Accessibility

**Web accessibility** (also referred to as a11y) is the **design and creation of websites** that can be used by everyone. Accessibility support is necessary to allow assistive technology to interpret web pages.

React fully supports building accessible websites, often by using standard HTML techniques.

**WAI-ARIA**

The **Web Accessibility Initiative - Accessible Rich Internet Applications** document contains techniques for building **fully accessible JavaScript widgets.**

<https://www.w3.org/WAI/standards-guidelines/aria/>

Note that all **aria-\* HTML attributes are fully supported in JSX.** Whereas most DOM properties and attributes in React are **camelCased**, these attributes should be **hyphen-cased (also known as kebab-case, lisp-case, etc)** as they are in plain HTML:

*<input*

*type="text"*

***aria-label={labelText}***

***aria-required="true"***

*onChange={onchangeHandler}*

*value={inputValue}*

*name="name"*

*/>*

**Semantic HTML**

Semantic HTML is the foundation of accessibility in a web application. Using the various HTML elements to reinforce the meaning of information in our websites will often give us accessibility for free.

Sometimes we break HTML semantics when we add <div> elements to our JSX to make our React code work, especially when working with lists (<ol>, <ul> and <dl>) and the HTML <table>. **In these cases we should rather use React Fragments to group together multiple elements.**

You can **map a collection of items to an array of fragments** as you would any other type of element as well:

*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>*

*);*

*}*

**When you don’t need any props on the Fragment tag** you can use the short syntax, if your tooling supports it:

*function ListItem({ item }) {*

*return (*

***<>***

*<dt>{item.term}</dt>*

*<dd>{item.description}</dd>*

***</>***

*);*

*}*

**Accessible Forms**

Labelling:

Every HTML form control, such as <input> and <textarea>, needs to be labeled accessibly. We need to provide descriptive labels that are also exposed to screen readers.

Although these standard HTML practices **can be directly used in React, note that the for attribute is written as htmlFor in JSX:**

*<label* ***htmlFor="namedInput"****>Name:</label>*

*<input id="namedInput" type="*text*" name="name"/>*

**Notifying User of Errors**

**Focus Control**

Ensure that your web application can be fully operated with the keyboard only.

Keyboard focus refers to the **current element in the DOM** **that is selected to accept input from the keyboard**. We see it everywhere as a focus outline similar to that shown in the following image:

[Blue keyboard focus outline around a selected link.](https://reactjs.org/static/keyboard-focus-dec0e6bcc1f882baf76ebc860d4f04e5-9d63d.png)

Only ever use CSS that removes this outline, for example **by setting outline: 0,** if you are replacing it with another focus outline implementation.

**Programmatically Managing Focus**

Our React applications continuously modify the HTML DOM during runtime, sometimes leading to keyboard focus being lost or set to an unexpected element. In order to repair this, we need to programmatically nudge the keyboard focus in the right direction. For example, **by resetting keyboard focus to a button that opened a modal window after that modal window is closed**.

To set focus in React, we can use **Refs** to DOM elements.

Using this, we first create a **ref to an element in the JSX** of a component class:

*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}***

*/>*

*);*

*}*

*}*

**Then we can focus it elsewhere in our component when needed:**

*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();***

*}*

**Sometimes a parent component needs to set focus to an element in a child component.** We can do this by **exposing DOM refs to parent components** through a special prop on the child component that forwards the parent’s ref to the child’s DOM node.

*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();**

**Code Splitting**

-Bundling

Most React apps will have their files “bundled” using tools like **Webpack** or **Browserify.** **Bundling is the process of following imported files and merging them into a single file: a “bundle”.**

**This bundle can then be included on a webpage to load an entire app at once.**

Example

**App:**

*// app.js*

*import { add } from './math.js';*

*console.log(add(16, 26)); // 42*

// math.js

export function add(a, b) {

return a + b;

}

**Bundle:**

*function add(a, b) {*

*return a + b;*

*}*

*console.log(add(16, 26)); // 42*

If you’re using **Create React App, Next.js, Gatsby, or a similar tool**, you will have a **Webpack** setup out of the box to bundle your app.

If you aren’t, you’ll need to setup bundling yourself. For example, see the Installation and Getting Started guides on the Webpack docs.

**Code Splitting**

Bundling is great, but **as your app grows, your bundle will grow too**. Especially if you are including large third-party libraries. You need to keep an eye on the code you are including in your bundle so that you don’t accidentally make it so large that your app takes a long time to load.

To avoid winding up with a large bundle, it’s good to get ahead of the problem and start “splitting” your bundle. **Code-Splitting is a feature supported by bundlers like** **Webpack and Browserify** **(via factor-bundle) which can create multiple bundles that can be dynamically loaded at runtime.**

Code-splitting your app can help you **“lazy-load”** **just the things that are currently needed by the user**, which can dramatically improve the performance of your app. While you haven’t reduced the overall amount of code in your app, you’ve avoided loading code that the user may never need, and reduced the amount of code needed during the initial load.

**import()**

**The best way to introduce code-splitting into your app is through the dynamic import() syntax.**

**Before:**

*import { add } from './math';*

*console.log(add(16, 26));*

**After:**

*import("./math").then(math => {*

*console.log(math.add(16, 26));*

*});*

Note:

The dynamic import() syntax is a ECMAScript (JavaScript) proposal not currently part of the language standard. It is expected to be accepted in the near future.

When Webpack comes across this syntax, it automatically starts code-splitting your app. If you’re using Create React App, this is already configured for you and you can start using it immediately. It’s also supported out of the box in Next.js.

If you’re setting up Webpack yourself, you’ll probably want to read Webpack’s guide on code splitting. Your Webpack config should look vaguely like this.

When using Babel, you’ll need to make sure that Babel can parse the dynamic import syntax but is not transforming it. For that you will need babel-plugin-syntax-dynamic-import.

**React.lazy**

Note:

**React.lazy and Suspense** is not yet available for server-side rendering. If you want to do code-splitting in a server rendered app, we recommend Loadable Components. It has a nice guide for bundle splitting with server-side rendering.

**The React.lazy function lets you render a dynamic import as a regular component.**

**Before:**

*import OtherComponent from './OtherComponent';*

*function MyComponent() {*

*return (*

*<div>*

*<OtherComponent />*

*</div>*

*);*

*}*

**After:**

***const OtherComponent = React.lazy(() => import('./OtherComponent'));***

*function MyComponent() {*

*return (*

*<div>*

*<OtherComponent />*

*</div>*

*);*

*}*

**This will automatically load the bundle containing the OtherComponent when this component gets rendered.**

**Suspense**

**If the module containing the OtherComponent is not yet loaded by the time MyComponent renders**, we must show some **fallback** content while we’re waiting for it to load - **such as a loading indicator**. This is done using the **Suspense component.**

***const OtherComponent = React.lazy(() => import('./OtherComponent'));***

*function MyComponent() {*

*return (*

*<div>*

***<Suspense fallback={<div>Loading...</div>}>***

*<OtherComponent />*

***</Suspense>***

*</div>*

*);*

*}*

The **fallback** prop accepts any React elements that you want to render **while waiting for the component to load**. You can place the Suspense component anywhere above the lazy component. **You can even wrap multiple lazy components with a single Suspense component.**

***const OtherComponent = React.lazy(() => import('./OtherComponent'));***

***const AnotherComponent = React.lazy(() => import('./AnotherComponent'));***

*function MyComponent() {*

*return (*

*<div>*

***<Suspense fallback={<div>Loading...</div>}>***

***<section>***

*<OtherComponent />*

*<AnotherComponent />*

***</section>***

***</Suspense>***

*</div>*

*);*

*}*

**React.lazy takes a function that must call a dynamic import().**

**This must return a Promise which resolves to a module with a default export containing a React component.**

**Error** **boundaries**

**If the other module fails to load** (for example, due to network failure), it will trigger an error. You can handle these errors to show a nice user experience and manage recovery with Error Boundaries. **Once you’ve created your Error Boundary, you can use it anywhere above your lazy components to display an error state when there’s a network error.**

***import MyErrorBoundary from './MyErrorBoundary';***

***const OtherComponent = React.lazy(() => import('./OtherComponent'));***

***const AnotherComponent = React.lazy(() => import('./AnotherComponent'));***

*const MyComponent = () => (*

*<div>*

***<MyErrorBoundary>***

***<Suspense fallback={<div>Loading...</div>}>***

***<section>***

*<OtherComponent />*

*<AnotherComponent />*

***</section>***

***</Suspense>***

***</MyErrorBoundary>***

*</div>*

*);*

**Route-based code splitting**

Deciding where in your app to introduce code splitting can be a bit tricky. You want to make sure you choose places that will split bundles evenly, but won’t disrupt the user experience.

**A good place to start is with routes. Most people on the web are used to page transitions taking some amount of time to load. You also tend to be re-rendering the entire page at once so your users are unlikely to be interacting with other elements on the page at the same time.**

Here’s an example of **how to setup route-based code splitting into your app using libraries like React Router with React.lazy.**

*import { BrowserRouter as Router, Route, Switch } from 'react-router-dom';*

***import React, { Suspense, lazy } from 'react';***

***const Home = lazy(() => import('./routes/Home'));***

***const About = lazy(() => import('./routes/About'));***

*const App = () => (*

***<Router>***

***<Suspense fallback={<div>Loading...</div>}>***

***<Switch>***

***<Route exact path="/" component={Home}/>***

***<Route path="/about" component={About}/>***

***</Switch>***

***</Suspense>***

***</Router>***

*);*

**Named Exports**

**React.lazy currently only supports default exports**. If the module you want to import uses named exports, you can create an intermediate module that **reexports** it as the default. This ensures that treeshaking keeps working and that you don’t pull in unused components.

*// ManyComponents.js*

***export const MyComponent = /\* ... \*/;***

***export const MyUnusedComponent = /\* ... \*/;***

*// MyComponent.js*

***export { MyComponent as default } from "./ManyComponents.js";***

*// MyApp.js*

***import React, { lazy } from 'react';***

***const MyComponent = lazy(() => import("./MyComponent.js"));***

**Context**

Context provides a way **to pass data through the component tree** without having to pass props down manually at every level.

In a typical React application, data is passed **top-down (parent to child)** **via** **props**, but this can be cumbersome for certain types of props (e.g. locale preference, UI theme) that are required by many components within an application. **Context provides a way to share values like these between components without having to explicitly pass a prop through every level of the tree.**

**When to use Context**

Context is designed to share data that can be considered **“global”** **for a tree of React components**, such as the current authenticated user, theme, or preferred language.

*class App extends React.Component {*

*render() {*

***return <Toolbar theme="dark" />;***

*}*

*}*

*function Toolbar(props) {*

***// The Toolbar component must take an extra "theme" prop***

***// and pass it to the ThemedButton. This can become painful***

***// if every single button in the app needs to know the theme***

***// because it would have to be passed through all components.***

*return (*

*<div>*

***<ThemedButton theme={props.theme} />***

*</div>*

*);*

*}*

*class ThemedButton extends React.Component {*

*render() {*

*return* ***<Button theme={this.props.theme} />;***

*}*

*}*

Using context, we can avoid passing props through **intermediate elements:**

***// Context lets us pass a value deep into the component tree***

***// without explicitly threading it through every component.***

***// Create a context for the current theme (with "light" as the default).***

***const ThemeContext = React.createContext('light');***

*class App extends React.Component {*

*render() {*

***// Use a Provider to pass the current theme to the tree below.***

***// Any component can read it, no matter how deep it is.***

***// In this example, we're passing "dark" as the current value.***

*return (*

***<ThemeContext.Provider value="dark">***

***<Toolbar />***

***</ThemeContext.Provider>***

*);*

*}*

*}*

***// A component in the middle doesn't have to***

***// pass the theme down explicitly anymore.***

*function Toolbar(props) {*

*return (*

*<div>*

***<ThemedButton />***

*</div>*

*);*

*}*

*class ThemedButton extends React.Component {*

***// Assign a contextType to read the current theme context.***

***// React will find the closest theme Provider above and use its value.***

***// In this example, the current theme is "dark".***

***static contextType = ThemeContext;***

*render() {*

*return* ***<Button theme={this.context} />****;*

*}*

*}*

**Before You Use Context**

Context is primarily used **when some data needs to be accessible by many components at different nesting levels.** Apply it sparingly because it makes component reuse more difficult.

If you only want to avoid passing some props through many levels, component composition is often a simpler solution than context.

For example, consider a Page component that passes a user and avatarSize prop several levels down so that deeply nested Link and Avatar components can read it:

*<Page* ***user={user}******avatarSize={avatarSize}*** */>*

*// ... which renders ...*

*<PageLayout* ***user={user}******avatarSize={avatarSize****} />*

*// ... which renders ...*

*<NavigationBar* ***user={user}******avatarSize={avatarSize}*** */>*

*// ... which renders ...*

*<Link* ***href={user.permalink}****>*

***<Avatar user={user} size={avatarSize} />***

*</Link>*

It might feel redundant to pass down the **user** and **avatarSize** props through many levels if **in the end only the Avatar component really needs it**. It’s also annoying that whenever the Avatar component needs more props from the top, you have to add them at all the intermediate levels too.

**One way to solve this issue without context is to pass down the Avatar component itself so that the intermediate components don’t need to know about the user or avatarSize props:**

*function Page(props) {*

*const user = props.user;*

*const userLink = (*

***<Link href={user.permalink}>***

***<Avatar user={user} size={props.avatarSize} />***

***</Link>***

*);*

*return <PageLayout* ***userLink={userLink}*** */>;*

*}*

*// Now, we have:*

*<Page user={user} avatarSize={avatarSize} />*

*// ... which renders ...*

*<PageLayout* ***userLink****={...} />*

*// ... which renders ...*

*<NavigationBar* ***userLink****={...} />*

*// ... which renders ...*

***{props.userLink}***

With this change, only the top-most Page component needs to know about the Link and Avatar components’ use of user and avatarSize.

This **inversion of control** can make your code cleaner in many cases by reducing the amount of props you need to pass through your application and giving more control to the root components. However, this isn’t the right choice in every case: moving more complexity higher in the tree makes those higher-level components more complicated and forces the lower-level components to be more flexible than you may want.

**You’re not limited to a single child for a component.** You may pass multiple children, or even have multiple separate “slots” for children, as documented here:

*function Page(props) {*

*const user = props.user;*

***const content = <Feed user={user} />;***

***const topBar = (***

***<NavigationBar>***

***<Link href={user.permalink}>***

***<Avatar user={user} size={props.avatarSize} />***

***</Link>***

***</NavigationBar>***

***);***

*return (*

***<PageLayout***

***topBar={topBar}***

***content={content}***

***/>***

*);*

*}*

This pattern is sufficient for many cases **when you need to decouple a child from its immediate parents.** You can take it even further with render props if the child needs to communicate with the parent before rendering.

However, **sometimes the same data needs to be accessible by many components in the tree, and at different nesting levels.**

**Context lets you “broadcast” such data, and changes to it, to all components below.** Common examples where using context might be simpler than the alternatives include **managing the current locale, theme, or a data cache.**

**API**

**React.createContext**

***const MyContext = React.createContext(defaultValue);***

**Creates a Context object**. **When React renders a component that subscribes to this Context object it will read the current context value from the closest matching Provider above it in the tree.**

**The defaultValue argument is only used when a component does not have a matching Provider above it in the tree**.

**This can be helpful for testing components in isolation without wrapping them.**

**Note**: passing undefined as a Provider value does not cause consuming components to use defaultValue.

**Context.Provider**

***<MyContext.Provider value={/\* some value \*/}>***

Every Context object comes with a **Provider** **React component that allows consuming components to subscribe to context changes.**

Accepts a **value** **prop** to be passed to consuming **components that are descendants of this Provider.**

**One Provider can be connected to many consumers.** Providers can be nested to override values deeper within the tree.

**All consumers that are descendants of a Provider will re-render whenever the Provider’s value prop changes.** The propagation from Provider to its descendant consumers is not subject to the **shouldComponentUpdate** method**, so the consumer is updated even when an ancestor component bails out of the update.**

**Changes are determined by comparing the new and old values using the same algorithm as Object.is.**

**Class.contextType**

*class MyClass extends React.Component {*

*componentDidMount() {*

*let value = this.context;*

***/\* perform a side-effect at mount using the value of MyContext \*/***

*}*

*componentDidUpdate() {*

***let value = this.context;***

*/\* ... \*/*

*}*

*componentWillUnmount() {*

***let value = this.context;***

*/\* ... \*/*

*}*

*render() {*

*let value = this.context;*

***/\* render something based on the value of MyContext \*/***

*}*

*}*

***MyClass.contextType = MyContext;***

**The contextType property on a class can be assigned a Context object created by React.createContext().**

**This lets you consume the nearest current value of that Context type using this.context.**

**You can reference this in any of the lifecycle methods including the render function.**

**Note:** You can only subscribe to a single context using this API.

If you are using the experimental public class fields syntax, you can use a static class field to initialize your contextType.

*class MyClass extends React.Component {*

***static contextType = MyContext;***

*render() {*

***let value = this.context;***

***/\* render something based on the value \*/***

*}*

*}*

**Context.Consumer**

***<MyContext.Consumer>***

***{value => /\* render something based on the context value \*/}***

***</MyContext.Consumer>***

**A React component that subscribes to context changes.** This lets you subscribe to a context within a function component.

**Requires a function as a child. The function receives the current context value and returns a React node.**

**The value argument passed to the function will be equal to the value prop of the closest Provider for this context above in the tree.** If there is no Provider for this context above, the value argument will be equal to the **defaultValue** that was passed to **createContext().**