynamisch gesetzte `aria-sort`-Attribute an den Spaltenüberschriften.

Um den Beitrag in das Open-Source-Projekt vorzubereiten, wurden in diesem Schritt bereits die Contributing Guidelines von Vuetify gesichtet. Entsprechend folgt das Forken des Repositories und das Anlegen eines Branches pro Issue, worauf dann die Implementierung erfolgt. So kann im Anschluss (Kapitel 3.3.5) eine Pull Request pro Issue erstellt werden und mit diesem (den Guidelines folgend) verbunden werden.

### **3.3.4. Analyse und Verifikation der Optimierungen**

Wie in Kapitel 1 beschrieben, dient das erneute Testen der Komponenten der Verifikation der vorgenommenen Änderungen. Die Testergebnisse aus Kapitel 3.2 werden in diesem Unterkapitel erneut aufgegriffen und stellen die Vergleichsbasis dar, anhand derer die Wirkung der Implementierungen messbar gemacht werden kann.

Das Testverfahren wurde zur Sicherstellung der Vergleichbarkeit unverändert übernommen und entspricht in Aufbau und Durchführung exakt dem in Kapitel 3.2 und im Praxisprojekt beschriebenen Vorgehen. Nach der Implementierung der Optimierungen konnten alle betrachteten Komponenten die zuvor nicht bestandenen Tests erfolgreich bestehen. Damit lässt sich für die ausgewählten Komponenten eine Verbesserung ihrer Screenreader-Kompatibilität nachweisen. Ein beispielhafter Vergleich des Vorher-Nachher-Zustands einer Komponente wird in Abbildung deutlich, während die vollständigen Testergebnisse im Anhang dokumentiert sind (s. A.3).

### 3. Methodik

WCAG-Richtlinie	Kriterium	Testmethode	Bewertung	Anmerkung
1.3.1 (A) Info & Beziehungen	Struktur- und DOM-Beziehungen müssen erkennbar sein	auto/tool	grün	
1.3.2 (A) Bedeutungstragende Reihenfolge	Reihenfolge muss softwarebestimmbar sein	auto/tool	grün	
2.1.1 (A) Keyboard	Komponente muss vollständig mit Tastatur bedienbar sein	man	rot	sorting buttons not accessible
2.4.3 (A) Fokusreihenfolge	Fokus bewegt sich logisch und vorhersehbar	man	grün	
4.1.2 (A) Name, Rolle, Wert	Rolle, Name, Zustand müssen durch Software erkennbar sein	auto/tool	rot	sorting buttons not accessible miro

WCAG-Richtlinie	Kriterium	Testmethode	Bewertung	Anmerkung
1.3.1 (A) Info & Beziehungen	Struktur- und DOM-Beziehungen müssen erkennbar sein	auto/tool	grün	
1.3.2 (A) Bedeutungstragende Reihenfolge	Reihenfolge muss softwarebestimmbar sein	auto/tool	grün	
2.1.1 (A) Keyboard	Komponente muss vollständig mit Tastatur bedienbar sein	man	grün	
2.4.3 (A) Fokusreihenfolge	Fokus bewegt sich logisch und vorhersehbar	man	grün	
4.1.2 (A) Name, Rolle, Wert	Rolle, Name, Zustand müssen durch Software erkennbar sein	auto/tool	grün	miro

Abbildung 3.3.: Vergleich der Testergebnisse für die Komponente **VDataTable** vor und nach der Optimierung

#### 3.3.5. Einbindung in den Open-Source-Entwicklungsprozess

Nach der Durchführung der Implementierungen und deren Verifikation durch wiederholte Testdurchläufe bildet die Einbindung der Änderungen in das offizielle Vuetify-Repository den Abschluss des praktischen Teils dieser Arbeit. Die Optimierungen wurden hierzu in Form von Pull Requests gemäß den von Vuetify bereitgestellten Contribution-Guidelines (Vuetify, 2025b) eingereicht. Die Guidelines geben vor, dass die Titelung und Beschreibung der Pull Requests nach dem Conventional Changelog Standard erfolgen soll. Zusätzlich wird der Pull Request eine Referenz auf das gelöste Issue angefügt. Die erstellten Pull-Requests mit den zugrundeliegenden Entscheidungen sind ausführlich in einem eigenen Repository dokumentiert<sup>1</sup>.

<sup>1</sup><https://github.com/mjung2605/screenreader-ally-in-vuetify>

## 4. Diskussion

Die Ergebnisse dieser Arbeit zeigen, dass gezielte Optimierungen ausgewählter Vuetify-Komponenten die Screenreader-Kompatibilität messbar verbessern können. Vor der Implementierung wiesen die Komponenten `VTreeview`, `VDataTable`, `VAutocomplete` und `VTimePicker` verschiedene Barrieren wie fehlende ARIA-Rollen, unvollständige Tastatursteuerung oder unklare semantische Strukturen auf. Nach der Umsetzung der entwickelten Strategie konnten alle Testkriterien erfolgreich erfüllt werden. Damit wird nicht nur die Forschungsfrage dieser Arbeit beantwortet, sondern es kann auch ein konkreter Beitrag für die Open-Source-Community geleistet werden, da die Änderungen über Pull Requests in das offizielle Vuetify-Repository eingebracht wurden.

Die Arbeit verdeutlicht, dass Frameworks wie Vuetify nicht automatisch barrierefreie Komponenten bereitstellen. Besonders bei komplexen Widgets, wie Menüs oder Tabellen, treten Barrieren auf, die durch die Abstraktion der Framework-Komponenten bedingt sind. Das zeigt, dass Barrierefreiheit in Vuetify nicht mitgeliefert wird, sondern aktiv überprüft und implementiert werden muss. Die Analyse bestätigt, dass eine Mischung aus nativen HTML-Elementen und Einsatz von ARIA-Attributen nötig ist, um dynamische, flexible Komponenten gleichzeitig barrierearm zu gestalten.

Ein zentraler Punkt der Methodik ist die Verwendung der Bewertungstabellen aus dem Praxisprojekt. Diese Tabellen bieten eine praktische Struktur, um Tests nachvollziehbar zu dokumentieren: Sie listen die relevanten WCAG-Kriterien auf, geben die Testmethode an (manuell, automatisiert, mit Tools (Screenreader)) und erlauben eine einfache Bewertung per Ampelsystem. Für die vorliegende Arbeit wurden die Tabellen auf Screenreader-relevante Kriterien reduziert. Ihre Stärke liegt vor allem darin, dass sie einen konkreten Bezug zu den WCAG 2.1 herstellen, anstatt sich auf beliebige, oft verallgemeinerte Barrierefreiheits-Checklisten aus dem Internet zu stützen. Die Tabellen geben allerdings nicht vor, welche Kriterien konkret für welche Komponenten relevant sind. Die Auswahl muss von der durchführenden Person eigenständig anhand des offiziellen WCAG-Dokuments getroffen werden. Die Tabellen bieten also keinen vollständigen Rahmen für reproduzierbare Tests, sondern lediglich eine orientierende Strukturierung des Testverfahrens. Eine zukünftige Weiterentwicklung könnte eine fundierte Zuordnung von Komponenten zu WCAG-Kriterien beinhalten, um die Validität der Tests abzusichern.

Einen weiteren Teil der Arbeit stellt der entwickelte Entscheidungsbaum als orientierende Strategie für die Implementierung dar. Der Baum erlaubt eine nachvollziehbare, schrittweise Entscheidung, die auf den APG und den offiziellen Empfehlungen aus dem WAI-ARIA-Dokument basieren. Er macht außerdem die Abwägung zwischen HTML-first und der notwendigen Flexibilität für Vuetify-Komponenten explizit: Native HTML-Elemente bieten standardisierte Interaktionen und sind wartungsarm, können aber in dynamischen, wiederverwendbaren Widgets Einschränkungen verursachen. ARIA ergänzt diese Lücken, bringt aber die Verantwortung für korrektes Verhalten mit sich. Gleichzeitig gibt der Entscheidungsbaum allerdings keine konkreten Implementierungsschritte vor, sondern bietet eine grobe Richtung. Das Durchlaufen des Baumes erfordert außerdem eine Auseinandersetzung mit verschiedenen Dokumenten wie den ARIA oder ARIA in HTML und kann dadurch zeitaufwendig und komplex werden, besonders bei vielen Teilkomponenten.

Insgesamt zeigt die Arbeit, wie wichtig eine systematische Herangehensweise bei der Verbesserung der Barrierefreiheit ist. Die Optimierungen funktionieren, weil sie auf einer klaren Analyse, einer validierten Bewertungsmethode und einer durchdachten Entscheidungsstrategie basieren. Gleichzeitig wird deutlich, dass Barrierefreiheit nicht automatisch entsteht, sondern eine kontinuierliche Reflexion und Anpassung von Methodik und Implementierung erfordert. Auch die hier vorgelegte Methodik sollte weiter validiert und ausgebaut werden.

## 5. Fazit und Ausblick

Die vorliegende Arbeit hat gezeigt, dass gezielte Optimierungen der Vuetify-Komponenten die Screenreader-Kompatibilität messbar verbessert haben. Durch die Anwendung des entwickelten Entscheidungsbaums und die Umsetzung von ARIA-Attributen auf Basis von semantischem HTML konnten die identifizierten Barrieren in den Komponenten VTreeview, VDataTable, VAutocomplete und VTimePicker beseitigt werden. Die Verbesserungen wurden erfolgreich über Pull Requests in das Vuetify-Repository eingebbracht, sodass die Optimierungen direkt der Open-Source-Community zugutekommen.

Der praktische Beitrag dieser Arbeit liegt nicht nur in der konkreten Verbesserung der Komponenten, sondern auch in der Validierung und Weiterentwicklung der Bewertungsmethodik. Die aus dem Praxisprojekt übernommenen Tabellen haben sich als nützliche Grundlage erwiesen, um die Vorher-Nachher-Situation zu dokumentieren und die Umsetzung der WCAG-Kriterien systematisch zu prüfen. Zukünftig wäre es jedoch sinnvoll, eine standardisierte Zuordnung von Komponenten zu WCAG-Kriterien zu entwickeln, um die Methodik reproduzierbarer zu machen und Fehler bei der Auswahl relevanter Tests zu vermeiden.

Die entwickelten Strategien und der Entscheidungsbaum bieten eine systematische Grundlage für die Implementierung barrierefreier Komponenten in Vuetify und sind prinzipiell auf andere Frameworks übertragbar. Sie machen explizit die Balance zwischen HTML-first-Prinzip und notwendiger Flexibilität durch ARIA deutlich und können Entwicklern helfen, fundierte Entscheidungen zu treffen, statt sich auf Versuch- und-Irrtum zu verlassen. Gleichzeitig zeigen die Limitationen der Arbeit, dass diese Strategien nur ein Teil des Gesamtprozesses sind: Sie erfordern Verständnis für ARIA und semantisches HTML und sind bisher nur auf Vuetify und Windows/NVDA getestet. Issues, die andere Betriebssysteme oder Screenreader betreffen, wurden nicht behandelt, sollten aber in zukünftigen Untersuchungen berücksichtigt werden.

Weitere Schritte, die auf dieser Arbeit aufbauen könnten, sind unter anderem:

- Übertragung und Validierung der Methodik auf andere Komponenten von Vuetify oder anderen Frameworks
- Weiterführende Optimierungen von Vuetify-Komponenten, die z.B. in Kapitel 3.1 aussortiert wurden

- Vereinheitlichung der Barrierefreiheits-Implementierungsstrategie im Code von Vuetify (s. Kapitel 3.3.1)

Insgesamt verdeutlicht die Arbeit, dass Barrierefreiheit ein kontinuierlicher Prozess ist. Nur durch systematische Analyse und Implementierung lässt sich eine nachhaltige Verbesserung erreichen. Der kombinierte Ansatz aus Analyse, Entscheidungsbaum und praktischer Umsetzung liefert hierfür eine Grundlage, die sowohl dem Vuetify-Team als auch der Community als Orientierung dienen kann.

## Literaturverzeichnis

- Accessibility requirements for ICT products and services: EN 301 549 V3.2.1 (2021-03).* (2021). [https://www.etsi.org/deliver/etsi\\_en/301500\\_301599/301549/03.02.01\\_60/en\\_301549v030201p.pdf](https://www.etsi.org/deliver/etsi_en/301500_301599/301549/03.02.01_60/en_301549v030201p.pdf)
- American Foundation for the Blind. (2025). *Screen Reading Technology and Refreshable Braille Displays* [Abgerufen am 06.01.2026]. American Foundation for the Blind. <https://www.afb.org/blindness-and-low-vision/using-technology/assistive-technology-videos/screen-reading-technology>
- Barrierefreiheitsstärkungsgesetz, 2021, <https://bfsg-gesetz.de/>
- Behindertengleichstellungsgesetz (BGG), 2002, <https://www.gesetze-im-internet.de/bgg/BJNR146800002.html>
- Help Tech. (2026). *Braillezeilen* [Abgerufen am 09.01.2026]. <https://www.helptech.de/braillezeilen>
- International Journal for Research in Applied Science & Engineering Technology (IJRASET). (2022). Comparative Analysis on Front-End Frameworks for Web Applications. *International Journal for Research in Applied Science & Engineering Technology (IJRASET)*, 10(7), 298–307. <https://doi.org/10.22214/IJRASET.2022.45260>
- MDN Web Docs. (2025a). *Acessibility* [Abgerufen am 09.01.2026]. <https://developer.mozilla.org/en-US/docs/Web/Accessibility>
- MDN Web Docs. (2025b). *Screen reader - Glossary* [Abgerufen am 09.01.2026]. [https://developer.mozilla.org/en-US/docs/Glossary/Screen\\_reader](https://developer.mozilla.org/en-US/docs/Glossary/Screen_reader)
- REHADAT-Statistik. (2025). *Blindheit und Sehbehinderung* [Abgerufen am 07.01.2026]. REHADAT-Statistik. <https://www.rehadat-statistik.de/statistiken/behinderung/behinderungsarten/blindheit-und-sehbehinderung/>
- Richtlinie (EU) 2016/2102 über den barrierefreien Zugang zu Websites und mobilen Anwendungen öffentlicher Stellen, 2016, <https://eur-lex.europa.eu/legal-content/DE/TXT/?uri=CELEX%3A32016L2102>
- Richtlinie (EU) 2019/882 über Barrierefreiheitsanforderungen für Produkte und Dienstleistungen, 2019, <https://eur-lex.europa.eu/legal-content/DE/TXT/?uri=CELEX%3A32019L0882>
- Verordnung zur Schaffung barrierefreier Informationstechnik nach dem Behinderten-gleichstellungsgesetz (Barrierefreie-Informationstechnik-Verordnung - BITV 2.0), 2021, <https://www.buzer.de/BITV.htm>

- Vuetify. (2024a). *Vuetify - A Vue Component Framework* [Abgerufen am 06.01.2026]. <https://vuetifyjs.com/en/>
- Vuetify. (2024b). *Why Vuetify? - Vuetify* [Abgerufen am 06.01.2026]. <https://vuetifyjs.com/en/introduction/why-vuetify/#flexible-components>
- Vuetify. (2025a). *Vue Component Framework* [Abgerufen am 06.01.2026]. <https://github.com/vuetifyjs/vuetify>
- Vuetify. (2025b). *Vue Component Framework - Commit Guidelines* [Abgerufen am 06.01.2026]. <https://github.com/vuetifyjs/vuetify?tab=contributing-overfile#commit-guidelines>
- Vuetify. (2026). *Releases* [Abgerufen am 19.01.2026]. <https://github.com/vuetifyjs/vuetify/releases>
- W3C. (2014). *HTML5* [Abgerufen am 12.01.2026]. World Wide Web Consortium (W3C). <https://www.w3.org/TR/2014/REC-html5-20141028/>
- W3C. (2025a). *About WAI* [Abgerufen am 06.01.2026]. <https://www.w3.org/WAI/about/>
- W3C. (2025b). *Understanding the Four Principles of Accessibility* [Abgerufen am 06.01.2026]. <https://www.w3.org/WAI/WCAG21/Understanding/intro#understanding-the-four-principles-of-accessibility>
- W3C. (2025c). *WAI-ARIA Overview* [Abgerufen am 06.01.2026]. <https://www.w3.org/WAI/standards-guidelines/aria/>
- W3C. (2025d). *Web Content Accessibility Guidelines (WCAG) 2.1* [Abgerufen am 06.01.2026]. <https://www.w3.org/TR/WCAG21/>
- W3C. (2025e). *Web Standards* [Abgerufen am 06.01.2026]. <https://www.w3.org/standards/>
- W3C. (2026a). *ARIA Authoring Practices Guide* [Abgerufen am 07.01.2026]. World Wide Web Consortium (W3C). <https://www.w3.org/WAI/ARIA/apg/>
- W3C. (2026b). *Read Me First* [Abgerufen am 07.01.2026]. World Wide Web Consortium (W3C). <https://www.w3.org/WAI/ARIA/apg/practices/read-me-first/>
- W3C. (2026c). *Structural Roles* [Abgerufen am 12.01.2026]. World Wide Web Consortium (W3C). <https://www.w3.org/WAI/ARIA/apg/practices/structural-roles/>

## A. Anhang

### A.1. Übersicht der a11y-Issues aus dem Vuetify-Repository

Aufgrund ihrer Größe befindet sich die Tabelle in einem externen, öffentlich zugängigen Google-Sheets-Dokument unter <https://bit.ly/4jAFFIE>.

### A.2. Testdokumentation vor der Optimierung

WCAG-Richtlinie	Kriterium	Testmethode	Bewertung	Anmerkung
2.1.1 (A) Keyboard	Komponente muss vollständig mit Tastatur bedienbar sein	man		
2.4.3 (A) Fokusreihenfolge	Fokus bewegt sich logisch und vorhersehbar	man		
2.5.3 (A) Beschriftung im Namen	Label muss im zugänglichen Namen vorkommen	auto/tool		
3.2.2 (A) Bei Eingabe	Änderung eines UI-Elements führt nicht zur Kontextänderung oder wird angekündigt	tool		hier getestet für Dropdown-Funktionalität: Fokus verschiebt sich nicht unvorhersehbar, deswegen bestanden
4.1.2 (A) Name, Rolle, Wert	Rolle, Name, Zustand müssen durch Software erkennbar sein	auto/tool		wurde während Projektaufzeit behoben: Commit #21184 

Abbildung A.1.: Dokumentation der Testergebnisse für die Komponente `VAutocomplete`

## A. Anhang

---

WCAG-Richtlinie	Kriterium	Testmethode	Bewertung	Anmerkung
1.3.1 (A) Info & Beziehungen	Struktur- und DOM-Beziehungen müssen erkennbar sein	auto/tool		
1.3.2 (A) Bedeutungstragende Reihenfolge	Reihenfolge muss softwarebestimmbar sein	auto/tool		
2.1.1 (A) Keyboard	Komponente muss vollständig mit Tastatur bedienbar sein	man		sorting buttons not accessible
2.4.3 (A) Fokusreihenfolge	Fokus bewegt sich logisch und vorhersehbar	man		
4.1.2 (A) Name, Rolle, Wert	Rolle, Name, Zustand müssen durch Software erkennbar sein	auto/tool		sorting buttons not accessible 

Abbildung A.2.: Dokumentation der Testergebnisse für die Komponente **VDataTable**

WCAG-Richtlinie	Kriterium	Testmethode	Bewertung	Anmerkung
1.3.1 (A) Info & Beziehungen	Struktur- und DOM-Beziehungen müssen erkennbar sein	auto/tool		"Input fields have no label". Zusammenhang nicht erkennbar
1.3.2 (A) Bedeutungstragende Reihenfolge	Reihenfolge muss softwarebestimmbar sein	auto/tool		
2.1.1 (A) Keyboard	Komponente muss vollständig mit Tastatur bedienbar sein	man		ja: Uhr ist entbehrlich, Input-Felder reichen
2.4.3 (A) Fokusreihenfolge	Fokus bewegt sich logisch und vorhersehbar	man		
4.1.2 (A) Name, Rolle, Wert	Rolle, Name, Zustand müssen durch Software erkennbar sein	auto/tool		s. 1.3.1 + kein "aria-disabled" für entsprechende Werte

Abbildung A.3.: Dokumentation der Testergebnisse für die Komponente **VTimePicker**

## A. Anhang

WCAG-Richtlinie	Kriterium	Testmethode	Bewertung	Anmerkung
1.3.1 (A) Info & Beziehungen	Struktur- und DOM-Beziehungen müssen erkennbar sein	auto/tool		nur als Liste bezeichnet, kein Zusammenhang zwischen "Ästen" erkennbar
1.3.2 (A) Bedeutungstragende Reihenfolge	Reihenfolge muss softwarebestimmt sein	auto/tool		
2.1.1 (A) Keyboard	Komponente muss vollständig mit Tastatur bedienbar sein	man		
2.4.3 (A) Fokusreihenfolge	Fokus bewegt sich logisch und vorhersehbar	man		chronologisch ja, aber "doppelt" (Node und Checkbox)
4.1.2 (A) Name, Rolle, Wert	Rolle, Name, Zustand müssen durch Software erkennbar sein	auto/tool		miro

Abbildung A.4.: Dokumentation der Testergebnisse für die Komponente VTreeview

### A.3. Testdokumentation nach der Optimierung

WCAG-Richtlinie	Kriterium	Testmethode	Bewertung	Anmerkung
2.1.1 (A) Keyboard	Komponente muss vollständig mit Tastatur bedienbar sein	man		
2.4.3 (A) Fokusreihenfolge	Fokus bewegt sich logisch und vorhersehbar	man		
2.5.3 (A) Beschriftung im Namen	Label muss im zugänglichen Namen vorkommen	auto/tool		
3.2.2 (A) Bei Eingabe	Änderung eines UI-Elements führt nicht zur Kontextänderung oder wird angekündigt	tool		hier getestet für Dropdown-Funktionalität: Fokus verschiebt sich nicht unvorhersehbar, deswegen bestanden
4.1.2 (A) Name, Rolle, Wert	Rolle, Name, Zustand müssen durch Software erkennbar sein	auto/tool		wurde während Projektaufzeit behoben; Commit #21184 miro

Abbildung A.5.: Dokumentation der Testergebnisse für die Komponente VAutocomplete nach der Optimierung

## A. Anhang

---

WCAG-Richtlinie	Kriterium	Testmethode	Bewertung	Anmerkung
1.3.1 (A) Info & Beziehungen	Struktur- und DOM-Beziehungen müssen erkennbar sein	auto/tool		
1.3.2 (A) Bedeutungstragende Reihenfolge	Reihenfolge muss softwarebestimmt sein	auto/tool		
<b>2.1.1 (A) Keyboard</b>	Komponente muss vollständig mit Tastatur bedienbar sein	man		
2.4.3 (A) Fokusreihenfolge	Fokus bewegt sich logisch und vorhersehbar	man		
<b>4.1.2 (A) Name, Rolle, Wert</b>	Rolle, Name, Zustand müssen durch Software erkennbar sein	auto/tool		miro

Abbildung A.6.: Dokumentation der Testergebnisse für die Komponente **VDataTable** nach der Optimierung

WCAG-Richtlinie	Kriterium	Testmethode	Bewertung	Anmerkung
1.3.1 (A) Info & Beziehungen	Struktur- und DOM-Beziehungen müssen erkennbar sein	auto/tool		
1.3.2 (A) Bedeutungstragende Reihenfolge	Reihenfolge muss softwarebestimmt sein	auto/tool		
<b>2.1.1 (A) Keyboard</b>	Komponente muss vollständig mit Tastatur bedienbar sein	man		ja: Uhr ist entbehrlich, Input-Felder reichen
2.4.3 (A) Fokusreihenfolge	Fokus bewegt sich logisch und vorhersehbar	man		
<b>4.1.2 (A) Name, Rolle, Wert</b>	Rolle, Name, Zustand müssen durch Software erkennbar sein	auto/tool		miro

Abbildung A.7.: Dokumentation der Testergebnisse für die Komponente **VTimePicker** nach der Optimierung

## A. Anhang

WCAG-Richtlinie	Kriterium	Testmethode	Bewertung	Anmerkung
1.3.1 (A) Info & Beziehungen	Struktur- und DOM-Beziehungen müssen erkennbar sein	auto/tool		
1.3.2 (A) Bedeutungstragende Reihenfolge	Reihenfolge muss softwarebestimmtbar sein	auto/tool		
2.1.1 (A) Keyboard	Komponente muss vollständig mit Tastatur bedienbar sein	man		
2.4.3 (A) Fokusreihenfolge	Fokus bewegt sich logisch und vorhersehbar	man		
4.1.2 (A) Name, Rolle, Wert	Rolle, Name, Zustand müssen durch Software erkennbar sein	auto/tool		miro

Abbildung A.8.: Dokumentation der Testergebnisse für die Komponente VTreeview nach der Optimierung

## **Erklärung**

Ich versichere, die von mir vorgelegte Arbeit selbstständig verfasst zu haben. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder nicht veröffentlichten Arbeiten anderer oder der Verfasserin/des Verfassers selbst entnommen sind, habe ich als entnommen kenntlich gemacht. Sämtliche Quellen und Hilfsmittel, die ich für die Arbeit benutzt habe, sind angegeben. Die Arbeit hat mit gleichem Inhalt bzw. in wesentlichen Teilen noch keiner anderen Prüfungsbehörde vorgelegen.

Düsseldorf, 01.02.2026

Ort, Datum



Unterschrift