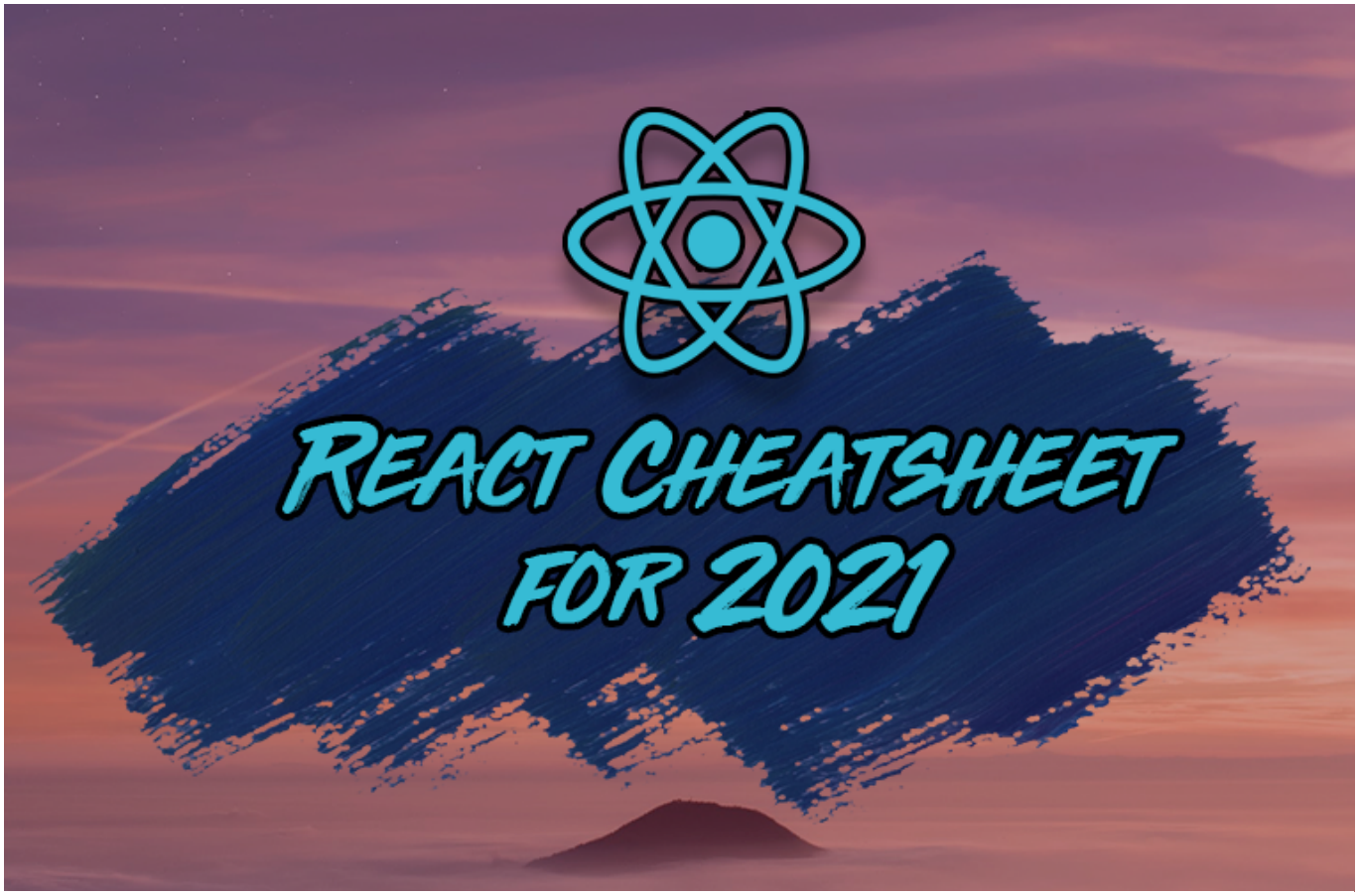
JANUARY 9, 2021 / #REACT

The React Cheatsheet for 2021 (+ Real-World Examples)

Reed Barger



I have put together a comprehensive visual cheatsheet to help you master all the major concepts and features of the React library in 2021.

It includes tons of practical examples to illustrate every feature of the library and how it works using patterns you can apply within your own projects.

Along with each code snippet, I have added many helpful comments. If you read these comments, you'll see what each line of code does, how different concepts relate to one another, and gain a more complete understanding of how React can be used.

Note that the keywords that are particularly useful for you to know as a React developer are highlighted in bold, so look out for those.

Want Your Own Copy of the Cheatsheet?

Download the cheatsheet in PDF format here (it takes 5 seconds).

Here are some quick wins from grabbing the downloadable version:

- ✓ Quick reference guide to review however and whenever
- ✓ Tons of copyable code snippets for easy reuse
- ✓ Read this massive guide wherever suits you best. On the train, at your desk, standing in line... anywhere.

There's a ton of great stuff to cover, so let's get started.

> Want to run any of the code snippets below? Create a new React application to try out any of these examples using the (free) online tool CodeSandbox. You can do so instantly by visiting react.new.

React Fundamentals

- JSX Elements
- Components and Props
- Lists and Keys
- Event Listeners and Handling Events

Essential React Hooks

- State and useState
- Side Effects and useEffect
- Refs and useRef

Hooks and Performance

- Preventing Re-renders and React.memo
- Callback functions and useCallback
- Memoization and useMemo

Advanced React Hooks

- Context and useContext
- Reducers and useReducer
- Writing custom hooks
- Rules of hooks

JSX Elements

React applications are structured using a syntax called **JSX**. This is the syntax of a basic **JSX element**.

```
/*
  JSX allows us to write in a syntax almost identical to plain HTM
  As a result, JSX is a powerful tool to structure our application
  JSX uses all valid HTML tags (i.e. div/span, h1-h6, form/input,
*/

<div>Hello React!</div>

/*
  Note: this JSX would not be visible because it needs to be rende
*/
```

JSX is the most common way to structure React applications, but it is not required for React.

```
/* JSX is a simpler way to use the function React.createElement()
In other words, the following two lines in React are the same: */

<div>Hello React!</div>  // JSX syntax

React.createElement('div', null, 'Hello React!'); // createElement
```

JSX is not understood by the browser. It needs to be compiled to plain JavaScript, which the browser can understand.

```
/*
  When our project is built to run in the browser, our JSX will be
  From this...
*/
const greeting = <div>Hello React!</div>;

/* ...into this: */
"use strict";

const greeting = /*#__PURE__*/React.createElement("div", null, "He
```

JSX differs from HTML in several important ways:

```
/*
  We can write JSX like plain HTML, but it's actually made using
  Because JSX is JavaScript, not HTML, there are some differences

  1) Some JSX attributes are named differently than HTML attribut

  Also, because JSX is JavaScript, attributes that consist of mul
*/

<div id="header">
  <h1 className="title">Hello React!</h1>
</div>

/*
  2) JSX elements that consist of only a single tag (i.e. input,
*/

<input type="email" /> // <input type="email"> is a syntax error

/*
  3) JSX elements that consist of an opening and closing tag (i.e
*/
```

Inline styles can be added to JSX elements using the style attribute. And styles are updated within an object, not a set of double quotes, as with HTML.

Note that style property names must be also written in camelcase.

```
/*
  Properties that accept pixel values (like width, height, padding
  For example: fontSize: 22. Instead of: fontSize: "22px"
*/
<h1 style={{ color: 'blue', fontSize: 22, padding: '0.5em 1em' }}>
  Hello React!
</h1>
```

JSX elements are JavaScript expressions and can be used as such. JSX gives us the full power of JavaScript directly within our user interface.

```
/*
  JSX elements are expressions (resolve to a value) and therefore
*/
const greeting = <div>Hello React!</div>;

const isNewToReact = true;

// ... or can be displayed conditionally
function sayGreeting() {
  if (isNewToReact) {
    // ... or returned from functions, etc.
    return greeting; // displays: Hello React!
  } else {
    return <div>Hi again, React</div>;
```

JSX allows us to insert (or embed) simple JavaScript expressions using the curly braces syntax:

```
const year = 2021;

/* We can insert primitive JS values (i.e. strings, numbers, boole
const greeting = <div>Hello React in {year}</div>;

/* We can also insert expressions that resolve to a primitive valu
const goodbye = <div>Goodbye previous year: {year - 1}</div>

/* Expressions can also be used for element attributes */
const className = 'title';
const title = <h1 className={className}>My title</h1>

/* Note: trying to insert object values (i.e. objects, arrays, map
```

JSX allows us to nest elements within one another, like we would HTML.

```
/*
  To write JSX on multiple lines, wrap in parentheses: ()
  JSX expressions that span multiple lines are called multiline ex
*/

const greeting = (
  // div is the parent element
  <div>
    {/* h1 and p are child elements */}
    <h1>Hello!</h1>
    <p>Welcome to React</p>
```

```
to one another, like we would talk about HTML elements */
```

Comments in JSX are written as multiline JavaScript comments, written between curly braces, like this:

```
const greeting = (
  <div>
    {/* This is a single line comment */}
    <h1>Hello!</div>
    <p>Welcome to React</p>
    {/* This is a
      multiline
      comment */}
  </div>
);
```

All React apps require three things:

1. `ReactDOM.render()`: used to render (show) our app by mounting it onto an HTML element
2. A JSX element: called a "root node", because it is the root of our application. Meaning, rendering it will render all children within it
3. An HTML (DOM) element: Where the app is inserted within an HTML page. The element is usually a div with an id of "root", located in an index.html file.

```
import ReactDOM from "react-dom";

// root node (usually a component) is most often called "App"
const App = <h1>Hello React!</h1>;

// ReactDOM.render(root node, HTML element)
ReactDOM.render(App, document.getElementById("root"));
```

Components and Props

JSX can be grouped together within individual functions called **components**.

There are two types of components in React: **function components** and **class components**.

Component names, for function or class components, are capitalized to distinguish them from plain JavaScript functions that do not return JSX:

```
import React from "react";

/*
  Function component
  Note the capitalized function name: 'Header', not 'header'
*/
function Header() {
  return <h1>Hello React</h1>;
}

// Function components which use an arrow function syntax are als
const Header = () => <h1>Hello React</h1>;
```

```
*/
class Header extends React.Component {
  render() {
    return <h1>Hello React</h1>;
  }
}
```

Components, despite being functions, are not called like ordinary JavaScript functions. They are executed by rendering them like we would JSX in our app.

```
// Do we call this function component like a normal function?

// No, to execute them and display the JSX they return...
const Header = () => <h1>Hello React</h1>;

// ...we use them as 'custom' JSX elements
ReactDOM.render(<Header />, document.getElementById("root"));
// renders: <h1>Hello React</h1>
```

The huge benefit of components is their ability to be reused across our apps, wherever we need them.

Since components leverage the power of JavaScript functions, we can logically pass data to them, like we would by passing it one or more arguments.

```
/*
  The Header and Footer components can be reused in any page in ou
  Components remove the need to rewrite the same JSX multiple time
```

```
  return (
    <div>
      <Header />
      <Hero />
      <Footer />
    </div>
  );
}

// AboutPage component, visible on the '/about' route
function AboutPage() {
  return (
    <div>
      <Header />
      <About />
      <Testimonials />
      <Footer />
    </div>
  );
}
```

Data passed to components in JavaScript are called **props**. Props look identical to attributes on plain JSX/HTML elements, but you can access their values within the component itself.

Props are available in parameters of the component to which they are passed. Props are always included as properties of an object.

```
/*
  What if we want to pass custom data to our component from a pare
  For example, to display the user's name in our app header.
*/

const username = "John";
```

```
  To pass the user's name to the header, we use a prop we appropri
*/
ReactDOM.render(
  <Header username={username} />,
  document.getElementById("root")
);
// We called this prop 'username', but can use any valid identifie

// props is the object that every component receives as an argumen
function Header(props) {
  // the props we make on the component (username)
  // become properties on the props object
  return <h1>Hello {props.username}</h1>;
}
```

Props must never be directly changed within the child component.

Another way to say this is that props should never be **mutated**, since props are a plain JavaScript object.

```
/*
  Components should operate as 'pure' functions.
  That is, for every input, we should be able to expect the same o
  This means we cannot mutate the props object, only read from it.
*/

// We cannot modify the props object :
function Header(props) {
  props.username = "Doug";

  return <h1>Hello {props.username}</h1>;
}
/*
  But what if we want to modify a prop value that is passed to our
  That's where we would use state (see the useState section).
*/
```

The **children** prop is useful if we want to pass elements / components as props to other components.

```
// Can we accept React elements (or components) as props?
// Yes, through a special property on the props object called 'chi

function Layout(props) {
  return <div className="container">{props.children}</div>;
}

// The children prop is very useful for when you want the same
// component (such as a Layout component) to wrap all other compon
function IndexPage() {
  return (
    <Layout>
      <Header />
      <Hero />
      <Footer />
    </Layout>
  );
}

// different page, but uses same Layout component (thanks to child
function AboutPage() {
  return (
    <Layout>
      <About />
      <Footer />
    </Layout>
  );
}
```

Again, since components are JavaScript expressions, we can use them in combination with if-else statements and switch statements to conditionally show content, like this:

```
  const isAuthenticated = checkAuth();

  /* if user is authenticated, show the authenticated app, otherwi
  if (isAuthenticated) {
    return <AuthenticatedApp />
  } else {
    /* alternatively, we can drop the else section and provide a s
    return <UnAuthenticatedApp />
  }
}
```

To use conditions within a component's returned JSX, you can use the ternary operator or short-circuiting (&& and || operators).

```
function Header() {
  const isAuthenticated = checkAuth();

  return (
    <nav>
      {/* if isAuth is true, show nothing. If false, show Logo  */
      {isAuthenticated || <Logo />}
      {/* if isAuth is true, show AuthenticatedApp. If false, show
      {isAuthenticated ? <AuthenticatedApp /> : <LoginScreen />}
      {/* if isAuth is true, show Footer. If false, show nothing *
      {isAuthenticated && <Footer />}
    </nav>
  );
}
```

Fragments are special components for displaying multiple components without adding an extra element to the DOM. They're ideal for conditional logic that has multiple adjacent components or elements.

```
/*
  We can improve the logic in the previous example.
  If isAuthenticated is true, how do we display both the Authentic
*/
function Header() {
  const isAuthenticated = checkAuth();

  return (
    <nav>
      <Logo />
      {/*
        We can render both components with a fragment.
        Fragments are very concise: <> </>
      */}
      {isAuthenticated ? (
        <>
          <AuthenticatedApp />
          <Footer />
        </>
      ) : (
        <Login />
      )}
    </nav>
  );
}
/*
  Note: An alternate syntax for fragments is React.Fragment:
  <React.Fragment>
     <AuthenticatedApp />
     <Footer />
  </React.Fragment>
*/
```

Lists and Keys

Use the **.map()** function to convert lists of data (arrays) into lists of elements.

`.map()` can be used for components as well as plain JSX elements.

```
function App() {
  const people = ['John', 'Bob', 'Fred'];
  // can interpolate returned list of elements in {}
  return (
    <ul>
      {/* we're passing each array element as props to Person */}
      {people.map(person => <Person name={person} />}
    </ul>
  );
}

function Person({ name }) {
  // we access the 'name' prop directly using object destructuring
  return <p>This person's name is: {name}</p>;
}
```

Each React element within a list of elements needs a special **key prop**. Keys are essential for React to be able to keep track of each element that is being iterated over with the `.map()` function.

React uses keys to performantly update individual elements when their data changes (instead of re-rendering the entire list).

Keys need to have unique values to be able to identify each of them according to their key value.

```
    { id: '6eAdl9', name: 'Fred' },
  ];

  return (
    <ul>
      {/* keys need to be primitive values, ideally a unique str
      {people.map(person =>
         <Person key={person.id} name={person.name} />
      )}
    </ul>
  );
}

// If you don't have some ids with your set of data that are uni

function App() {
  const people = ['John', 'Bob', 'Fred'];

  return (
    <ul>
      {/* use array element index for key */}
      {people.map((person, i) => <Person key={i} name={person} />
    </ul>
  );
}
```

Event Listeners and Handling Events

Listening for events on JSX elements versus HTML elements differs in a few important ways.

First, you cannot listen for events on React components – only on JSX elements. Adding a prop called `onClick`, for example, to a React component would just be another property added to the props object.

```
function handleToggleTheme() {
  // code to toggle app theme
}

/* In HTML, onclick is all lowercase, plus the event handler incl
<button onclick="handleToggleTheme()">
  Toggle Theme
</button>

/*
  In JSX, onClick is camelcase, like attributes / props.
  We also pass a reference to the function with curly braces.
*/
<button onClick={handleToggleTheme}>
  Toggle Theme
</button>
```

The most essential React events to know are `onClick`, `onChange`, and `onSubmit`.

- `onClick` handles click events on JSX elements (namely on buttons)
- `onChange` handles keyboard events (namely a user typing into an input or textarea)
- `onSubmit` handles form submissions from the user

```
function App() {
  function handleInputChange(event) {
    /* When passing the function to an event handler, like onChang
    const inputText = event.target.value; // text typed into the i
    const inputName = event.target.name; // 'email' from name attr
  }
```

```
    const eventType = event.type; // click
    const eventTarget = event.target; // <button>Submit</button>
  }

  function handleSubmit(event) {
    /*
     When we hit the return button, the form will be submitted, as
     We call event.preventDefault() to prevent the default form be
    */
    event.preventDefault();
    const formElements = event.target.elements; // access all elem
    const inputValue = event.target.elements.emailAddress.value; /
  }

  return (
    <form onSubmit={handleSubmit}>
      <input id="emailAddress" type="email" name="email" onChange=
      <button onClick={handleClick}>Submit</button>
    </form>
  );
}
```

Essential React Hooks

State and useState

The `useState` hook gives us state in a function component. **State** allows us to access and update certain values in our components over time.

Local component state is managed by the React hook `useState` which gives us both a state variable and a function that allows us to update it.

When we call `useState` we can give our state a default value by

```
import React from 'react';

/*
  How do you create a state variable?
  Syntax: const [stateVariable] = React.useState(defaultValue);
*/
function App() {
  const [language] = React.useState('JavaScript');
  /*
    We use array destructuring to declare state variable.
    Like any variable, we declare we can name it what we like (in
  */

  return <div>I am learning {language}</div>;
}
```

Note: Any hook in this section is from the React core library and can be imported individually.

```
import React, { useState } from "react";

function App() {
  const [language] = useState("javascript");

  return <div>I am learning {language}</div>;
}
```

`useState` also gives us a 'setter' function to update the state after it is created.

```
  */
  const [language, setLanguage] = React.useState("javascript");

  return (
    <div>
      <button onClick={() => setLanguage("python")}>
        Learn Python
      </button>
      {/*
        Why use an inline arrow function here instead of immediat
        If so, setLanguage would be called immediately and not wh
        */}
      <p>I am now learning {language}</p>
    </div>
  );
}

/*
 Note: whenever the setter function is called, the state updates,
 and the App component re-renders to display the new state.
 Whenever state is updated, the component will be re-rendered
*/
```

`useState` can be used once or multiple times within a single component. And it can accept primitive or object values to manage state.

```
function App() {
  const [language, setLanguage] = React.useState("python");
  const [yearsExperience, setYearsExperience] = React.useState(0);

  return (
    <div>
      <button onClick={() => setLanguage("javascript")}>
        Change language to JS
      </button>
```

```
        onChange={event => setYearsExperience(event.target.value)}
      />
      <p>I am now learning {language}</p>
      <p>I have {yearsExperience} years of experience</p>
    </div>
  );
}
```

If the new state depends on the previous state, to guarantee the update is done reliably we can use a function within the setter function that gives us the correct previous state.

```
/* We have the option to organize state using whatever is the most
function App() {
  const [developer, setDeveloper] = React.useState({
    language: "",
    yearsExperience: 0
  });

  function handleChangeYearsExperience(event) {
    const years = event.target.value;
    /* We must pass in the previous state object we had with the s
    setDeveloper({ ...developer, yearsExperience: years });
  }

  return (
    <div>
      {/* No need to get previous state here; we are replacing the
      <button
        onClick={() =>
          setDeveloper({
            language: "javascript",
            yearsExperience: 0
          })
        }
      >
        Change language to JS
```

```
          type="number"
          value={developer.yearsExperience}
          onChange={handleChangeYearsExperience}
        />
        <p>I am now learning {developer.language}</p>
        <p>I have {developer.yearsExperience} years of experience</p
      </div>
    );
}
```

If you are managing multiple primitive values, using `useState` multiple times is often better than using it once with an object. You don't have to worry about forgetting to combine the old state with the new state.

```
function App() {
  const [developer, setDeveloper] = React.useState({
    language: "",
    yearsExperience: 0,
    isEmployed: false
  });

  function handleToggleEmployment(event) {
    /* We get the previous state variable's value in the parameter
       We can name 'prevState' however we like.
    */
    setDeveloper(prevState => {
      return { ...prevState, isEmployed: !prevState.isEmployed };
      // It is essential to return the new state from this functio
    });
  }

  return (
    <button onClick={handleToggleEmployment}>Toggle Employment Sta
  );
}
```

Side effects and useEffect

`useEffect` lets us perform side effects in function components. So what are side effects?

Side effects are where we need to reach into the outside world. For example, fetching data from an API or working with the DOM.

They are actions that can change our component state in an unpredictable fashion (that have cause 'side effects').

`useEffect` accepts a callback function (called the 'effect' function), which will by default run every time there is a re-render.

It runs once our component mounts, which is the right time to perform a side effect in the component lifecycle.

```
/* What does our code do? Picks a color from the colors array and
import React, { useState, useEffect } from 'react';

function App() {
  const [colorIndex, setColorIndex] = useState(0);
  const colors = ["blue", "green", "red", "orange"];

  /*
    We are performing a 'side effect' since we are working with an
    We are working with the DOM, a browser API outside of React.
  */
  useEffect(() => {
    document.body.style.backgroundColor = colors[colorIndex];
  });
  /* Whenever state is updated, App re-renders and useEffect runs

  function handleChangeColor() {
    /* This code may look complex, but all it does is go to the ne
```

```
  return (
    <button onClick={handleChangeColor}>
      Change background color
    </button>
  );
}
```

To avoid executing the effect callback after each render, we provide a second argument, an empty array.

```
function App() {
  ...
  /*
    With an empty array, our button doesn't work no matter how many
    The background color is only set once, when the component first
  */
  useEffect(() => {
    document.body.style.backgroundColor = colors[colorIndex];
  }, []);

  /*
    How do we not have the effect function run for every state upda
  */

  return (
    <button onClick={handleChangeIndex}>
      Change background color
    </button>
  );
}
```

`useEffect` lets us conditionally perform effects with the dependencies array.

values in the array changes, the effect function runs again.

```
function App() {
  const [colorIndex, setColorIndex] = React.useState(0);
  const colors = ["blue", "green", "red", "orange"];

  /*
    Let's add colorIndex to our dependencies array
    When colorIndex changes, useEffect will execute the effect fun
  */
  useEffect(() => {
    document.body.style.backgroundColor = colors[colorIndex];
    /*
      When we use useEffect, we must think about what state values
      we want our side effect to sync with
    */
  }, [colorIndex]);

  function handleChangeIndex() {
    const next = colorIndex + 1 === colors.length ? 0 : colorIndex
    setColorIndex(next);
  }

  return (
    <button onClick={handleChangeIndex}>
      Change background color
    </button>
  );
}
```

`useEffect` lets us unsubscribe from certain effects by returning a function at the end.

```
function MouseTracker() {
  const [mousePosition, setMousePosition] = useState({ x: 0, y: 0
```

```
    // .addEventListener() sets up an active listener...
    window.addEventListener("mousemove", handleMouseMove);

    /* ...So when we navigate away from this page, it needs to be
       removed to stop listening. Otherwise, it will try to set
       state in a component that doesn't exist (causing an error)

     We unsubscribe any subscriptions / listeners w/ this 'cleanup'
     */
    return () => {
      window.removeEventListener("mousemove", handleMouseMove);
    };
  }, []);

function handleMouseMove(event) {
   setMousePosition({
     x: event.pageX,
     y: event.pageY
   });
}

  return (
    <div>
      <h1>The current mouse position is:</h1>
      <p>
        X: {mousePosition.x}, Y: {mousePosition.y}
      </p>
    </div>
  );
}
```

`useEffect` is the hook to use when you want to make an HTTP request (namely, a GET request when the component mounts).

Note that handling promises with the more concise async/await syntax requires creating a separate function. (Why? The effect callback function cannot be async.)

```
// Using .then() callback functions to resolve promise
function App() {
  const [user, setUser] = React.useState(null);

  React.useEffect(() => {
    fetch(endpoint)
      .then(response => response.json())
      .then(data => setUser(data));
  }, []);
}

// Using async / await syntax to resolve promise:
function App() {
  const [user, setUser] = React.useState(null);
  // cannot make useEffect callback function async
  React.useEffect(() => {
    getUser();
  }, []);

  // We must apply async keyword to a separate function
  async function getUser() {
    const response = await fetch(endpoint);
    const data = await response.json();
    setUser(data);
  }
}
```

Refs and useRef

Refs are a special attribute that are available on all React components. They allow us to create a reference to a given element / component when the component mounts.

`useRef` allows us to easily use React refs. We call useRef (at the top of the component) and attach the returned value to the element's ref attribute to refer to it.

(mutate) the element's properties or can call any available methods on that element (like `.focus()` to focus an input).

```
function App() {
  const [query, setQuery] = React.useState("react hooks");
  /* We can pass useRef a default value.
     We don't need it here, so we pass in null to reference an emp
  */
  const searchInput = useRef(null);

  function handleClearSearch() {
    /*
      .current references the input element upon mount
      useRef can store basically any value in its .current property
    */
    searchInput.current.value = "";
    searchInput.current.focus();
  }

  return (
    <form>
      <input
        type="text"
        onChange={event => setQuery(event.target.value)}
        ref={searchInput}
      />
      <button type="submit">Search</button>
      <button type="button" onClick={handleClearSearch}>
        Clear
      </button>
    </form>
  );
}
```

Hooks and Performance

components are rendered.

In particular, it performs a process called **memoization** that helps us prevent our components from re-rendering when they do not need to do so (see React.useMemo for more complete definition of memoization).

`React.memo` helps most with preventing lists of components from being re-rendered when their parent components re-render.

```
/*
  In the following application, we are keeping track of our progra
*/

function App() {
  const [skill, setSkill] = React.useState('')
  const [skills, setSkills] = React.useState([
    'HTML', 'CSS', 'JavaScript'
  ])

  function handleChangeInput(event) {
    setSkill(event.target.value);
  }

  function handleAddSkill() {
    setSkills(skills.concat(skill))
  }

  return (
    <>
      <input onChange={handleChangeInput} />
      <button onClick={handleAddSkill}>Add Skill</button>
      <SkillList skills={skills} />
    </>
  );
}

/* But the problem, if you run this code yourself, is that when we
```

```
  console.log('rerendering');
  return (
    <ul>
    {skills.map((skill, i) => <li key={i}>{skill}</li>)}
    </ul>
  )
})

export default App
```

Callback functions and useCallback

`useCallback` is a hook that is used for improving our component performance. **Callback functions** are the name of functions that are "called back" within a parent component.

The most common usage is to have a parent component with a state variable, but you want to update that state from a child component. What do you do? You pass down a callback function to the child from the parent. That allows us to update state in the parent component.

`useCallback` functions in a similar way as `React.memo`. It memoizes callback functions, so it is not recreated on every re-render. Using `useCallback` correctly can improve the performance of our app.

```
/* Let's keep the exact same App as above with React.memo, but ac

function App() {
  const [skill, setSkill] = React.useState('')
  const [skills, setSkills] = React.useState([
    'HTML', 'CSS', 'JavaScript'
  ])
```

```
  function handleAddSkill() {
    setSkills(skills.concat(skill))
  }

  function handleRemoveSkill(skill) {
    setSkills(skills.filter(s => s !== skill))
  }

  /* Next, we pass handleRemoveSkill down as a prop, or since thi
  return (
    <>
      <input onChange={handleChangeInput} />
      <button onClick={handleAddSkill}>Add Skill</button>
      <SkillList skills={skills} handleRemoveSkill={handleRemoveS
    </>
  );
}

/* When we try typing in the input again, we see rerendering in t

What is happening is the handleRemoveSkill callback function is b

To fix our app, replace handleRemoveSkill with:

const handleRemoveSkill = React.useCallback((skill) => {
  setSkills(skills.filter(s => s !== skill))
}, [skills])

Try it yourself!
*/
const SkillList = React.memo(({ skills, handleRemoveSkill }) => {
  console.log('rerendering');
  return (
    <ul>
    {skills.map(skill => <li key={skill} onClick={() => handleRem
    </ul>
  )
})


export default App
```

Memoization and useMemo

`useMemo` is very similar to `useCallback` and is for improving performance. But instead of being for callbacks, it is for storing the results of expensive calculations

`useMemo` allows us to **memoize**, or remember the result of expensive calculations when they have already been made for certain inputs.

Memoization means that if a calculation has been done before with a given input, there's no need to do it again, because we already have the stored result of that operation.

`useMemo` returns a value from the computation, which is then stored in a variable.

```
/* Building upon our skills app, let's add a feature to search thr
*/

function App() {
  const [skill, setSkill] = React.useState('')
  const [skills, setSkills] = React.useState([
    'HTML', 'CSS', 'JavaScript', ...thousands more items
  ])

  function handleChangeInput(event) {
    setSkill(event.target.value);
  }

  function handleAddSkill() {
    setSkills(skills.concat(skill))
  }

  const handleRemoveSkill = React.useCallback((skill) => {
    setSkills(skills.filter(s => s !== skill))
  }, [skills])
```

```
      <SearchSkills skills={skills} />
      <input onChange={handleChangeInput} />
      <button onClick={handleAddSkill}>Add Skill</button>
      <SkillList skills={skills} handleRemoveSkill={handleRemoveSk
    </>
  );
}

/* Let's imagine we have a list of thousands of skills that we wan
function SearchSkills() {
  const [searchTerm, setSearchTerm] = React.useState('');

  /* We use React.useMemo to memoize (remember) the returned value
  const searchResults = React.useMemo(() => {
    return skills.filter((s) => s.includes(searchTerm));
  }), [searchTerm]);

  function handleSearchInput(event) {
    setSearchTerm(event.target.value);
  }

  return (
    <>
    <input onChange={handleSearchInput} />
    <ul>
      {searchResults.map((result, i) => <li key={i}>{result}</li>
    </ul>
    </>
  );
}


export default App
```

Advanced React Hooks

Context and useContext

```
/*
  React Context helps us avoid creating multiple duplicate props.
  This pattern is also called props drilling.
*/

/* In this app, we want to pass the user data down to the Header c
function App() {
  const [user] = React.useState({ name: "Fred" });

  return (
    // First 'user' prop
    <Main user={user} />
  );
}

const Main = ({ user }) => (
  <>
    {/* Second 'user' prop */}
    <Header user={user} />
    <div>Main app content...</div>
  </>
);

const Header = ({ user }) => <header>Welcome, {user.name}!</header>
```

Context is helpful for passing props down multiple levels of child components from a parent component.

```
/*
  Here is the previous example rewritten with Context.
  First we create context, where we can pass in default values
  We call this 'UserContext' because we're passing down user data
*/
const UserContext = React.createContext();
```

```
  return (
    {/*
      We wrap the parent component with the Provider property
      We pass data down the component tree on the value prop
     */}
    <UserContext.Provider value={user}>
      <Main />
    </UserContext.Provider>
  );
}

const Main = () => (
  <>
    <Header />
    <div>Main app content</div>
  </>
);

/*
  We can't remove the two 'user' props. Instead, we can just use t
*/
const Header = () => (
    {/* We use a pattern called render props to get access to the
    <UserContext.Consumer>
      {user => <header>Welcome, {user.name}!</header>}
    </UserContext.Consumer>
);
```

The `useContext` hook allows us to consume context in any function component that is a child of the Provider, instead of using the render props pattern.

```
function Header() {
  /* We pass in the entire context object to consume it and we ca
  const user = React.useContext(UserContext);
```

Reducers and useReducer

Reducers are simple, predictable (pure) functions that take a previous state object and an action object and return a new state object.

```
/* This reducer manages user state in our app: */

function userReducer(state, action) {
  /* Reducers often use a switch statement to update state in one v

  switch (action.type) {
    /* If action.type has the string 'LOGIN' on it, we get data fr
    case "LOGIN":
      return {
        username: action.payload.username,
        email: action.payload.email
        isAuth: true
      };
    case "SIGNOUT":
      return {
        username: "",
        email: "",
        isAuth: false
      };
    default:
      /* If no case matches the action received, return the previo
      return state;
  }
}
```

Reducers are a powerful pattern for managing state that is used in the popular state management library Redux (commonly used with

Reducers can be used in React with the `useReducer` hook in order to manage state across our app, as compared to useState (which is for local component state).

`useReducer` can be paired with `useContext` to manage data and pass it around components easily.

Thus `useReducer` + `useContext` can be an entire state management system for our apps.

```
const initialState = { username: "", isAuth: false };

function reducer(state, action) {
  switch (action.type) {
    case "LOGIN":
      return { username: action.payload.username, isAuth: true };
    case "SIGNOUT":
      // could also spread in initialState here
      return { username: "", isAuth: false };
    default:
      return state;
  }
}

function App() {
  // useReducer requires a reducer function to use and an initial
  const [state, dispatch] = useReducer(reducer, initialState);
  // we get the current result of the reducer on 'state'

  // we use dispatch to 'dispatch' actions, to run our reducer
  // with the data it needs (the action object)
  function handleLogin() {
    dispatch({ type: "LOGIN", payload: { username: "Ted" } });
  }

  function handleSignout() {
    dispatch({ type: "SIGNOUT" });
  }
```

```
        Current user: {state.username}, isAuthenticated: {state.isA
        <button onClick={handleLogin}>Login</button>
        <button onClick={handleSignout}>Signout</button>
      </>
    );
}
```

Writing custom hooks

Hooks were created to easily reuse behavior between components, similar to how components were created to reuse structure across our application.

Hooks let us add custom functionality to our apps that suits our needs and can be combined with all the existing hooks that we've covered.

Hooks can also be included in third-party libraries for the sake of all React developers. There are many great React libraries that provide custom hooks such as `@apollo/client`, `react-query`, `swr` and more.

```
/* Here is a custom React hook called useWindowSize that I wrote i

import React from "react";

export default function useWindowSize() {
  const isSSR = typeof window !== "undefined";
  const [windowSize, setWindowSize] = React.useState({
    width: isSSR ? 1200 : window.innerWidth,
    height: isSSR ? 800 : window.innerHeight,
  });

  function changeWindowSize() {
    setWindowSize({ width: window.innerWidth, height: window.innerH
```

```
    return () => {
      window.removeEventListener("resize", changeWindowSize);
    };
  }, []);

  return windowSize;
}

/* To use the hook, we just need to import it where we need, call

// components/Header.js

import React from "react";
import useWindowSize from "../utils/useWindowSize";

function Header() {
  const { width } = useWindowSize();

  return (
    <div>
      {/* visible only when window greater than 500px */}
      {width > 500 && (
        <>
          Greater than 500px!
        </>
      )}
      {/* visible at any window size */}
      <p>I'm always visible</p>
    </div>
  );
}
```

Rules of hooks

There are two essential rules of using React hooks that we cannot violate for them to work properly:

- Hooks can only be called at the top of components (they cannot be in conditionals, loops, or nested functions)

Conclusion

There are other worthwhile concepts you can learn, but if you commit to learning the concepts covered in this cheatsheet, you'll have a great grasp of the most important and powerful parts of the React library.

Want to keep this guide for future reference?

Download a complete PDF version of this cheatsheet here.

Reed Barger
React developer who loves to make incredible apps. Showing you how at ReactBootcamp.com

---

thousands of freeCodeCamp study groups around the world.

Donations to freeCodeCamp go toward our education initiatives and help pay for servers, services, and staff.

You can make a tax-deductible donation here.

Trending Guides

10 to the Power of 0	Recursion
Git Reset to Remote	ISO File
R Value in Statistics	ADB
What is Economics?	MBR VS GPT
Module Exports	Debounce
Python VS JavaScript	Helm Chart
Model View Controller	80-20 Rule
React Testing Library	OSI Model
ASCII Table Chart	HTML Link Code
Data Validation	SDLC
Inductive VS Deductive	JavaScript Keycode List
JavaScript Empty Array	JavaScript Reverse Array
Best Instagram Post Time	How to Screenshot on Mac
Garbage Collection in Java	How to Reverse Image Search
Auto-Numbering in Excel	Ternary Operator JavaScript

Our Nonprofit