SOP 38

Pagebuilder

Projektleitung | Geschäftsführung | IT | Recht

So schließt du die Hintertüren in deiner barrierefreien Website

Weil Pagebuilder (wie Elementor, Divi) und Frontend-Frameworks (wie React) zwar die schnelle Erstellung von Websites ermöglichen, aber oft eine „Black Box“ sind. Sie erzeugen im Hintergrund Code, der von Haus aus häufig nicht barrierefrei ist: Sie nutzen -Elemente anstelle von semantisch korrekten <button>-Elementen, bringen die Überschriften-Hierarchie durcheinander oder machen es schwer, ARIA-Attribute für komplexe Komponenten zu ergänzen.

Als Betreiber:in der Website bist du für das Endergebnis verantwortlich, egal, welches Werkzeug du benutzt. Diese SOP gibt dir Strategien, um die Risiken dieser Tools zu minimieren und sie für deine Zwecke zu zähmen.

Wann musst du das machen?

- Bevor du dich für den Einsatz eines Pagebuilders oder eines bestimmten Frameworks entscheidest.
- Während der gesamten Entwicklung und Gestaltung einer Website mit diesen Werkzeugen.
- Bei jedem Audit einer Website, die mit einem solchen Tool erstellt wurde.

Welches Gesetz verlangt das?

EAA / BFSG: Das Gesetz bewertet das Endprodukt – deine Website oder App –, nicht das Werkzeug, mit dem sie gebaut wurde. Die Verantwortung, die WCAG-Standards zu erfüllen, liegt immer bei dir als Betreiber:in.

Der Prozess im Detail

1 🟧 Phase 1: Die goldene Regel – Nutze die richtigen Module

Pagebuilder bieten eine riesige Auswahl an Modulen, aber nicht alle sind gleich gut.

- Semantik vor Optik: Nutze das „Überschriften“-Modul, um eine <h2> zu erzeugen, nicht das „Animierte Überschrift“-Modul, das nur einen -Container ausgibt. Nutze das „Button“-Modul, nicht eine „Klickbare Box“.
- Grundlagen bevorzugen: Oft sind die einfachen Module (Text-Editor, Bild, Button) besser anpassbar und erzeugen saubereren Code als hochkomplexe All-in-One-Module.

2 🟨 Phase 2: Die Überschriften-Hierarchie aktiv managen

Dies ist eine der größten Fehlerquellen in Pagebuildern.

- Eine H1 pro Seite: Jede Seite darf nur eine einzige <h1> haben.
- Logische Reihenfolge: Stelle sicher, dass die Hierarchie (H1 → H2 → H3) eingehalten wird.
- HTML-Tag prüfen: In den Einstellungen fast jedes Text- oder Überschriften-Moduls kannst du den „HTML-Tag“ manuell auswählen. Prüfe und setze hier aktiv das korrekte Tag (H2, H3, P etc.).

3 🟩 Phase 3: Vorsicht bei komplexen Modulen (Tabs, Slider, Akkordeons)

Die eingebauten Module für komplexe Komponenten sind oft nicht barrierefrei.

- Teste sie rigoros: Prüfe die Tastaturbedienung und das Verhalten mit einem Screenreader. Kannst du alle Tabs mit der Tastatur erreichen? Wird der Zustand (offen/geschlossen) eines Akkordeons angesagt?
- Alternative suchen: Wenn ein Modul den Test nicht besteht, nutze es nicht. Suche stattdessen nach einem spezialisierten, barrierefreien Plugin eines Drittanbieters für diese Funktion oder lasse sie von Entwickler:innen individuell und barrierefrei programmieren.

4 🟥 Phase 4: Barrierefreie Komponenten-Bibliotheken wählen

Das Rad nicht neu erfinden.

- Accessibility-First: Bevorzuge UI-Bibliotheken, die Barrierefreiheit als Kernfunktion bewerben und dokumentieren. Gute Beispiele sind Chakra UI oder Reach UI .
- Dokumentation prüfen: Prüfe in der Dokumentation der Bibliothek, ob sie Hinweise zur Tastaturbedienung und ARIA-Implementierung gibt. Wenn das Thema Barrierefreiheit dort gar nicht vorkommt, ist das ein Warnsignal.

5 🟦 Phase 5: Semantisches HTML schreiben, auch im Framework

Frameworks verleiten dazu, alles aus s und s zu bauen. Widerstehe dieser Versuchung.

- JSX ist HTML: Nutze die korrekten HTML-Elemente in deinem Code. Ein Button ist <button onClick={…}>, kein . Eine Navigation ist <nav>, kein .
- Fragmente nutzen: Um unnötige -Container zu vermeiden, nutze die <>-Syntax (Fragmente) in React.

6 🟪 Phase 6: Fokus-Management in Single-Page-Anwendungen (SPAs)

Dies ist eine der größten Herausforderungen bei SPAs. Wenn die URL sich ändert, aber die Seite nicht neu lädt, bleibt der Fokus oft auf dem angeklickten Link zurück.

- Fokus manuell setzen: Nach jedem Routen-Wechsel muss dein Code den Fokus aktiv auf den neuen Inhaltsbereich setzen, idealerweise auf die neue <h1>-Überschrift der Seite. Dies signalisiert Screenreader-Nutzer:innen, dass sich der Seiteninhalt geändert hat.
- Seitentitel aktualisieren: Der <title> des Dokuments ändert sich bei SPAs nicht automatisch. Implementiere eine Lösung (z. B. mit react-helmet), um den Seitentitel bei jedem Routen-Wechsel dynamisch zu aktualisieren. Dies ist entscheidend für die Orientierung.

Ergebnis

Am Ende dieser SOP hast du:

- Die Fähigkeit, moderne und mächtige Werkzeuge zu nutzen, ohne die Barrierefreiheit zu opfern.
- Eine klare Strategie zur Bewertung, Konfiguration und Ergänzung dieser Tools, um konforme Websites zu erstellen.
- Weniger „Black Box“-Probleme, bei denen das Werkzeug einen unzugänglichen Output diktiert.
- Die Kontrolle über deinen Code und deine Barrierefreiheit zurückgewonnen.

Und hier noch ein Textbaustein, der dir helfen soll, wenn eine Agentur oder ein Entwickler deinen Online Shop oder deine Website für dich erstellen.

Betreff: Anforderungen an den Einsatz von Pagebuildern / Frameworks für unser Projekt

Hallo Team,

für die technische Umsetzung unseres neuen Webprojekts gelten folgende Anforderungen an die verwendeten Werkzeuge:

- Semantischer Output: Das gewählte Tool (Pagebuilder oder Framework) muss die Erzeugung von semantisch korrektem HTML (korrekte Überschriften-Hierarchie, nav, main etc.) ermöglichen.
- Manuelle Kontrolle: Es muss möglich sein, HTML-Tags und Attribute (wie aria-label) bei Bedarf manuell anzupassen.
- Komplexe Komponenten: Eingebaute Module für Slider, Tabs oder Akkordeons müssen entweder nachweislich barrierefrei sein oder durch eine zugängliche Alternative (z. B. eine externe Bibliothek oder eine Eigenentwicklung) ersetzt werden.
- Bei SPAs: Ein robustes Fokus-Management und die dynamische Aktualisierung des Seitentitels bei Routen-Wechseln müssen implementiert werden.

Bitte berücksichtigt diese Punkte bei der Auswahl der technologischen Basis und während der gesamten Entwicklung.

Danke!

Viele Grüße [Dein Name]

Fragen & Antworten

Ich nutze einen Pagebuilder. Was ist der einfachste Weg, um zu prüfen, welche Module semantisch korrekt sind und welche nur </p> s ausgeben?

Der einfachste Weg, das zu prüfen, du, ist der Browser-Inspektor (Rechtsklick auf das Element → „Untersuchen“ oder „Element untersuchen“).

- Wähle das Modul auf deiner Seite aus: Klick es an und schau dir den generierten HTML-Code an.
- Prüfe den Tag: Wenn du ein „Überschriften“-Modul verwendest, sollte dort <h1 , <h2 , etc. stehen. Siehst du stattdessen <div oder mit Inline-Styles, ist das ein Warnsignal, dass es nicht semantisch ist.
- Teste Button vs. Klickbare Box: Ein echtes Button-Modul sollte einen <button> -Tag generieren. Eine „Klickbare Box“ ist oft nur ein oder mit einem onClick -Handler, was Tastaturnutzende ausschließt.

Das erfordert ein wenig technisches Verständnis, aber es ist der direkteste Weg zur Wahrheit über die Semantik deiner Module.

Meine Designer bestehen auf einem speziellen %22Custom Module%22 für Animationen oder komplexe Layouts, das vom Pagebuilder nicht nativ unterstützt wird. Wie sorge ich hier für Barrierefreiheit?

Bei Custom Modules, die außerhalb der Standard-Pagebuilder-Module liegen, liegt die volle Verantwortung für die Barrierefreiheit bei dir und deinem Entwicklungsteam.

- Definiere klare Anforderungen: Gib den Entwicklern, die dieses Custom Module bauen, eine klare Spezifikation mit allen relevanten SOPs (z.B. für Animationen, Modale, Fokus-Management, Semantik). Es sollte von Anfang an „Accessibility by Design“ umgesetzt werden.
- Regelmäßiges Testing: Baue von Anfang an Tastatur- und Screenreader-Tests in den Entwicklungszyklus ein. Warte nicht bis zum Ende.
- Code-Reviews: Implementiert Code-Reviews, die einen Fokus auf die Einhaltung von Barrierefreiheitsstandards legen.
- Dokumentation: Bestehe auf einer detaillierten Dokumentation der Barrierefreiheitseigenschaften des Custom Modules, damit es von anderen Teams korrekt genutzt und gewartet werden kann.

Ein Custom Module bietet mehr Flexibilität, birgt aber auch ein höheres Risiko für Barrierefreiheitslücken, wenn es nicht sorgfältig umgesetzt wird.

Was ist, wenn mein Pagebuilder es mir erlaubt, mehrere H1-Module auf einer Seite zu platzieren? Wie gehe ich damit um?

Das ist ein sehr häufiges Problem bei Pagebuildern, du! Die Möglichkeit, mehrere H1s zu platzieren, ist ein Designfehler des Builders. Deine Aufgabe ist es, diese Funktion nicht zu nutzen und diszipliniert zu bleiben:

- Klares Kommunikations-Briefing: Informiere alle Content-Ersteller und Redakteure über diese Regel. H1 ist für den Haupttitel der Seite (z.B. der Titel des Blogposts oder der Produktseite) und sollte nur einmal vorkommen.
- HTML-Tag-Management: Nutze die Funktion, den HTML-Tag manuell einzustellen (wie in SOP Phase 2 beschrieben), um sicherzustellen, dass nur der eigentliche Seitentitel ein H1 ist und alle anderen „großen“ Überschriften H2, H3, etc. sind.
- Audit-Prozess: Implementiere einen regelmäßigen Audit-Prozess (z.B. mit Tools wie WAVE oder axe DevTools), der Seiten auf multiple H1s prüft. So kannst du Fehler identifizieren und korrigieren.

Ein einziger, klar definierter H1 ist entscheidend für die Navigation und das Verständnis der Seitenstruktur durch Screenreader.

Ich verwende ein JavaScript-Framework (z.B. React, Angular, Vue). Wie genau helfen mir %22Fragmente%22 dabei, unnötige </p> -Container zu vermeiden, und warum ist das wichtig für die Barrierefreiheit?

Das ist ein exzellenter Punkt! In Frameworks wie React können Komponenten oft zusätzliche -Wrapper um ihre Inhalte legen, die eigentlich nicht benötigt werden und das HTML unnötig aufblähen.

- Fragmente (<>...</> oder <React.Fragment>...</React.Fragment>): Erlauben es dir, mehrere Elemente in einer Komponente zurückzugeben, ohne einen zusätzlichen DOM-Knoten (also einen) zu erstellen.
- Wichtigkeit für Barrierefreiheit:
- Weniger unnötige DOM-Elemente: Ein „saubererer“ DOM-Baum ist leichter zu verarbeiten für Browser und Screenreader. Jeder unnötige kann theoretisch die Lesereihenfolge stören oder die Struktur unübersichtlicher machen.
- Keine Brechung von semantischen Strukturen: Stell dir vor, du hast eine Tabelle <table> und möchtest Zeilen (<tr>) in einer Unterkomponente rendern. Wenn diese Unterkomponente einen als Wrapper hätte, würde sie die <tr> s direkt in das <table> rendern und damit die semantische Struktur von HTML-Tabellen brechen (ein <tr> darf nur direkt im <table> oder <tbody> sein). Fragmente verhindern dies, indem sie den -Wrapper entfernen und die <tr> s direkt rendern. Fragmente sind also ein kleines, aber feines Detail, das zu einem saubereren, semantisch korrekteren und damit barrierefreieren Code beiträgt.

Meine SPA hat viele dynamische Inhalte, die sich beim Routen-Wechsel ändern. Reicht es, den Fokus auf die neue </p> <h1> zu setzen, oder muss ich auch ARIA Live Regions für andere dynamische Änderungen nutzen?

Den Fokus auf die neue <h1> zu setzen, ist ein hervorragender Startpunkt für die Orientierung bei einem Routen-Wechsel, du! Es signalisiert dem Nutzer, dass er auf einer „neuen Seite“ gelandet ist.

- Fokus für Hauptinhaltswechsel: Der Fokus auf die <h1> ist ideal, um den Startpunkt des neuen Inhalts zu markieren.
- ARIA Live Regions für Statusänderungen: Wenn innerhalb der neuen Seite weitere dynamische Inhalte geladen oder aktualisiert werden, die nicht den Hauptinhalt betreffen (z.B. eine Erfolgsmeldung nach einem Formularversand, ein Lade-Spinner, der plötzlich erscheint, oder eine Filterung, die die angezeigte Liste ändert), solltest du weiterhin ARIA Live Regions (aria-live="polite" oder role="status") nutzen. Diese stellen sicher, dass Screenreader diese kurzlebigen oder asynchronen Statusänderungen vorlesen, ohne den Fokus zu verschieben.

Es ist also ein Zusammenspiel: Der Fokus für den „großen“ Wechsel der Seite, Live Regions für die „kleineren“ Statusänderungen innerhalb der Seite.