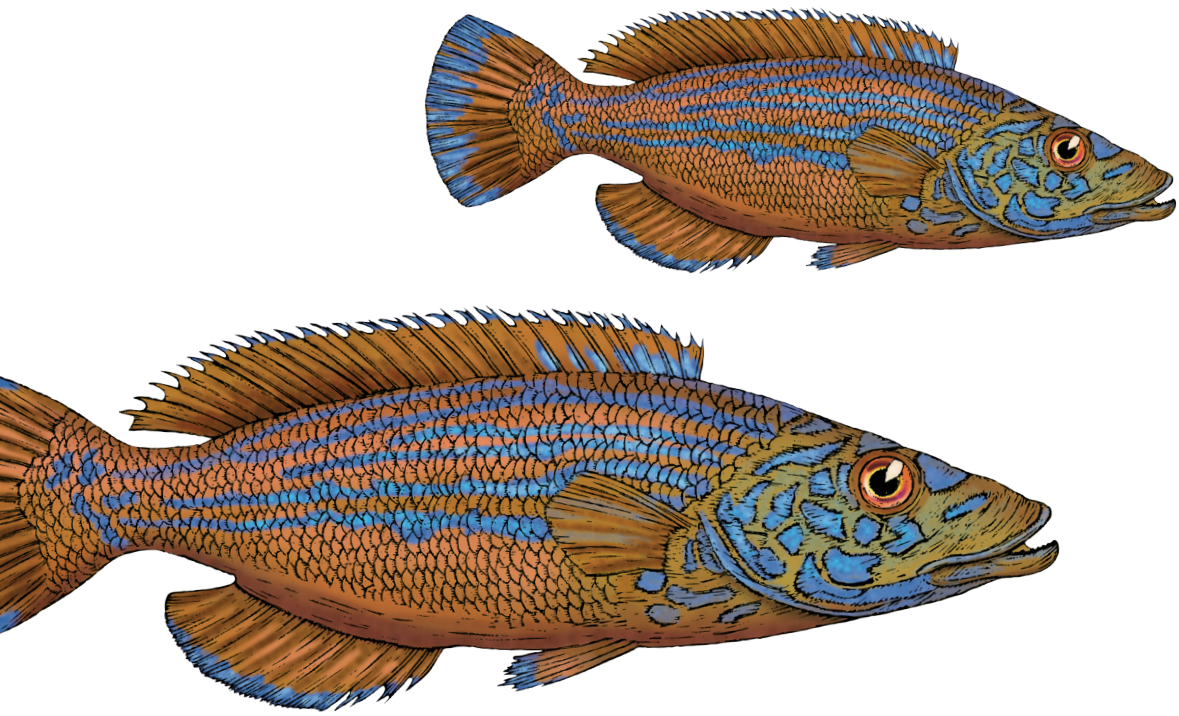
O'REILLY®

gemäß Barrierefreiheitsstärkungsgesetz

Barrierefreie Webentwicklung

Von den Grundlagen über die rechtlichen Aspekte bis zur praktischen Umsetzung



Maria Korneeva

3.2 Accessibility Tree

Der Accessibility Tree ist eine spezialisierte Struktur, die nur Informationen aus dem DOM enthält, die für assistive Technologien relevant sind. Zum Beispiel können Layout-Divs oder visuelle Dekorationselemente die Benutzerfreundlichkeit von Screenreadern beeinträchtigen, wenn sie nicht gefiltert werden. Nicht relevante Elemente werden weggelassen, und wichtige Informationen wie ARIA-Informationen werden hinzugefügt. Dies kann ARIA-Attribute, Rollen, Zustände und Beschreibungen umfassen, die speziell dafür vorgesehen sind, assistive Technologien zu unterstützen.

Den Unterschied und das Zusammenspiel zwischen dem DOM und dem Accessibility Tree möchte ich an folgendem Codebeispiel verdeutlichen. Es handelt sich dabei um ein HTML-Dokument, das aus einem Header, einem Hauptteil und einem Footer besteht. Der Header enthält eine Navigationsleiste mit einigen Links (im Code abgekürzt). Der Hauptteil hat die Überschrift »Über uns« und einen Artikel mit dem Titel »Unsere Werte«. Im Artikel wird ein Bild verwendet, das mit `alt=""` als dekorativ markiert wurde.

```
<body>
  <header>
    <nav>
      <a href="/home">Home</a>
      ...
    </nav>
  </header>

  <main>
    <h1>Über uns</h1>
    <article>
      <h2>Unsere Werte</h2>
      <p>Lorem ipsum</p>
      <img alt="" src="smile.png" />
    </article>
  </main>

  <footer>...</footer>
</body>
```

Listing 3.15: Ein einfaches HTML-Dokument mit strukturellen, inhaltlichen und dekorativen Elementen

Vergleicht man die beiden baumartigen Strukturen in Abbildung 3.9, sieht man einen eindeutigen Zusammenhang. Aus dem `<header>` wird ein Banner, aus dem `<main>` ein Main usw. Lediglich dekorative bzw. bewusst von assistiven Technologien versteckte HTML-Elemente landen nicht im Accessibility Tree und sind nur im DOM vorhanden.

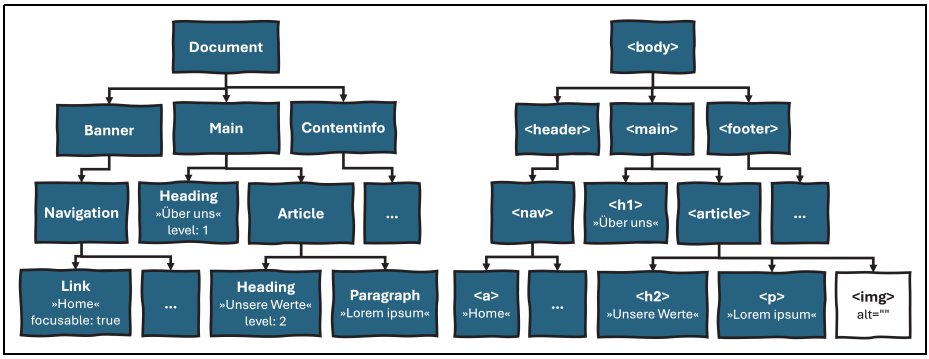


Abbildung 3.9: Code-Abstraktion als Accessibility Tree (links) und DOM (rechts)

3.2.1 Struktur und Inhalte eines Accessibility Trees

Der Accessibility Tree ist eine hierarchische Struktur, ähnlich wie das DOM. Er besteht aus Knoten (Nodes), die in einer Baumstruktur angeordnet sind. Die Knoten sind in Eltern-Kind-Beziehungen organisiert, was bedeutet, dass jedes Element eine hierarchische Position im Baum einnimmt. Dies spiegelt die Struktur und Beziehungen der Elemente auf der Webseite wider. Jeder Knoten im Baum repräsentiert ein Element auf der Webseite, das für die barrierefreie Bedienung relevant ist. Diese Knoten enthalten Informationen über das Element, einschließlich seiner Rolle, seines Namens, seines Zustands und seiner Eigenschaften.

Die **Rolle (role)** eines Elements beschreibt seine Funktion auf der Seite. Beispiele für Rollen sind `"button"`, `"link"`, `"heading"`, `"checkbox"` und `"image"`. Die Rolle hilft assistiven Technologien, zu verstehen, wie mit dem Element interagiert werden soll. Der zugängliche **Name (name)** eines Elements ist eine textuelle Beschreibung, die das Element identifiziert. Dies kann durch den Inhalt des Elements, durch ARIA-Labels oder durch alternative Textattribute bereitgestellt werden. Der Name ist entscheidend, um BenutzerInnen zu vermitteln, was das Element darstellt oder wozu es dient (mehr dazu in Abschnitt 3.3). **Zustände und Eigenschaften (states and properties)** geben zusätzliche Informationen über das Element an. Zustände können dynamische Eigenschaften wie `"activated"`, `"focused"` oder `"selected"` umfassen. Eigenschaften sind statische Attribute wie `"required"` oder `"autocomplete"`. Es können auch Interaktionsmöglichkeiten für das jeweilige Element ergänzt werden: Ein Link kann verfolgt werden, ein Button kann geklickt werden, usw. Die **Beschreibung (description)** liefert zusätzliche kontextuelle Informationen über das Element, die nicht durch den Namen oder die Rolle vermittelt werden können. Dies kann durch das `aria-describedby`-Attribut bereitgestellt werden (mehr dazu ebenso in Abschnitt 3.3). Einige Elemente haben **Werte**, die ihren aktuellen Zustand oder Inhalt darstellen. Beispielsweise kann ein Schieberegler (Slider) einen numerischen Wert haben, der seine Position angibt. Die Baumstruktur des Accessibility Trees vermittelt zusätzlich Informationen hinsichtlich

Beziehungen zwischen einzelnen Knoten (Elternteil/Kind, Beschreibung/beschriebenes Objekt, vorheriges Objekt/nächstes Objekt etc.).

Abbildung 3.10 zeigt den Accessibility Tree zum Codebeispiel aus der Einleitung dieses Abschnitts. Der Knoten `<h2>Das WIR</h2>` ist gerade ausgewählt. Mittig in der Abbildung befindet sich der Accessibility Tree selbst. Rechts sehen Sie die Informationen, die im Accessibility Tree zu diesem Knoten enthalten sind: der Name »Das WIR«, die ARIA-Rolle »heading« und der Level 2 (für `<h2>`). Die Inhalte variieren je nach Element. Links und Buttons haben zum Beispiel zusätzlich eine Angabe, ob sie per Tastatur fokussierbar sind. Bei Input-Feldern wird vermerkt, ob es sich dabei um Pflichtangaben handelt. Der Smiley taucht im Accessibility Tree gar nicht auf, da er als dekorativ markiert wurde.

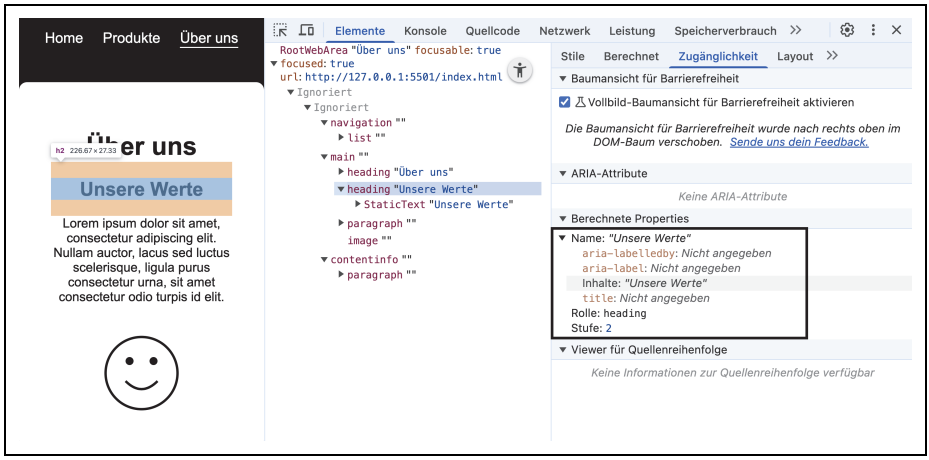


Abbildung 3.10: Informationen zu einem Element im Accessibility Tree

3.2.2 Accessibility Tree und assistive Technologien

Der Browser spielt eine zentrale Rolle bei der Erstellung des Accessibility Trees. Er analysiert das DOM und den Renderbaum, um relevante Informationen für den Accessibility Tree zu extrahieren. Der Accessibility Tree wird dynamisch aktualisiert, wenn sich das DOM ändert. Die Rendering Engine des Browsers nutzt dazu interne Algorithmen, um zu entscheiden, welche Elemente aufgenommen werden sollen. Blink (Chromium), Gecko und Webkit haben zwar eigene Mechanismen zur Erstellung des Accessibility Trees, nutzen aber eine gemeinsame Basis: Accessibility API Mappings (AAM).

AAM umfassen Core Accessibility API Mappings 1.2 und darauf basierende Mappings für HTML und SVG. Sie definieren, wie HTML- und SVG-Elemente auf die zugänglichen ARIA-Rollen, Zustände und Eigenschaften in den verschiedenen Accessibility APIs abgebildet werden sollen. Diese werden vom Betriebssystem bereitgestellt, das dafür sorgt, dass assistive Technologien über diese APIs mit An-

wendungen kommunizieren können. Folgende APIs stehen je nach Betriebssystem zur Verfügung:

- **Windows:** Microsoft Active Accessibility (MSAA), erweitert durch IAccessible2 (IA2), UIAExpress (Internet Explorer 8.0–11, veraltet), UI Automation (UIA) (Nachfolger von MSAA für Microsoft). Browser unter Windows können zwischen den beiden (MSAA oder UI Automation) entscheiden.
- **macOS:** NSAccessibility (AXAPI)
- **Linux:** Assistive Technology Service Provider Interface AT-SPI für Browser, Accessibility Toolkit (ATK) für assistive Technologien
- **ChromeOS:** chrome.accessibilityFeatures API
- **Android:** Accessibility framework
- **iOS:** UIAccessibility

Diese APIs ermöglichen es dem Browser, den Accessibility Tree an unterschiedliche assistive Technologien (Braille-Displays, Screenreader, Sprachsteuerungssoftware) weiterzugeben. Sie bieten eine Schnittstelle, über die diese Technologien auf die Informationen im Accessibility Tree zugreifen können. Bei Interaktionen geht die Informationskette in die andere Richtung. Klicken BenutzerInnen auf einen Button, leiten die assistiven Technologien die Aktion über die Accessibility-API an die Webseite weiter. Die Webseite reagiert darauf und aktualisiert ggf. das DOM und den Accessibility Tree. Events können auch browserseitig getriggert werden, z. B. wenn Inhalte nachgeladen oder aktualisiert werden. In diesem Fall benachrichtigt der Browser über die Accessibility-API das Betriebssystem, das die Benachrichtigung an die assistiven Technologien weitergibt. Zusammengefasst lässt sich die Kommunikationskette wie folgt darstellen (siehe Abbildung 3.11).

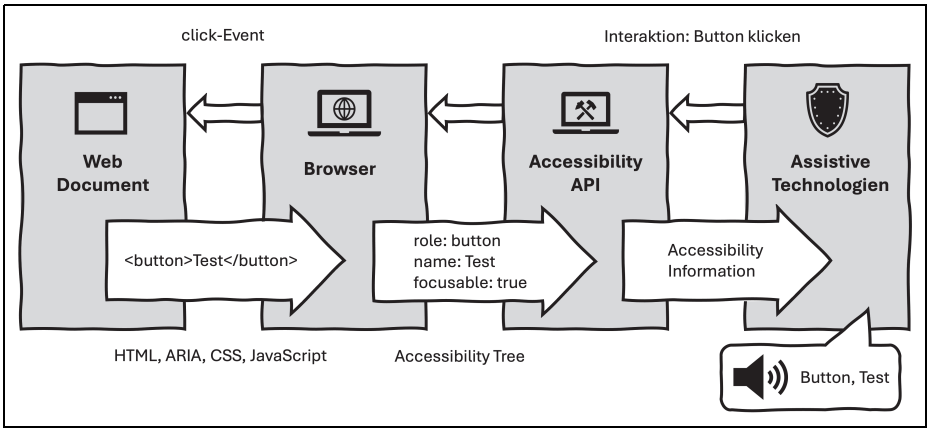


Abbildung 3.11: Der Browser analysiert das DOM und den Renderbaum, um den Accessibility Tree zu erstellen. Der Accessibility Tree wird über die Accessibility-APIs des Betriebssystems an assistive Technologien weitergegeben. Das Betriebssystem sorgt dafür, dass assistive Technologien über diese APIs auf den Accessibility Tree zugreifen können.

Drei Akteure sind daran beteiligt, dass relevante Informationen an die NutzerInnen der assistiven Technologien weitergereicht werden: Browser, Accessibility-APIs des jeweiligen Betriebssystems und die assistive Technologie selbst. Dazu kommen verschiedene Softwareversionen (auf allen drei Seiten). Das erschwert das Testen enorm und macht das Testen aller möglichen Kombinationen unrealistisch. Diese kann ich nicht einmal in Abbildung 3.12 vollständig abbilden, weil mir eine Dimension fehlt. Zusätzlich erhöht diese Vielzahl der Optionen potenzielle Kompatibilitätsprobleme. Einen Vorgeschmack für einige bekannte Bugs bekommen Sie beim Lesen des Artikels »It's Mid-2022 and Browsers (Mostly Safari) Still Break Accessibility via Display Properties«[28] von Adrian Roselli. »Accessibility Support«[29] und »Assistive Technology Compatibility Tests«[30] bieten umfangreiche Übersichten darüber, welche Lösungen (HTML, CSS, ARIA) von welchen Screenreadern und Sprachsteuerungssoftware unterstützt werden.

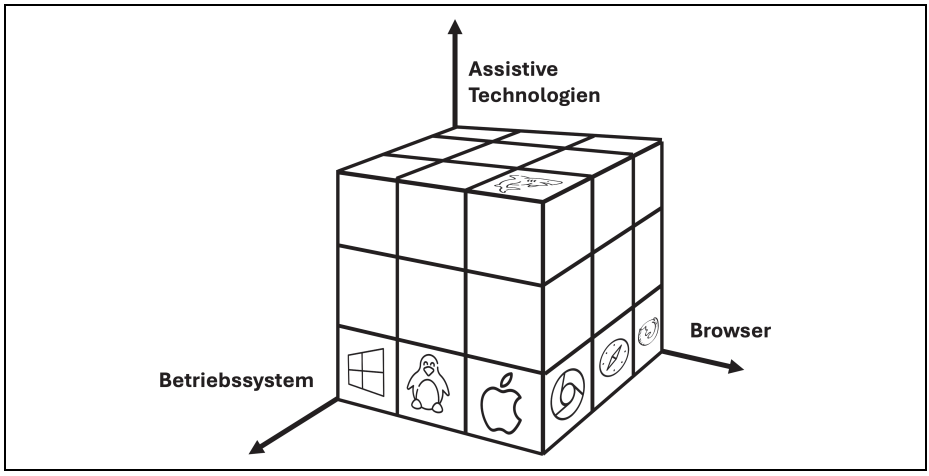


Abbildung 3.12: Würfel-Matrix der möglichen Kombinationen aus Browsern und Accessibility-APIs des jeweiligen Betriebssystems und der assistiven Technologie

Womit sollen Sie testen? Idealerweise wissen Sie, welche Tools Ihre NutzerInnen verwenden. Die Information zu den eingesetzten Screenreadern (und anderen assistiven Technologien) wird nicht standardmäßig ermittelt, deswegen können Sie hier wahrscheinlich nur Umfragen verwenden. Wenn das nicht möglich ist, bleibt Ihnen die Annahme über die gängigen Kombinationen von Browsern und Screenreadern (andere assistive Technologien sind nicht so vielfältig). Als Grundlage für Ihre Abwägungen können Sie zumindest teilweise die jährliche WebAIM-Umfrage[31] nutzen. Die Ergebnisse sind eventuell nur eingeschränkt auf Ihre Kund-

28 *https://adrianroselli.com/2022/07/its-mid-2022-and-browsers-mostly-safari-still-break-accessibility-via-display-properties.html*

29 *https://a11ysupport.io*

30 *https://www.powermapper.com/tests*

31 *https://webaim.org/projects/screenreadersurvey10*

schaft übertragbar, da es sich um ein internationales Publikum mit einem großen Anteil von NordamerikanerInnen handelt (47,2 %). Eine Studie, die zumindest für die EU relevanter wäre, ist mir aktuell nicht bekannt. Zu den eingesetzten Screenreader-Versionen gibt es auch keine Informationen. Es ist jedoch durchaus bekannt, dass oft veraltete Versionen von Screenreadern und Betriebssystemen genutzt werden, da Updates bei manchen Screenreadern (z. B. JAWS) kostenpflichtig und Systemupdates mit Anschaffung neuer Geräte verbunden sind. Also bleibt hier die allgemeine Annahme, dass Chrome in Kombination mit JAWS (24,7 %) bzw. NVDA (21,3 %) am häufigsten unter Windows (86,1 %) verwendet wurde. Manche Quellen, wie z. B. Accessibility Developer Guide[32] oder Assistiv Labs[33], verweisen jedoch auf die Kombination von NVDA und Firefox, da sie die wenigsten Bugs hat und die meisten Implementierungsmankos ausgleichen kann. Auf mobilen Geräten herrscht VoiceOver mit iOS vor (70,6 %). Für andere assistive Technologien (Switches, Braille-Displays, Sprachsteuerungssoftware etc.) gibt es gar keine nennenswerten Daten. Umso wichtiger ist es, dass die Informationen im Accessibility Tree korrekt und vollständig sind.

Der Accessibility Tree kann nur indirekt durch JavaScript, HTML und CSS beeinflusst werden, indem man das DOM anpasst. Aus dem Wunsch, auf den Accessibility Tree programmatisch zuzugreifen, ist das Accessibility Object Model (AOM)[34] entstanden. Das AOM soll die Lücke zwischen der Darstellung von Webseiten im DOM und der Wahrnehmung von diesen Inhalten durch assistive Technologien schließen. Mit AOM sollen z. B. Webkomponenten ihre Standardsemantik ohne ARIA deklarieren können, was verhindert, dass diese Implementierungsdetails ins DOM »durchsickern«. Um ARIA-Beziehungen zu spezifizieren, muss derzeit eine eindeutige ID auf jedem Element angegeben werden, um Elemente programmatisch miteinander in eine Relation zu bringen. Dies kann bei großen UI-Elementen, die sich auf `aria-activedescendant` und weitere ARIA-Beziehungen verlassen, sehr umständlich und unübersichtlich sein. Das AOM soll die Notwendigkeit eliminieren, mit IDs arbeiten zu müssen. Das AOM bietet auch nicht nativen Elementen (Webkomponenten) die Möglichkeit, auf Eingabeereignisse zu reagieren, die speziell für BenutzerInnen mit Behinderungen nützlich sind, wie z. B. inkrementelle oder dekrementelle Aktionen für benutzerdefinierte Slider oder Paging-Aktionen für Scrollansichten. Darüber hinaus ermöglicht das AOM das Hinzufügen von Nicht-DOM-Knoten (»virtuellen Knoten«) zum Accessibility Tree: Dies ist nützlich für komplexe UIs, die beispielsweise aus einem `<canvas>`-Element bestehen oder einen Remote-Desktop in einem `<video>`-Element streamen.

32 *https://www.accessibility-developer-guide.com/knowledge/screen-readers/relevant-combinations*

33 *https://assistivlabs.com/articles/screen-reader-browser-pairing-stats*

34 *https://wicg.github.io/aom*

Das Projekt befindet sich noch im Entwurfsstadium innerhalb der Web Incubator Community Group des W3C. Es wird derzeit als experimentelle Funktion in Browsern wie Chrome und Safari getestet und ist standardmäßig deaktiviert.

3.2.3 In den Accessibility Tree hineinschauen

Die gute Nachricht ist, dass die gängigen Browser den Accessibility Tree in den Entwicklertools darstellen. So können Sie überprüfen, ob die assistiven Technologien korrekte und vollständige Informationen bekommen, ohne sie selbst jedes Mal zu nutzen. In Chrome und Edge[35] sehen Sie den Accessibility Tree, wenn Sie zum Elemente-Panel in den Entwicklertools navigieren, einen Knoten auswählen und das Zugänglichkeit-Panel anklicken (siehe Abbildung 3.13). Rechts erscheinen Informationen zu dem ausgewählten Knoten: seine Elternknoten, ARIA-Attribute (falls vorhanden) und berechnete Eigenschaften wie Name, Beschreibung, Role und weitere Attribute (mehr dazu in Abschnitt 3.3).

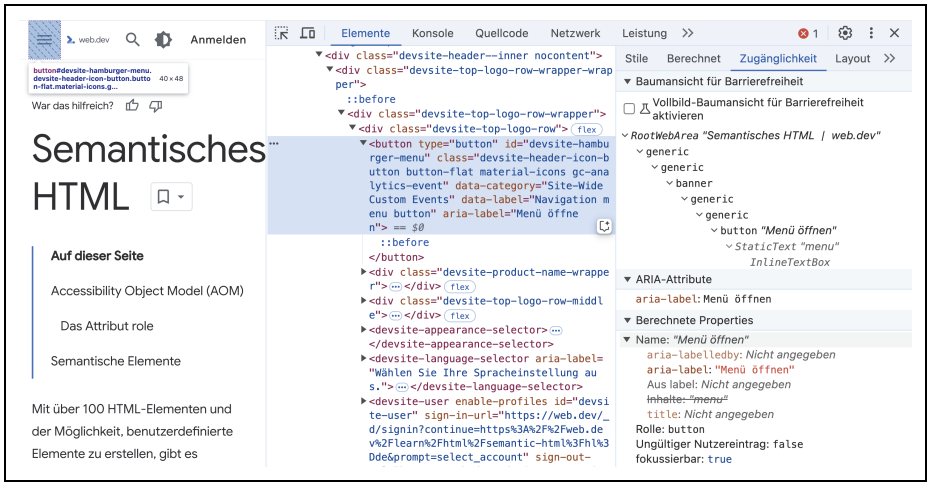


Abbildung 3.13: Informationen zur ARIA-Rolle und -Attributen sowie zum zugänglichen Namen im Zugänglichkeit-Panel des Tabs »Elemente« in Chrome

Sie können den vollständigen Accessibility Tree anzeigen lassen, indem Sie ein entsprechendes Häkchen aktivieren und die Entwicklertools neu laden (siehe Abbildung 3.14).

Nach dem Neuladen erscheint ein neuer Button direkt in der DOM-Ansicht, mit dem Sie zu der Accessibility-Tree-Ansicht wechseln können (siehe Abbildung 3.15).

35 *https://developer.chrome.com/docs/devtools/accessibility/reference*

Abbildung 3.14: Das Häkchen zur Aktivierung der vollständigen Ansicht des Accessibility Tree in Chrome

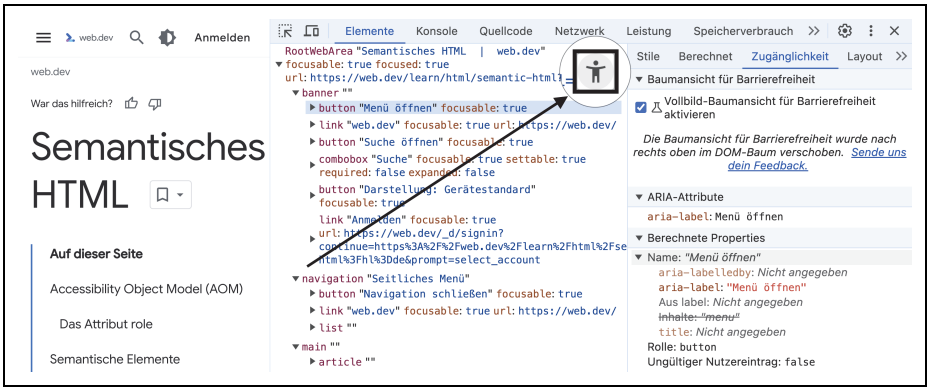


Abbildung 3.15: Der Toggler zwischen der DOM-Ansicht und dem Accessibility Tree in Chrome

Firefox zeigt den vollständigen Accessibility Tree standardmäßig in seinem Barrierefreiheit-Panel an (siehe Abbildung 3.16). Zu dieser Ansicht gelangen Sie ebenfalls, wenn Sie direkt mit der rechten Maustaste auf ein Element in der Webseite klicken und im Kontextmenü die Option »Barrierefreiheit-Eigenschaften untersuchen« auswählen. Das Panel enthält auch weitere Informationen und Funktionen aus dem Bereich Barrierefreiheit, die in relevanten Abschnitten vorgestellt werden. Beim Rechtsklick auf einen Knoten können Sie den Accessibility Tree als JSON in einem separaten Tab (JSON Viewer) ausgeben lassen.

Safari zeigt den vollständigen Accessibility Tree gar nicht an, sondern lediglich Informationen zu jedem Knoten (siehe Abbildung 3.17).[36] Diese befinden sich im

36 *https://support.apple.com/en-gb/guide/safari-developer/dev160f70435/mac*

Elemente-Panel in dem Tab »Knoten« (Node) im Abschnitt »Bedienungshilfen« (»Accessibility«) und sind im Vergleich zu anderen Browsern überschaubar.

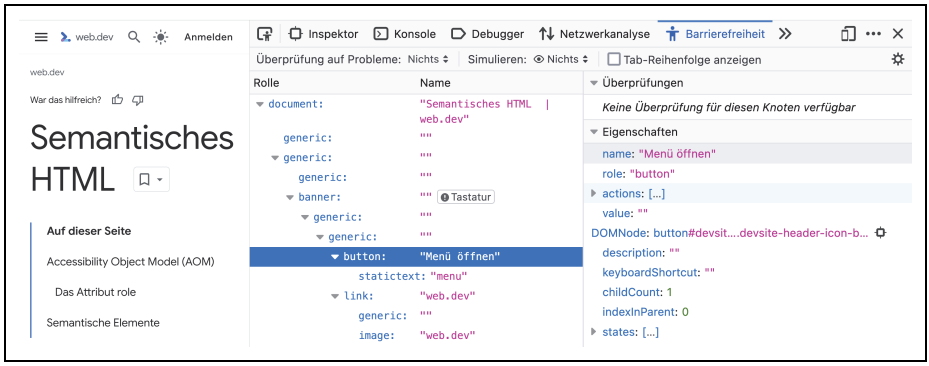


Abbildung 3.16: Accessibility Tree im Barrierefreiheit-Panel in Firefox

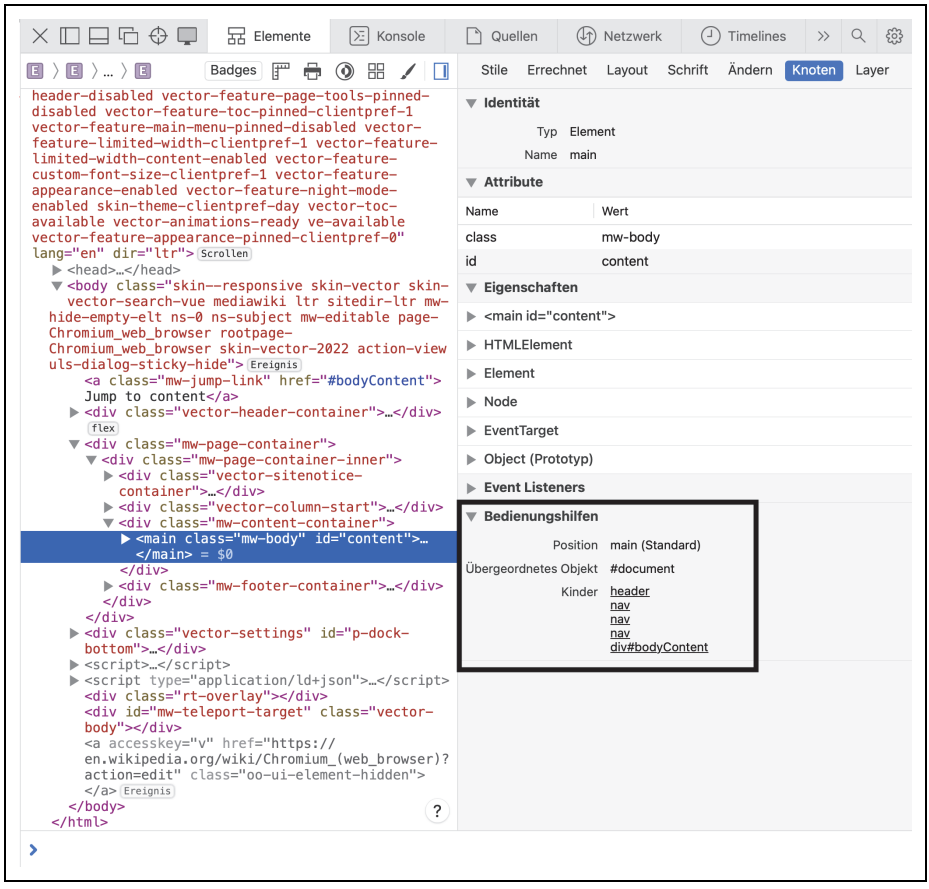


Abbildung 3.17: Die Bedienungshilfen zum jeweiligen DOM-Knoten in Safari

Der Accessibility Tree in den Entwicklertools – Vorsicht vor Abstraktion!

Die Möglichkeit, direkt in den Entwicklertools in den Accessibility Tree zu schauen, ist extrem wertvoll. So müssen Sie nicht jedes Mal eine der assistiven Technologien »anschmeißen«, um Ihren Code zu testen. Bei dem Märchen »Der Browser und der Accessibility Tree« wäre man bei »und wenn sie nicht gestorben sind, dann leben sie noch heute«, wenn da nicht dieser Satz aus einem Artikel von Adrian Roselli wäre, der mich dazu gebracht hat, die Anzeige des Accessibility Trees in den Entwicklertools genauer zu überprüfen: »Das Problem besteht darin, dass die Barrierefreiheitsinspektoren **eine Abstraktion des Accessibility Trees** darstellen. Das bedeutet, dass sie möglicherweise **nicht übereinstimmen**.«[37]

Der Chrome-for-Developers-Blog[38] führt aus: Der Barrierefreiheitsinspektor nutzt dieselben Accessibility-APIs, die assistive Technologien über Änderungen am Accessibility Tree informieren, und verwendet diese, um Ereignisse mit aktualisierten Knoten an das Entwicklertools-Frontend auszugeben. Da sich die Accessibility-APIs je nach Betriebssystem unterscheiden, können auch die Namen der Attribute für Baumknoten unterschiedlich sein. Die Entwicklertools zeigen stattdessen die interne Struktur von Chromium, in der Rollen und Attribute tendenziell mit den in der *ARIA*-Spezifikation definierten Rollen und Attributen übereinstimmen (Abstraktion!). Der Inspektor zeigt auch die ignorierten Knoten, die für den Accessibility Tree irrelevant sind, um eine 1:1-Beziehung zum DOM und schnelleres Rendering der Visualisierung zu unterstützen.

Laut Melanie Richards ist auch die hierarchische Struktur des Accessibility Trees eine gewisse Abstraktion.[39] Zum Beispiel ist es nicht unbedingt erforderlich, alle Knoten im Accessibility Tree für jedes Element im DOM zu generieren, sobald der DOM-Baum konstruiert wird. Aus Performancegründen können sich Browser dafür entscheiden, nur mit einer Teilmenge von Objekten und deren Beziehungen zu arbeiten – nämlich nur so viel, wie nötig ist, um die Anfragen von unterstützenden Technologien zu erfüllen. Die Rendering-Engine könnte diese Berechnungen immer ausführen oder nur dann, wenn assistive Technologien aktiv laufen.

Obwohl Entwicklertools im Browser viel über die Barrierefreiheit Ihres Markups verraten, zeigen sie normalerweise keine Barrierefreiheitsevents an. Unter

37 *https://adrianroselli.com/2020/12/be-careful-with-dynamic-accessible-names.html*

38 *https://developer.chrome.com/blog/full-accessibility-tree*. Für technische Dokumentation darüber, wie Chromium Informationen für die Accessibility-APIs aufbereitet, schauen Sie sich die Dokumentation »Accessibility Overview« und »How Chrome Accessibility Works« an: *https://chromium.googlesource.com/chromium/src/+/master/docs/accessibility/overview.md* und *https://source.chromium.org/chromium/chromium/src/+/main:docs/accessibility/browser/how_a11y_works.md*

39 *https://alistapart.com/article/semantics-to-screen-readers*

Windows gibt es im Windows SDK erweiterte Tools wie Inspect[40] und AccEvent[41], die beim Debuggen von MSAA- oder UIA-Zuordnungen helfen können. Auf macOS gibt es in Xcode den Barrierefreiheitsinspektor[42], mit dem Sie Webinhalte in Safari überprüfen können.

Da der Accessibility Tree in Entwicklertools also »nur« eine Abstraktion ist, empfehle ich Ihnen, kritische Stellen trotzdem mit assistiven Technologien unter verschiedenen Betriebssystemen zu testen.

Der Accessibility Tree ist ein Vermittler (sogar **der** Vermittler) zwischen Ihrer Anwendung und den assistiven Technologien. Wenn der Knoten im DOM fehlt, kann er (ohne AOM-Manipulation) nicht im Accessibility Tree erscheinen. Wenn die Information im Accessibility Tree fehlt, kann sie gar nicht an die Accessibility-APIs weitergegeben werden und fehlt dann den assistiven Technologien. Deswegen ist es wichtig, das Vorhandensein aller relevanten Details über Entwicklertools zu überprüfen. Stimmen die Rollen und Bezeichnungen, ist der Job des Entwicklungsteams jedoch noch nicht erledigt. Der Accessibility Tree hat keinen Einfluss auf das Verhalten des Browsers bzw. das Verhalten der Elemente. Dass etwas als Button aufgelistet ist, heißt noch lange nicht, dass es sich auch wie ein Button verhält. Die Tastaturbedienung ist dadurch nicht automatisch gegeben und muss von den EntwicklerInnen sichergestellt werden. Grundsätzlich liegt es in der Verantwortung des Entwicklungsteams, dafür zu sorgen, dass sich die Elemente so verhalten, wie sie im Accessibility Tree aufgelistet sind. Stimmt das Verhalten und die Rolle des Elements nicht überein, muss eines davon angepasst werden. Bei manchen Knoten soll zusätzlich ein zugänglicher Name vorhanden sein, der z. B. einen Button von dem anderen unterscheidet. Die textuelle Information zu einzelnen Knoten ist so wichtig, dass sie sogar ihre eigene Spezifikation von der ARIA-Initiative und einen dedizierten Abschnitt in diesem Buch bekommen hat.

Weiterführende Informationen

- »Accessibility Inspector« – Dokumentation von Firefox (*https://firefox-source-docs.mozilla.org/devtools-user/accessibility_inspector*)

3.3 Zugänglicher Name und Beschreibung

Betrachten Sie die Pfeile und Wörter in Abbildung 3.18 und versuchen Sie, laut zu sagen, in welche Richtung der Pfeil zeigt. An den Stellen, wo die Beschreibung nicht zu der eigentlichen Pfeilrichtung passt, ist die Aufgabe schwieriger und ein

40 *https://learn.microsoft.com/en-gb/windows/win32/winauto/inspect-objects*

41 *https://learn.microsoft.com/en-gb/windows/win32/winauto/accessible-event-watcher*

42 *https://developer.apple.com/documentation/accessibility/accessibility-inspector*

Fehler wahrscheinlicher. Der letzte Pfeil ist gar nicht sichtbar, also müssen Sie sich auf den Text verlassen. Ähnlich ist es, wenn interaktive Elemente im Web bzw. Knoten im Accessibility Tree nicht oder inkorrekt benannt werden.

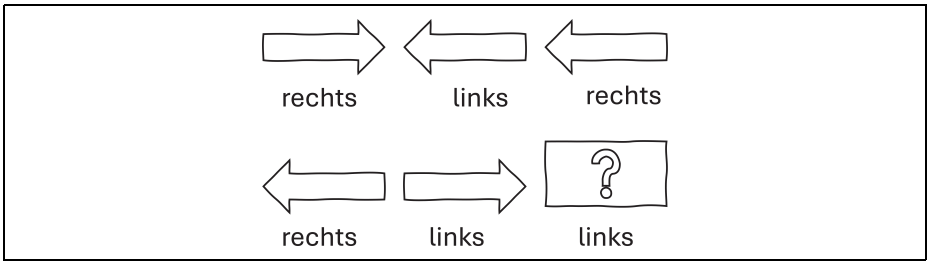


Abbildung 3.18: Textuelle und visuelle Informationen stimmen nicht überein und sorgen für Verwirrung.

3.3.1 Zugänglicher Name

Zugängliche Namen und Beschreibungen werden – falls vorhanden – vom Browser im Accessibility Tree bestimmt und von Screenreadern zusammen mit der dazugehörigen Rolle angekündigt. In Abbildung 3.19 sehen Sie einen Teil des Accessibility Trees mit einem Knoten mit der Rolle `"button"` und dem Namen »Anmelden«. Der Screenreader wird ihn als »Anmelden, Schaltfläche«[43] ankündigen, sobald das Element den Fokus bekommt. In den Entwicklertools können Sie nachschauen, woher der ermittelte zugängliche Name kommt. In Abbildung 3.19 ist z. B. der innere Button-Text »Anmelden« der Namensgeber.

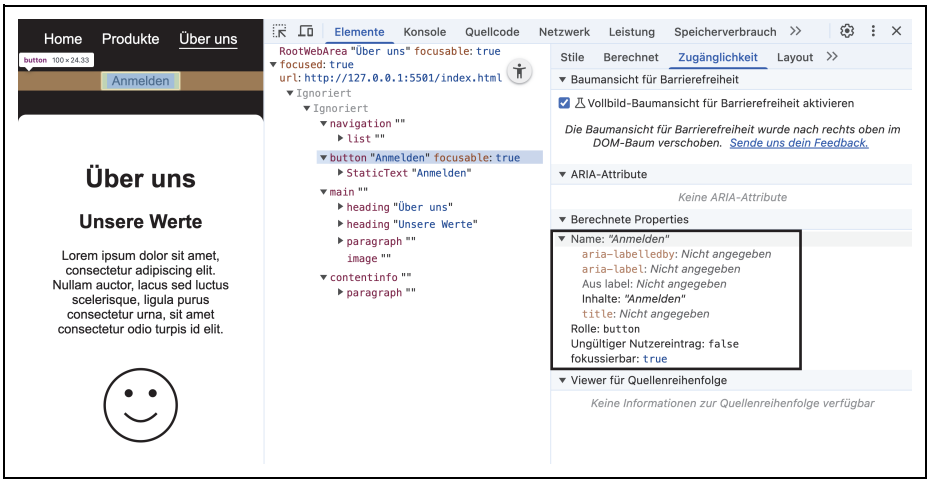


Abbildung 3.19: Der berechnete zugängliche Name im Accessibility Tree in den Chrome-Entwicklertools

43 Der genaue Text und die Reihenfolge der Informationen hängen von der Browser-Screenreader-Kombination und den Screenreader-Einstellungen ab.

Ein zugänglicher Name ist eine kurze Zeichenkette, typischerweise ein bis drei Wörter lang, die einem Element zugeordnet ist, um BenutzerInnen von assistiven Technologien eine Beschriftung für das Element bereitzustellen. Diese Beschriftung (engl. »label«) wird nicht nur durch das HTML-Element `<label>` gegeben, sondern es gibt mehrere Möglichkeiten dafür, die im Folgenden beschrieben werden. Die Beschriftung muss auch nicht zwingend mit dem sichtbaren Text des Elements übereinstimmen (leider!).

Zugängliche Namen unterscheiden das Element von den anderen und vermitteln seinen Zweck. Dies hilft BenutzerInnen zu verstehen, wofür das Element gedacht ist und wie sie damit interagieren können. NutzerInnen von Spracheingabe-Software können beispielsweise einen Button mithilfe des zugänglichen Namens (der Beschriftung) gezielt ansprechen und bedienen. Deswegen sollten zugängliche Namen für Elemente auf einer Seite eindeutig sein. Das WCAG-Erfolgskriterium *4.1.2 Name, Rolle, Wert (A)* verlangt für alle fokussierbaren, interaktiven Elemente (Links, Buttons, Formularelemente) einen zugänglichen Namen. Das gilt auch für Grafiken, eingebettete Inhalte oder Widgets (zum Beispiel modale Dialoge oder Schieberegler). Nicht interaktive Elemente brauchen in der Regel keinen zugänglichen Namen.

Die ARIA-Spezifikation definiert einen zusätzlichen Regelsatz und schreibt vor, welche Rollen einen zugänglichen Namen erfordern, welche ihn haben können und für welche er sogar verboten ist[44]. Diese Regeln entsprechen der Anforderung der WCAG und der allgemeinen Logik, dass nur interaktive Elemente (z. B. `role="link"`, `<a>`), Elemente, die sonst keine textuelle Information haben (z. B. `role="img"`, `<img>`) oder die selbst mehrere Inhalte umschließen (z. B. `role="table"`, `<table>`), einen zugänglichen Namen brauchen. Das Setzen eines zugänglichen Namens auf einem `<div>`-Element (`<div aria-label="Test">`) hat zum Beispiel unvorhersehbare Konsequenzen bei der Screenreader-Ankündigung. Doch wie kommt der zugängliche Name zustande?

Ich möchte damit anfangen, was alles einem Web-Element einen Namen verleihen kann. Je nach Element und HTML-Markup haben Sie verschiedene Möglichkeiten. Es ist am besten, **bereits existierenden sichtbaren Text** als zugänglichen Namen zu verwenden. Viele Elemente, z. B. `<a>`, `<td>` und `<button>`, können ihren zugänglichen Namen aus ihrem Inhalt beziehen. Zum Beispiel hat der Hyperlink `<a href="https://www.gesetze-im-internet.de/.html">Gesetze im Internet</a>` den zugänglichen Namen »Gesetze im Internet«. Diese Methode hat klare Vorteile: Der Name und der sichtbare Text stimmen überein und können logischerweise gleichzeitig in eine andere Sprache übersetzt werden.

44 *https://www.w3.org/TR/wai-aria-1.2/#namefromprohibited*

```
/* Zugänglicher Name: Anmelden */
<button>Anmelden</button>

/* Zugänglicher Name: Gesetze im Internet */
<a href="https:...">Gesetze im Internet</a>

/* Zugänglicher Name: 1000 EUR */
<td>1000 EUR</td>
```

Listing 3.16: Zugängliche Namen aus sichtbarem Textinhalt für Elemente wie `<button>`, `<a>` *und* `<td>`

Einige Elemente erhalten ihren zugänglichen Namen aus dem Inhalt zugeordneter Elemente. Das `<legend>`-Element innerhalb eines `<fieldset>`-Elements gibt ihm den Namen. Das Gleiche gilt für `<caption>` innerhalb `<table>`. Bei Formularelementen wie `<textarea>` und `<input>` stammt der zugängliche Name vom zugehörigen `<label>`-Element. Diese Verbindung wird explizit durch das `for`-Attribut im `<label>`-Element definiert, das mit der ID des Formularelements übereinstimmt. Alternativ wird eine implizite Verbindung hergestellt, wenn das Formularelement innerhalb des `<label>`-Elements liegt (mehr dazu in Abschnitt 4.10). Klickt man auf das `<label>`-Element, wird automatisch das Eingabefeld aktiviert. Dadurch wird die Klickfläche größer. Davon profitieren Menschen mit motorischen Einschränkungen und alle NutzerInnen bei viel zu kleinen Inputs.

```
/* Zugänglicher Name der Tabelle: BIP je nach Land */
<table>
  <caption>
    BIP je nach Land
  </caption>
  <tr>
    <th>Land</th>
    <th>BIP</th>
[...]

/* Zugänglicher Name des Eingabefelds: Weiblich */
<label for="geschlecht">Weiblich</label>
<input type="radio" name="frau" id="geschlecht" />
```

Listing 3.17: Zugängliche Namen durch zugeordnete Elemente: `<caption>` *für* `<table>` *und* `<label>` *für Eingabefelder*

Bei einigen Elementen stammt der zugängliche Name aus den Attributen des Elements, zum Beispiel dem `alt`-Attribut im Fall von `<img>` und `<input type="image">`. Bilder sind zwar nicht interaktiv, sollten aber dennoch textuelle Informationen (falls nicht dekorativ verwendet) für blinde und sehbehinderte NutzerInnen bieten. Das `title`-Attribut kann zwar ebenfalls einen zugänglichen Namen verleihen, wird dafür jedoch selten verwendet, da es vom `alt`-Attribut überstimmt wird und in diesem Fall als zugängliche Beschreibung fungiert. Der häufigste Anwendungsfall für das `title`-Attribut ist die Benennung von iframes. In diesem Fall ist der Name nützlich, damit NutzerInnen entscheiden können, ob sie in das iframe »reinspringen« möchten.

```
/* Zugänglicher Name: Apfel */
<img src="apfel.jpg" alt="Apfel"/>

/* Zugänglicher Name: Apfel */
<input type="image" src="apfel.jpg" alt="Apfel">

/* Zugänglicher Name: Wikipedia Hauptseite */
<iframe title="Wikipedia Hauptseite" src="https://de.wikipedia.org/wiki/Wikipedia:
Hauptseite"></iframe>
```

Listing 3.18: Zugängliche Namen aus Attributen: `alt` *für Bilder und* `title` *für iframes*

Die nativen Möglichkeiten der HTML-Elemente, ihre zugänglichen Namen zu bestimmen, sollten immer bevorzugt werden. So kann Browserunterstützung sichergestellt werden. Sollten diese Möglichkeiten nicht ausreichen, können Sie zu ARIA-Mitteln greifen. Häufig ist es der Fall, wenn Links und Buttons keinen Text, sondern Icons enthalten. Auch Landmarks (z. B. `<footer>`) haben keine nativen Möglichkeiten, sich selbst zu benennen. Sie sind zwar nicht interaktiv, brauchen aber ausnahmsweise ebenso einen zugänglichen Namen, um in der Übersicht aller Landmarks von Screenreader eindeutig identifiziert werden zu können.

Wenn die Elemente keinen eigenen Text haben, können Sie einen anderen Text der Webseite referenzieren. Das können Sie mit dem ARIA-Attribut `aria-labelledby` tun. Dafür muss der zu referenzierende Text mit einer ID versehen werden, die als Wert an `aria-labelledby` übergeben wird.

```
<p id="submit-text">Reichen Sie Ihren Antrag jetzt ein!</p>
[...]
/* Zugänglicher Name: Reichen Sie Ihren Antrag jetzt ein! */
<button aria-labelledby="submit-text">
  <img src="submit.png" alt="" />
</button>
```

Listing 3.19: Zugängliche Namen durch Referenzierung: `aria-labelledby` *verknüpft Elemente mit vorhandenen Texten über ihre ID.*

Die `aria-labelledby`-Eigenschaft nimmt als Wert eine durch Leerzeichen getrennte Liste von ID-Referenzen an. So können Sie mehrere Referenzen zu einem einzigen barrierefreien Namen kombinieren. Ein Element kann auf sich selbst verweisen, um seinen eigenen Inhalt als Teil des barrierefreien Namens zu nutzen. Das `aria-labelledby`-Attribut integriert den Wert von Eingabeelementen. Wenn der Wert eine `<input>`-Referenz enthält, wird der aktuelle Wert des Formularelements in den zugänglichen Namen aufgenommen und ändert sich, wenn der Wert aktualisiert wird. Die Reihenfolge der Werte in `aria-labelledby` ist wichtig. Wenn mehr als ein Element durch `aria-labelledby` referenziert wird, wird der Inhalt jedes referenzierten Elements in der Reihenfolge kombiniert, in der sie im Wert von `aria-labelledby` erscheinen. Bei wiederholten IDs wird die erste ausgewertet, alle weiteren Wiederholungen jedoch ignoriert.

```
<h3 id="links">Barrierefreie Links</h3>
[...]
/* Zugänglicher Name: Zum Artikel Barrierefreie Links */
<a href="/accessible-links" id="link-1"
   aria-labelledby="link-1 links">
   Zum Artikel
</a>

/* Zugänglicher Name wird je nach Eingabe dynamisch gesetzt*/
<label for="task-input">Enter a task:</label>
<input type="text" id="task-input"/>
<button type="button" id="dynamic-button" aria-labelledby="dynamic-button task-
input">Add task
</button>
```

Listing 3.20: Dynamische und kombinierte zugängliche Namen mit `aria-labelledby`

Mit `aria-labelledby` zeigt die ARIA-Spezifikation ihre Mächtigkeit. Dieses Attribut hat die höchste Priorität, wenn Browser zugängliche Namen bestimmen. Es überschreibt andere Methoden zur Benennung des Elements, sogar die, die aus den Inhalten des Elements abgeleitet werden.

```
/* Zugänglicher Name des Bildes: Banane */
<p id="fruit">Banane</p>
<img src="birne.jpg" alt="Apfel" aria-labelledby="fruit"/>

/* Zugänglicher Name des Buttons: Abmelden */
<p id="logout">Abmelden</p>
<button aria-labelledby="logout">Anmelden</button>
```

Listing 3.21: Die Macht von `aria-labelledby`*: Ein bestehender zugänglicher Name, der durch Inhalt oder Attribute entsteht, wird überschrieben.*

Das `aria-labelledby`-Attribut ist so mächtig, dass es sogar versteckte Inhalte, die sonst für Screenreader nicht verfügbar sind, ankündigen kann.[45] Sie erfahren mehr zum barrierefreien Verstecken im nächsten Abschnitt, hier aber ein kleiner Teaser:

```
/* Zugänglicher Name des Buttons: Unsichtbar */
<p id="invisible" style="display: none;">Unsichtbar</p>
<button aria-labelledby="invisible">
  <img src="icon.png" alt=""/>
</button>

/* Zugänglicher Name des Buttons: Auch unsichtbar */
<p id="invisible" aria-hidden="true">Auch unsichtbar</p>
<button aria-labelledby="invisible">
  <img src="icon.png" alt=""/>
</button>
```

Listing 3.22: Die Macht von `aria-labelledby`*: Nutzung versteckter Inhalte zur Benennung von Elementen*

45 *https://www.w3.org/TR/using-aria/#label-support*

Wenn man den Text nur visuell versteckt, aber im DOM belässt, sodass er weiterhin für den Screenreader verfügbar ist, können Sie ihn als Inhalt für interaktive Elemente verwenden. Dann brauchen Sie auch kein `aria-labelledby`-Attribut. Der Nachteil dieser Methode ist, dass NutzerInnen der Sprachsteuerung keinen sichtbaren Referenztext haben, um das UI-Element sofort zu identifizieren.

```
/* Zugänglicher Name: Unsichtbar */
<button>
  <p class="visually-hidden">Unsichtbar</p>
  <img src="icon.png" alt=""/>
</button>
```

Listing 3.23: Visuell versteckter Textinhalt des Buttons als zugänglicher Name

Wenn kein passender Text auf der Seite vorhanden ist, der mit dem UI-Element verknüpft werden kann, das einen zugänglichen Namen braucht, kann das Attribut `aria-label` verwendet werden. Im Unterschied zu `aria-labelledby` bietet dieses Attribut die Möglichkeit, eigene Texte zu definieren, die auf der Oberfläche nicht sichtbar sind. Das bringt die Gefahr, dass der Wert des `aria-label`-Attributs von der sichtbaren Beschriftung abweicht. Das UI-Element lässt sich dann nicht oder nur über Umwege mittels Spracheingabe aktivieren.

Manchmal wird `aria-label` verwendet, um sichtbare Beschriftungen zu erweitern. Dies geschieht meist in der Absicht, den zugänglichen Namen für nicht sehende Menschen aussagekräftiger zu machen, wie z. B. in Listing 3.24.

```
/* Zugänglicher Name: (öffnet sich im neuen Tab) */
<a aria-label="(öffnet sich im neuen Tab)" href="https:..." target="_blank">
Gesetze im Internet</a>
```

Listing 3.24: Vorsicht bei `aria-label`*: Sichtbare Beschriftungen können dadurch überschrieben werden.*

Damit wird jedoch vergessen, dass auch `aria-label` ein mächtiges Tool ist. Es hat zwar eine geringere Priorität als `aria-labelledby`, überschreibt jedoch ebenso den nativen Namen. Platziert man `aria-label` z. B. auf einem Link, wird der Linktext ignoriert. Im folgenden Beispiel wird statt »Gesetze im Internet (öffnet sich im neuen Tab)« lediglich »öffnet sich im neuen Tab« angekündigt. Der überschriebene Name ist nicht mehr hilfreich, automatische Tools können aber leider kein Problem in diesem Fall identifizieren. Dieser Anwendungsfall hat es sogar in die Sammlung von »unglücklichen Beispielen«[46] von Adrian Roselli geschafft, als Veranschaulichung von gut gemeintem Eifer, der zu mehr Barrieren führt. Stellen Sie deswegen sicher, dass die sichtbare Beschriftung im zugänglichen Namen enthalten ist, am besten am Anfang des Textes, so wie das WCAG-Erfolgskriterium *2.5.3 Beschriftung (Label) im Namen (A)* das empfiehlt.

Wenn der Link bereits einen Text enthält, sollten Sie also kein `aria-label` verwenden. Dazu kommt noch, dass der Text in `aria-label` leider nicht übersetzt wird –

46 *https://adrianroselli.com/2019/02/uncanny-a11y.html#ARIA-Label*

weder von der eingebauten Übersetzung im Browser noch von einer externen automatischen Übersetzung wie Google Translate. Für alle Texte, die nicht auf der Oberfläche erscheinen, gilt auch, dass sie vergessen werden können, wenn professionelle Übersetzung in Auftrag gegeben wird oder sich das Design ändert. Das kann auch deswegen passieren, weil sie nicht kopiert werden können. Leider kann man nach `aria-label`-Texten auf der Seite auch nicht suchen.

Basierend auf den Einschränkungen von `aria-label` soll dieses Attribut als Mittel der letzten Wahl verwendet werden, wenn nichts anderes mehr geht. Eric Baley geht sogar weiter und deklariert die Verwendung von `aria-label` als Code-Smell, also als Merkmale im Quellcode, die auf ein potenzielles Problem hinweisen. Bevor Sie also `aria-label` einsetzen, lesen Sie seinen Artikel »aria-label is a code smell«[47] dazu.

Manche HTML-Elemente bzw. ARIA-Rollen haben spezifische Möglichkeiten, den zugänglichen Namen zu definieren. Mit `aria-valuetext` können Sie z. B. ein verständliches Äquivalent für eine Spannweite (wie etwa beim `<meter>`-Element) angeben (z. B. »8 % (34 Minuten)« für den Akkustand), während `aria-valuenow` den eigentlichen Wert angibt. Bei Input-Feldern stehen Ihnen zusätzlich `placeholder` und `aria-placeholder` zur Verfügung. Diese Methoden sollten jedoch lieber nicht verwendet werden, da die oben erwähnten Ansätze deutlich besser sind. Mehr Informationen zum Umgang mit den Platzhaltern finden Sie in Abschnitt 4.10.

Download, downloading, downloaded

In manchen Szenarien will man den zugänglichen Namen dynamisch anpassen, z. B. bei einem Button zum Herunterladen von Unterlagen in Abbildung 3.20. Er soll nach dem Klicken ein Icon mit der Animation zum laufenden Prozess anzeigen und am Ende eine Statusmeldung ausgeben (Erfolg oder Fehler).

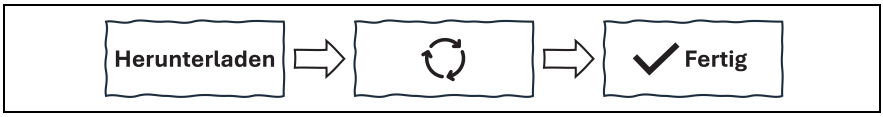


Abbildung 3.20: Ein Button, der drei Zustände abbildet: »Herunterladen«, Ladeindikator und Erfolgsmeldung

Adria Roselli merkt in seinem Artikel »Be Careful with Dynamic Accessible Names«[48] an, dass Screenreader den aktualisierten Namen (unabhängig von der verwendeten Methode) nicht immer ankündigen. Darif Senneff erweitert in seinem Artikel »Dynamic Accessible Descriptions Reference«[49] das Problem auch auf zugängliche Beschreibungen. Als Lösung bietet Adrian Roselli einen

47 *https://ericwbailey.website/published/aria-label-is-a-code-smell*

48 *https://adrianroselli.com/2020/12/be-careful-with-dynamic-accessible-names.html*

49 *https://www.darins.page/articles/dynamic-accessible-descriptions-reference*

Ansatz von Live-Regionen (siehe Abschnitt 3.5), die für die Ankündigung der Veränderungen besser geeignet sind. Mehr Details finden Sie in seinem Artikel »Multi-Function Button«[50]. Wenn Ihr Button jedoch nur zwei Zustände haben soll, reicht es, wenn Sie ihn als einen Toggle mit `aria-pressed` implementieren. Mehr dazu finden Sie in Abschnitt 4.6.

3.3.2 Zugängliche Beschreibung

Eine zugängliche Beschreibung ist die Beschreibung eines UI-Elements, die **zusätzliche** Informationen liefert, die über den barrierefreien Namen hinausgehen, um BenutzerInnen von assistiven Technologien zu helfen, das Element sowie seinen Zweck und Kontext zu verstehen. Eine zugängliche Beschreibung ist Teil des Accessibility Trees (siehe Abbildung 3.21).

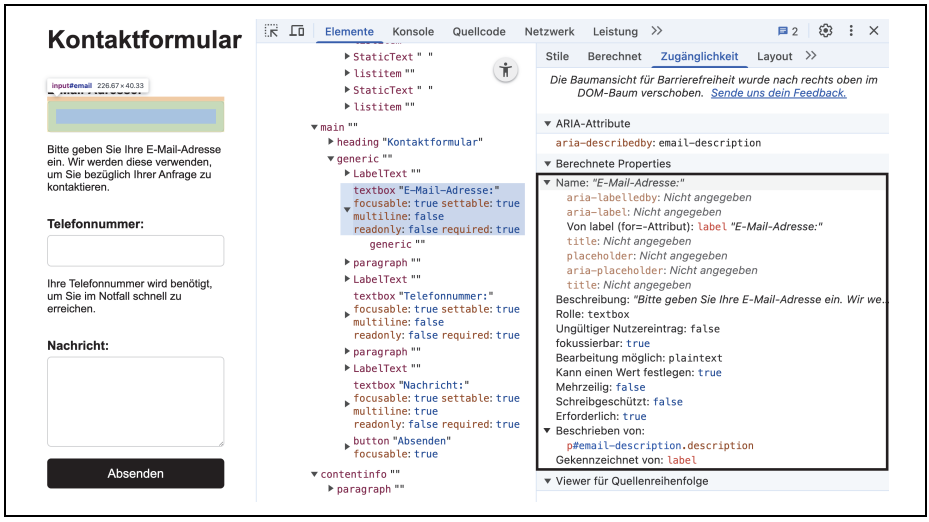


Abbildung 3.21: Das Eingabefeld »E-Mail-Adresse« ist so programmiert, dass der Text darunter als zusätzliche zugängliche Beschreibung genutzt wird.

Da Beschreibungen optional und normalerweise deutlich länger als Namen sind, werden sie von Screenreadern zuletzt angekündigt, manchmal nach einer kurzen Verzögerung. Zum Beispiel: »E-Mail-Adresse: Bitte geben Sie Ihre E-Mail-Adresse ein. Wir werden diese verwenden, um Sie bezüglich Ihrer Anfrage zu kontaktieren. Erforderlich. Einblendmenü«[51]. Je nach Einstellung kündigen einige Screenreader Beschreibungen nicht automatisch an, sondern informieren BenutzerInnen über deren Vorhandensein, sodass diese eine Taste drücken können, um sich die Beschreibung anzuhören.

50 *https://adrianroselli.com/2021/01/multi-function-button.html*

51 Der Punkt nach »Erforderlich« dient besserer Lesbarkeit. Die tatsächliche Screenreader-Ausgabe enthält keine zusätzlichen Satzzeichen.

Auch für die zugängliche Beschreibung gibt es einige native Methoden (je nach HTML/SVG-Element). Zum Beispiel kann die zugängliche Beschreibung einer `<table>` durch ihre erste `<caption>` bereitgestellt werden, wenn diese nicht schon für den zugänglichen Namen verwendet wird.

```
/* Zugänglicher Name: BIP pro Land */
/* Zugängliche Beschreibung: Liste der Länder nach Bruttoinlandsprodukt,
alphabetisch sortiert */
<h2 id="bip-title">BIP pro Land</h2>
<table aria-labelledby="bip-title">
  <caption>
    Liste der Länder nach Bruttoinlandsprodukt, alphabetisch sortiert
  </caption>
[...]
```

Listing 3.25: Zugängliche Beschreibung: Nutzung von `<caption>` *für zusätzliche Kontextinformationen bei Tabellen, die bereits einen zugänglichen Namen haben*

Ein `<summary>` wird durch den Inhalt des `<details>` beschrieben, in dem es verschachtelt ist. `<input>`-Buttons (mit dem `type`-Attribut `"button"`, `"submit"` oder `"reset"`) werden durch den Wert ihres `value`-Attributs beschrieben. In SVG wird der Inhalt des `<desc>`-Elements (falls vorhanden) genommen, andernfalls der Text, der in nachfolgenden Textcontainern (d. h. `<text>`) enthalten ist, sofern diese nicht bereits für den zugänglichen Namen verwendet werden. Wenn das Element keine weiteren Mittel für die Beschreibung hat, wird das `title`-Attribut verwendet, sofern der `title` nicht der zugängliche Name für dieses Element ist.

Die ARIA-Spezifikation bietet ebenso Attribute für eine zugängliche Beschreibung: `aria-description` und `aria-describedby`. Diese stehen in derselben Relation zueinander wie `aria-label` und `aria-labelledby`. Das `aria-describedby`-Attribut ist dem `aria-labelledby`-Attribut sehr ähnlich. Während `aria-labelledby` die IDs der Elemente auflistet, die den zugänglichen Namen beschreiben, listet `aria-describedby` die IDs der Beschreibungen auf. Beide Attribute verweisen auf andere Elemente, um eine Textalternative zu berechnen. Es ist möglich, mit `aria-describedby` ein Element zu referenzieren, auch wenn dieses Element ausgeblendet ist. Im nachstehenden Beispiel kommt der Text im `<p>`-Element nach dem Button. Dieser Hinweis könnte von Screenreader-NutzerInnen verpasst werden, deswegen ist es hilfreich, in der Button-Beschreibung darauf hinzuweisen. Das Gleiche gilt für Anforderungen für ein neues Passwort, Hinweise darauf, dass Änderungen nicht rückgängig gemacht werden können usw.

```
/* Zugänglicher Name: In den Papierkorb verschieben */
/* Zugängliche Beschreibung: Elemente im Papierkorb werden nach 30 Tagen endgültig
entfernt. */
<button type="button" aria-describedby="trash-desc">In den Papierkorb
verschieben</button>
<p id="trash-desc">Elemente im Papierkorb werden nach 30 Tagen endgültig
entfernt.</p>
```

Listing 3.26: Zugängliche Beschreibung mit `aria-describedby`

Analog zu `aria-label` bietet `aria-description`-Attribut eine Möglichkeit, den Beschreibungstext selbst zu definieren. Da dieser Text nicht sichtbar ist, gelten für ihn alle Nachteile der `aria-label`-Methode. Sie werden ebenso nicht automatisch übersetzt. Darüber hinaus wird `aria-description` noch nicht konsistent unterstützt.

```
/* Zugänglicher Name: In den Papierkorb verschieben */
/* Zugängliche Beschreibung: Elemente im Papierkorb werden nach 30 Tagen endgültig
entfernt. */
<button aria-description="Elemente im Papierkorb werden nach 30 Tagen endgültig
entfernt.">
  In den Papierkorb verschieben
</button>
```

Listing 3.27: Zugängliche Beschreibung mit `aria-description`

3.3.3 Berechnung: Viele Wege, ein Rom

Auch wenn viele Wege bekanntlich nach Rom führen, gibt es Rom selbst nur einmal. Bei so vielen verschiedenen Methoden, einen zugänglichen Namen bzw. eine zugängliche Beschreibung hinzuzufügen, wer entscheidet, wie der finale Name lautet? Um das Problem zu verdeutlichen, konstruiere ich ein unwahrscheinliches Beispiel, in dem möglichst viele verschiedene Namenskandidaten vorhanden sind.

```
/* Zugänglicher Name: ??? */
<p id="p-name">Name 1</p>
<a aria-labelledby="p-name" aria-label="Name 2" title="Name 3" href="https:...">
Name 4</a>
```

Listing 3.28: Prioritätsreihenfolge für zugängliche Namen: Welcher Name wird verwendet, wenn mehrere Kandidaten existieren?

Zuerst zu der Frage, wer den finalen Namen auflösen darf: der Browser. Dieser muss schließlich den Namen an den Accessibility Tree übergeben. Wie wird der Name bestimmt? Dafür hat der W3C die Spezifikation »Accessible Name and Description Computation 1.1«[52] veröffentlicht. Sie definiert relevante Begriffe und gibt einen Algorithmus im Pseudocode vor, wie zugängliche Namen und Beschreibungen zu berechnen sind.

Vereinfacht lautet der Prozess wie folgt: Die Berechnung prüft die Bedingungen von oben nach unten und steigt vorzeitig aus, sobald eine erfüllt ist. Daraus ergibt sich automatisch die Hierarchie der namensgebenden Methoden.

1. Ist das aktuelle Element versteckt und nicht durch `aria-labelledby` oder `aria-describedby` referenziert, ist das Ergebnis ein leerer String.
2. Hat das aktuelle Element ein `aria-labelledby`-Attribut (für den zugänglichen Namen) oder ein `aria-describedby`-Attribut (für die zugängliche Beschreibung),

52 *https://www.w3.org/TR/accname-1.1*; Version 1.2 hat noch den Status eines Arbeitsentwurfs: *https://www.w3.org/TR/accname-1.2*

Inhalt

O'REILLY®

Barrierefreie Webentwicklung

Spätestens zum 28. Juni 2025 muss das Barrierefreiheitsstärkungsgesetz angewendet werden. Dies bedeutet nicht nur eine gesetzliche Verpflichtung für viele Unternehmen, sondern bietet auch Vorteile: Barrierefreie Websites erreichen eine größere Zielgruppe, sorgen für eine bessere UX und stärken das Markenimage.

Dieses praxisorientierte Buch ist essenziell für alle, die sich mit der Erstellung barrierefreier Webinhalte befassen. Maria Korneeva vermittelt die Grundlagen der digitalen Barrierefreiheit und die relevanten rechtlichen Rahmenbedingungen und Richtlinien, darunter das Barrierefreiheitsstärkungsgesetz (BFSG), die Web Content Accessibility Guidelines (WCAG) und EN 301 549. Anhand von anschaulichen Beispielen lernen Sie, Barrieren im Web selbst zu erleben und zu verstehen. Zudem erhalten Sie verständliche Anleitungen und praxisnahe Codebeispiele, die Ihnen helfen, Barrierefreiheit erfolgreich in Ihre Projekte zu integrieren. Automatisierte Tests unterstützen Sie schließlich dabei, die langfristige Einhaltung der Barrierefreiheitsanforderungen sicherzustellen.

Aus dem Inhalt:

- Bedeutung und Vorteile digitaler Barrierefreiheit
- Gesetzliche Vorgaben, Normen und Richtlinien (inkl. BFSG, EAA und WCAG)
- Umgang mit Screenreadern
- Grundsätze der Barrierefreiheit: Responsiveness, Farbgestaltung, Tastaturbedienbarkeit, Alternativtexte
- Semantisches HTML, ARIA, Accessibility Tree, zugänglicher Name und Beschreibung
- Barrierefreiheit von Web Components und Single Page Applications
- Tools für automatisierte Barrierefreiheitschecks
- Linters und Plug-ins für Unit- und End-to-End-Tests
- Integration in CI/CD-Pipelines
- Künstliche Intelligenz, zukunftsweisende Entwicklungen und aktuelle Einschränkungen
- Barrierefreie Implementierung von Navigation, Links, Buttons, Bildern, Cookie-Bannern, Tabellen, Formularen und weiteren UI-Elementen

Maria Korneeva ist Frontend Technology Lead und Google Developer Expert mit Fokus auf Angular und Barrierefreiheit. Sie arbeitet freiberuflich an der Entwicklung von Frontend-Anwendungen und leitet Workshops zu Web-Technologien. Ihre Erkenntnisse aus dem Coding-Alltag teilt sie gerne auf Konferenzen und Meetups.

9 783960 092537

www.dpunkt.de

Euro 39,90 (D)
ISBN 978-3-96009-253-7

Gedruckt in Deutschland
Mineralölfreie Druckfarben
Zertifiziertes Papier