Frameworks para desarrollo web y la accesibilidad

Análisis de desarrollo web con React y accesibilidad

La accesibilidad web (también conocida como a11y) es el diseño y la creación de sitios web que pueden ser utilizados por todos. El soporte de accesibilidad es necesario para permitir que la tecnología de asistencia interprete las páginas web.

React es totalmente compatible con la creación de sitios web accesibles, a menudo mediante el uso de técnicas estándar de HTML.

Normas y lineamientos

Las normas y lineamientos utilizadas son variadas entre las cuales podemos encontrar las detalladas en las siguientes subsecciones:

WCAG

Las `Pautas de Accesibilidad de Contenido Web` (*WCAG por sus siglas en inglés*) proporcionan pautas para crear sitios web accesibles.

WAI-ARIA

El documento `Iniciativa de Accesibilidad Web - Aplicaciones de Internet Enriquecidas y Accesibles` (*WAI-ARIA por sus siglas en inglés*) contiene técnicas para construir widgets de JavaScript totalmente accesibles.

Ten en cuenta que todos los atributos HTML `aria-*` son totalmente compatibles con JSX. Mientras que la mayoría de las propiedades y atributos de DOM en React son `camelCase`, estos atributos **deben tener un guión** (también conocido como kebab-case, lisp-case, etc.) ya que están en HTML simple:

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

HTML Semántico

El HTML semántico es la base de la accesibilidad en una aplicación web. Haciendo uso de los diversos elementos HTML para reforzar el significado de la información en nuestros sitios web a

menudo nos dará accesibilidad de forma sencilla y sin costo.

A veces rompemos la semántica HTML cuando agregamos elementos `<div>` a nuestro JSX para hacer que nuestro código React funcione, especialmente cuando trabajamos con listas (`<ol>`, `<ul>` y `<dl>`) y la etiqueta `<table>` de HTML. En estos casos, deberíamos usar `Fragmentos React` para agrupar varios elementos.

```
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>
  )
}
```

Puedes asignar una colección de elementos a un arreglo de fragmentos como lo haría con cualquier otro tipo de elemento:

```
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>
  )
}
```

o utilizando la sintaxis corta cuando el Fragment no necesita ninguna prop en la etiqueta,

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

Formularios accesibles

Etiquetas

Todos los controles de formulario HTML, como `<input>` y `<textarea>`, deben ser etiquetados de forma accesible. Necesitamos proporcionar etiquetas descriptivas que también estén expuestas a los lectores de pantalla.

Los siguientes recursos nos muestran cómo hacer esto:

- El W3C nos muestra cómo etiquetar elementos
- WebAIM nos muestra cómo etiquetar elementos
- El Grupo Paciello explica los nombres accesibles

Aunque estas prácticas estándar de HTML se pueden usar directamente en React, se debe tener en cuenta que el atributo `for` se escribe como `htmlFor` en JSX, por lo que la migración de las mismas no es completamente directa. Cabe destacar sin embargo que el mapeo de atributos es muy sencillo utilizando esta librería:

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

Eventos

El desarrollador debe asegurarse de que el usuario pueda acceder a todos los eventos asociados al mouse. Para esto uno debería poder hacer utilizando solo el teclado.

En el ejemplo siguiente esto no sucede dado solo puede funcionar bien para los usuarios con dispositivos de puntero, como un ratón, pero si lo hace solo con el teclado la funcionalidad se rompe al pasar al elemento siguiente, ya que el objeto `window` nunca recibe el evento `click`. Esto puede llevar a una funcionalidad oculta que impide que los usuarios utilicen su aplicación.

```
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>
    )
  }
}
```

La misma funcionalidad se puede lograr utilizando un controlador de eventos apropiado, como `onBlur` y `onFocus`:

```
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
    }))
  }

  // Cerramos la ventana emergente en el siguiente tick usando setTimeout.
  // Esto es necesario porque primero debemos comprobar
  // si otro hijo del elemento ha recibido el foco ya que
  // el evento de desenfoque se dispara antes del nuevo evento de foco.
  onBlurHandler() {
    this.timeOutId = setTimeout(() => {
      this.setState({
        isOpen: false
      })
    })
  }

  // Si un hijo recibe el foco, no cerrar la ventana emergente.
  onFocusHandler() {
    clearTimeout(this.timeOutId)
  }

  render() {
    // React nos ayuda burbujeando los eventos de desenfoque
    // y enfoque hacia los padres.
    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>
    )
  }
}
```

Herramientas de desarrollo

Hay una serie de herramientas que podemos utilizar para ayudar en la creación de aplicaciones web accesibles.

El teclado

La comprobación más fácil y también una de las más importantes es, por mucho, comprobar si se puede acceder a todo el sitio web y usarlo solo con el teclado. Hágalo de la siguiente forma:

- Desconecte su mouse.
- Usando `Tab` y `Shift + Tab` para navegar.
- Usando Enter para activar elementos.

Cuando sea necesario, utilice las teclas de flecha del teclado para interactuar con algunos elementos, como menús y menús desplegables.

Asistencia para el desarrollo

El complemento [eslint-plugin-jsx-a11y] (https://github.com/evcohen/eslint-plugin-jsx-a11y) para ESLint proporciona linting de AST sobre los problemas de accesibilidad en tu JSX. Muchos IDE’s te permiten integrar estos hallazgos directamente en el análisis de código y las ventanas de código fuente.

`Create React App` tiene este complemento con un subconjunto de reglas activadas. Si desea habilitar aún más reglas de accesibilidad, puede crear un archivo `.eslintrc` en la raíz de su proyecto con este contenido:

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

Probando accesibilidad en un navegador

Existen varias herramientas que pueden ejecutar auditorías de accesibilidad en las páginas web de su navegador. Utilízalas en combinación con otras comprobaciones de accesibilidad que se mencionan aquí, ya que solo pueden probar la accesibilidad técnica de su HTML.

Inspectores de accesibilidad y el Árbol de Accesibilidad

El `Árbol de Accesibilidad` es un subconjunto del árbol DOM que contiene objetos accesibles para cada elemento del DOM que debería ser expuesto a la tecnología de asistencia, como los lectores de pantalla.

En algunos navegadores podemos ver fácilmente la información de accesibilidad para cada elemento en el árbol de accesibilidad:

- Usando el inspector de accesibilidad en Firefox

- Activar el inspector de accesibilidad en Chrome

Lectores de pantalla

Las pruebas con un lector de pantalla deben formar parte de sus pruebas de accesibilidad.

Ten en cuenta que las combinaciones de navegador/lector de pantalla son importantes. Se recomienda que pruebe su aplicación en el navegador que mejor se adapte a su lector de pantalla.