アクセシビリティ

なぜアクセシビリティが必要なのか？

Web アクセシビリティ（**a11y** とも呼ばれます）とは、誰にでも使えるようウェブサイトを設計・構築することです。ユーザ補助技術がウェブページを解釈できるようにするためには、サイトでアクセシビリティをサポートする必要があります。
React はアクセシビリティを備えたウェブサイトの構築を全面的にサポートしており、大抵は標準の HTML の技術が用いられます。

標準およびガイドライン

WCAG

Web Content Accessibility Guidelines はアクセシビリティを備えたウェブサイトを構築するためのガイドラインを提供しています。
以下の WCAG のチェックリストはその概要を示します。

- WCAG checklist from Wuhcag
- WCAG checklist from WebAIM
- Checklist from The A11Y Project

WAI-ARIA

Web Accessibility Initiative - Accessible Rich Internet Applications には十分なアクセシビリティを持つ JavaScript ウィジェットの構築テクニックが含まれています。
補足として、JSX ではすべての `aria-*` で始まる HTML 属性がサポートされています。React においてほとんどの DOM プロパティと属性がキャメルケースである一方で、これらの属性は純粋な HTML と同じようにハイフンケース（ケバブケースやリスプケースなどとも言われる）である必要があります。

```
<input
  type="text"
  aria-label={labelText}
  aria-required="true"
  onChange={onchangeHandler}
  value={inputValue}
  name="name"
/>
```

セマンティックな HTML

セマンティック（意味論的）な HTML はウェブアプリケーションにおけるアクセシビリティの基礎となります。ウェブサイト内の情報の意味を明確にするための多様な HTML 要素を使うことにより、大抵の場合は少ない手間でアクセシビリティを手に入れられます。

- MDN HTML 要素リファレンス

ときおり、React コードを動くようにするために JSX に `<div>` を追加すると、HTML のセマンティックが崩れることがあります。とりわけ、リスト (`<ol>`, `<ul>`, `<dl>`) や `<table>` タグと組み合わせるときに問題になります。そんなときは複数の要素をグループ化するために React フラグメントを使う方がよいでしょう。
具体例です。

```
import React, { Fragment } from 'react';

function ListItem({ item }) {
  return (
    <Fragment>
      <dt>{item.term}</dt>
      <dd>{item.description}</dd>
    </Fragment>
  );
}

function Glossary(props) {
  return (
    <dl>
      {props.items.map(item => (
        <ListItem item={item} key={item.id} />
      ))}
    </dl>
  );
}
```

項目の集合をフラグメントの配列に変換することができますし、他の任意の要素でも同様です。

```
function Glossary(props) {
  return (
    <dl>
      {props.items.map(item => (
        // Fragments should also have a `key` prop when mapping collections
        <Fragment key={item.id}>
          <dt>{item.term}</dt>
          <dd>{item.description}</dd>
        </Fragment>
      ))}
    </dl>
  );
}
```

もし、フラグメントタグに props を渡す必要がなく、かつ使用しているツールがサポートしているのであれば、省略記法が使えます。

```
function ListItem({ item }) {
  return (
    <>
      <dt>{item.term}</dt>
      <dd>{item.description}</dd>
    </>
  );
}
```

より詳細な情報はフラグメントドキュメントにあります。

アクセシブルなフォーム

ラベル付け

`<input>` や `<textarea>` のような各 HTML フォームコントロールには、アクセシブルな形でのラベル付けが必要です。スクリーンリーダに公開される、説明的なラベルを提供する必要があります。以下の資料にはその方法が示されています：

- W3C による要素にラベルを付ける方法の解説
- WebAIM による要素にラベルを付ける方法の解説
- The Paciello Group によるアクセシブルな名前についての解説

React でこれらの標準的な HTML の実践知識を直接使用できますが、JSX では `for` 属性は `htmlFor` として記述されることに注意してください。

```
<label htmlFor="namedInput">Name:</label>
<input id="namedInput" type="text" name="name"/>
```

ユーザーへのエラー通知

すべてのユーザがエラーの起きた状況を理解できる必要があります。以下のリンクはどのようにエラーテキストをユーザーと同じくスクリーンリーダにも公開するかを解説しています。

- W3C によるユーザーへの通知方法の例示
- WebAIM によるフォームバリデーションの解説

フォーカス制御

あなたのウェブアプリケーションが完全にキーボードだけで操作できることを確かめてください：

- WebAIM によるキーボードアクセシビリティの解説

キーボードフォーカスとフォーカス時のアウトライン（輪郭線）

キーボードフォーカスは DOM の中でキーボードからの入力を受け付けるために選択されている要素を示します。フォーカスを輪郭線で示した以下の画像のような例を、様々な場所で見かけることができます：

例えば `outline: 0` のようにしてこのアウトラインを CSS で削除できますが、これは他の実装でフォーカス線を置き換える場合にのみ行うようにしてください。

目的のコンテンツまで飛べる仕組み

キーボードによる操作を補助して高速化するために、あなたのアプリケーションのナビゲーション（メニューや目次）部分をユーザが読み飛ばせるような仕組みを提供しましょう。
スキップリンクやスキップナビゲーションリンクとは、ユーザがキーボードでページを操作する場合にのみ出現する、隠れたナビゲーションリンクです。これらのリンクはページ内アンカーといくらかのスタイルを用いて、とても簡単に実装できます：

- WebAIM - Skip Navigation Links

`<main>` や `<aside>` のようなランドマーク要素とロール属性も活用してページの領域を区切り、補助技術を使うユーザが素早くこれらのセクションに移動できるようにしてください。

アクセシビリティを強化する、これらの要素の使い方についての詳細は以下を読んでください：

- Accessible Landmarks

プログラムによりフォーカスを管理する
React アプリケーションは実行されている間、継続的に HTML の DOM を変更するため、時にキーボードフォーカスが失われたり、予期しない要素にセットされたりすることがあります。これを修正するためには、プログラムによってキーボードフォーカスを正しい位置に移動させる必要があります。例えばモーダルウィンドウを閉じた後には、モーダルを開いたボタンにキーボードフォーカスを戻すことなどです。
MDN のウェブドキュメントには、キーボードで移動可能な JavaScript ウィジェット の作り方が解説されています。
React でフォーカスをセットするには、DOM 要素への Ref が使えます。
これを使って、まずコンポーネントクラスの JSX 要素に ref を作成します：

```
class CustomTextInput extends React.Component {
  constructor(props) {
    super(props);
    // Create a ref to store the textInput DOM element
    this.textInput = React.createRef();
  }
  render() {
  // Use the `ref` callback to store a reference to the text input DOM
  // element in an instance field (for example, this.textInput).
    return (
      <input
        type="text"
        ref={this.textInput}
      />
    );
  }
}
```

これで必要な場合にはコンポーネントのほかの場所からその要素にフォーカスすることができます。

```
focus() {
  // Explicitly focus the text input using the raw DOM API
  // Note: we're accessing "current" to get the DOM node
  this.textInput.current.focus();
}
```

ときおり、親コンポーネントは子コンポーネント内にフォーカスをセットする必要があります。これは、子コンポーネントの特定のプロパティを通して親コンポーネントに DOM の ref をさらし 、親の ref を子の DOM ノードに伝播させることで可能になります。

```
function CustomTextInput(props) {
  return (
    <div>
      <input ref={props.inputRef} />
    </div>
  );
}

class Parent extends React.Component {
  constructor(props) {
    super(props);
    this.inputElement = React.createRef();
  }
  render() {
    return (
      <CustomTextInput inputRef={this.inputElement} />
    );
  }
}

// Now you can set focus when required.
this.inputElement.current.focus();
```

コンポーネントを拡張するのに高階コンポーネント (HOC) を使う場合は、React の `forwardRef` 関数を用いて、関数に囲われたコンポーネントに ref を差し出す ことをおすすめします。もし、サードパーティの高階コンポーネントが ref フォワーディングを実装していないときでも、上記のパターンはフォールバックとして使えます。
良いフォーカス管理の例は react-aria-modal です。これは完全にアクセシブルなモーダルウィンドウの比較的珍しい例です。このライブラリは、最初のフォーカスをキャンセルボタンに設定し（これは、キーボードを使っているユーザがうっかり次のアクションに移ってしまうのを防ぎます）、モーダルの中でキーボードフォーカスが閉じているだけでなく、最初にモーダルを開いた要素にフォーカスを戻してもくれます。

> **補足：**
> これはとても重要なアクセシビリティ機能ですが、慎重に使用されるべきテクニックでもあります。このテクニックはキーボードフォーカスの流れが妨げられた場合の修正に使用し、ユーザがアプリケーションをどのように使いたいかを試したり予測するのに使わないでください。

マウスとポインタのイベント

マウスまたは、ポインタのイベントを通じて使われる機能がキーボード単体でも同じように使用できるようにしてください。ポインタデバイスだけに依存した実装は、多くの場合にキーボードユーザがアプリケーションを使えない原因になります。

これを説明するために、クリックイベントによってアクセシビリティが損なわれるよくある例を見てみましょう。以下の画像はアウトサイドクリックパターンというユーザが要素の外側をクリックして開いている要素を閉じられるパターンです。

Select an option

Load the option

Remove the option

これは通常、ポップアップを閉じる役割をもつ `click` イベントを `window` オブジェクトに付与して実装します。

```
class OuterClickExample extends React.Component {
constructor(props) {
    super(props);

    this.state = { isOpen: false };
    this.toggleContainer = React.createRef();

    this.onClickHandler = this.onClickHandler.bind(this);
    this.onClickOutsideHandler = this.onClickOutsideHandler.bind(this);
  }

  componentDidMount() {
    window.addEventListener('click', this.onClickOutsideHandler);
  }

  componentWillUnmount() {
    window.removeEventListener('click', this.onClickOutsideHandler);
  }

  onClickHandler() {
    this.setState(currentState => ({
      isOpen: !currentState.isOpen
    }));
  }

  onClickOutsideHandler(event) {
    if (this.state.isOpen && !this.toggleContainer.current.contains(event.target)) {
      this.setState({ isOpen: false });
    }
  }

  render() {
    return (
      <div ref={this.toggleContainer}>
        <button onClick={this.onClickHandler}>Select an option</button>
        {this.state.isOpen ? (
          <ul>
            <li>Option 1</li>
            <li>Option 2</li>
            <li>Option 3</li>
          </ul>
        ) : null}
      </div>
    );
  }
}
```

これはマウスのようなポインタデバイスでは問題なく機能しますが、キーボード単体で操作しようとした場合、タブキーによって次の要素に移動しても `window` オブジェクトは `click` イベントを受け取らないため、うまく機能しません。

Select an option

Load the option

Remove the option

これと同じ機能は `onBlur` と `onFocus` のような適切なイベントハンドラを代わりに用いることで実現できます。

```
class BlurExample extends React.Component {
  constructor(props) {
    super(props);

    this.state = { isOpen: false };
    this.timeOutId = null;

    this.onClickHandler = this.onClickHandler.bind(this);
    this.onBlurHandler = this.onBlurHandler.bind(this);
```

```
    this.onFocusHandler = this.onFocusHandler.bind(this);
  }

  onClickHandler() {
    this.setState(currentState => ({
      isOpen: !currentState.isOpen
    }));
  }

  // We close the popover on the next tick by using setTimeout.
  // This is necessary because we need to first check if
  // another child of the element has received focus as
  // the blur event fires prior to the new focus event.
  onBlurHandler() {
    this.timeOutId = setTimeout(() => {
      this.setState({
        isOpen: false
      });
    });
  }

  // If a child receives focus, do not close the popover.
  onFocusHandler() {
    clearTimeout(this.timeOutId);
  }

  render() {
    // React assists us by bubbling the blur and
    // focus events to the parent.
    return (
      <div onBlur={this.onBlurHandler}
           onFocus={this.onFocusHandler}>

        <button onClick={this.onClickHandler}
                aria-haspopup="true"
                aria-expanded={this.state.isOpen}>
          Select an option
        </button>
        {this.state.isOpen ? (
          <ul>
            <li>Option 1</li>
            <li>Option 2</li>
            <li>Option 3</li>
          </ul>
        ) : null}
      </div>
    );
  }
}
```

上記のコードは、ポインタデバイス使うユーザとキーボードを使うユーザの双方にこの機能を公開します。さらに、 `aria-*` props を加えるとスクリーンリーダを使うユーザーもサポートできます。話を簡単にするため、ポップアップの選択肢を 矢印キー で操作できるようにするキーボードイベントは実装していません。

Select an option

Load the option

Remove the option

これはポインタデバイスとマウスイベントだけに依存するとキーボードを使うユーザにとって機能が損なわれてしまう数多くの具体例のうちのひとつです。つねにキーボードによるテストをすれば、キーボードに対応するイベントを使うことで解決できる問題をすばやく発見できるでしょう。

より複雑なウィジェット

ユーザ体験がより複雑であるほど、よりアクセシビリティが損なわれるということがあってはいけません。できるだけ HTML に近くなるようコーディングすればアクセシビリティを最も簡単に達成できますが、一方でかなり複雑なウィジェットでもアクセシビリティを保ってコーディングすることができます。
ここでは ARIA のロール や ARIA のステートとプロパティについての知識も必要となります。これらは JSX で完全にサポートされている HTML 属性が詰まったツールボックスであり、十分にアクセシブルで高機能な React コンポーネントの構築を可能にしてくれます。
それぞれの種類のウィジェットはそれぞれ特定のデザインパターンを持っており、ユーザやユーザーエージェントはそれらが特定の方法で機能することを期待します：

- WAI-ARIA Authoring Practices - Design Patterns and Widgets
- Heydon Pickering - ARIA Examples
- Inclusive Components

その他に考慮すべきポイント

言語設定

ページテキストで使用する自然言語を明示して、読み上げソフトが適切な音声設定を選ぶために利用できるようにしてください：

- WebAIM - Document Language

ドキュメントの title の設定
ドキュメントの `<title>` は、ユーザが現在いるページのコンテキストを認識していられるように、そのページのコンテンツを正しく説明するものにしてください：

- WCAG - Understanding the Document Title Requirement

React では React Document Title Component を使用することで title を設定できます。

色のコントラスト
あなたのウェブサイトにある全ての読めるテキストが、色弱のユーザにも最大限読めるように配慮した色のコントラストがあることを確認してください：

- WCAG - Understanding the Color Contrast Requirement
- Everything About Color Contrast And Why You Should Rethink It
- A11yProject - What is Color Contrast

適切な色の組み合わせをウェブサイト内の全てのケースについて手作業で行うのは面倒になりがちなので、代わりにアクセシブルなカラーパレット全体を Colorable で計算することができます。
以下に述べる aXe および WAVE ツールのどちらも同じように色のコントラストのテストを備えておりコントラストの違反を報告してくれます。
コントラストをチェックする能力を拡張したい場合は、以下のツールが利用できます：

- WebAIM - Color Contrast Checker
- The Paciello Group - Color Contrast Analyzer

開発とテストのツール
アクセシブルなウェブアプリケーションの作成を支援するために利用できる様々なツールがあります。

キーボード
最も簡単で最も重要なチェックのうちのひとつは、ウェブサイト全体がキーボード単体であまねく探索でき、使えるかどうかのテストです。これは以下の手順でチェックできます。

1. マウスを外します。
2. `Tab` と `Shift+Tab` を使ってブラウズします。
3. 要素を起動するのに `Enter` を使用します。
4. 必要に応じて、キーボードの矢印キーを使ってメニューやドロップダウンリストなどの要素を操作します。

開発支援
アクセシビリティ機能には JSX のコード内で直接チェックできるものもあります。JSX に対応した IDE では、ARIA ロールやステートやプロパティに対する intellisense によるチェックが既に提供されていることが多いでしょう。他にも以下のツールを使うこともできます：

eslint-plugin-jsx-a11y
ESLint の eslint-plugin-jsx-a11y プラグインはあなたの JSX コードのアクセシビリティに対して、AST による lint のフィードバックを提供します。多くの IDE はコード解析とソースコードのウィンドウに直接そのフィードバックを統合できるようになっています。
Create React App はこのプラグインを備えており、一部のルールを有効化しています。もし、より多くのアクセシビリティルールを有効化したいときは、プロジェクトルートに `.eslintrc` ファイルを作成し、以下の内容を書き込んでください：

```
{
  "extends": ["react-app", "plugin:jsx-a11y/recommended"],
  "plugins": ["jsx-a11y"]
}
```

ブラウザでアクセシビリティをテストする
ブラウザからウェブページのアクセシビリティを検査できるツールは沢山あります。それらのツールはあなたが作成した HTML の技術的なアクセシビリティしかチェックできないため、ここで言及した他のアクセシビリティ確認法と組み合わせて使用してください。

aXe, aXe-core and react-axe
Deque System はアプリケーションの自動化された E2E アクセシビリティテストを行う aXe-core を提供しています。このモジュールは Selenium に統合できます。
The Accessibility Engine もしくは aXe は、`aXe-core` により構築されたアクセシビリティを検査するブラウザ拡張機能です。
react-axe モジュールを使用して、開発時やデバッグ時にこれらによるアクセシビリティの検査結果を直接コンソールへ出力させることもできます。

WebAIM WAVE
Web Accessibility Evaluation Tool はアクセシビリティに関する別のブラウザ拡張機能です。

アクセシビリティ検査ツールとアクセシビリティツリー

アクセシビリティツリー (The Accessibility Tree) は、スクリーンリーダのような補助技術に公開されるべきすべての要素についてアクセス可能なオブジェクトを含んだ DOM ツリーのサブセットです。
一部のブラウザではアクセシビリティツリー内の各要素のアクセシビリティに関する情報を簡単に見ることができます：

- Using the Accessibility Inspector in Firefox
- Activate the Accessibility Inspector in Chrome
- Using the Accessibility Inspector in OS X Safari

スクリーンリーダ
アクセシビリティのテストの一環として、スクリーンリーダによるテストを行うべきです。
ブラウザとスクリーンリーダの相性に注意してください。選択したスクリーンリーダに最適なブラウザでアプリケーションのテストをすることをおすすめします。

よく使われるスクリーンリーダ

NVDA と FireFox
NonVisual Desktop Access または NVDA は広く利用されているオープンソースの Windows 向けスクリーンリーダーです。
NVDA を最大限に活用する方法は以下のガイドを参照してください：

- WebAIM - Using NVDA to Evaluate Web Accessibility
- Deque - NVDA Keyboard Shortcuts

VoiceOver と Safari
VoiceOver は Apple 社製品に統合されたスクリーンリーダです。
VoiceOver を有効化して使用する方法は以下のガイドを参照してください：

- WebAIM - Using VoiceOver to Evaluate Web Accessibility
- Deque - VoiceOver for OS X Keyboard Shortcuts
- Deque - VoiceOver for iOS Shortcuts

JAWS と Internet Explorer
Job Access With Speech もしくは JAWS は、Windows 上での使用例が豊富なスクリーンリーダです。
JAWS を最大限に活用する方法は以下のガイドを参照してください：

- WebAIM - Using JAWS to Evaluate Web Accessibility
- Deque - JAWS Keyboard Shortcuts

他のスクリーンリーダ

ChromeVox と Google Chrome
ChromeVox は Chromebook に統合されたスクリーンリーダで、Google Chrome では拡張機能として利用可能です。
ChromeVox を最大限に活用する方法は以下のガイドを参照してください：

- Google Chromebook Help - Use the Built-in Screen Reader
- ChromeVox Classic Keyboard Shortcuts Reference

このページを編集する