The React Cheatsheet for 2022

Table of Contents

- React useCallback Hook
- React useMemo Hook

React Elements

React elements are written just like regular HTML elements. You can write any valid HTML element in React.

```
<h1>My Header</h1>
<p>My paragraph>
<button>My button</button>
```

We write React elements using a feature called *JSX*.

However, because JSX is really just JavaScript functions (and not HTML), the syntax is a bit different.

Unlike HTML, single-tag elements (like the img element), must be self-closing. They must end in a forward slash `/`:

```
<img src="my-image.png" />
<br />
<hr />
```

React Element Attributes

Additionally, JSX requires a different syntax for its attributes.

Since JSX is really JavaScript and JavaScript uses a camelcase naming convention (that is, “camelCase”), attributes are written differently than HTML.

The most common example is the `class` attribute, which we write as `className`.

```
<div className="container"></div>
```

React Element Styles

To apply inline styles, instead of using double quotes (“”), we use two sets of curly braces.

Inline styles are not written as plain strings, but as properties on objects:

```
<h1 style={{ fontSize: 24, margin: '0 auto', textAlign: 'center' }}>My he
```

React Fragments

React also gives us an element called a *fragment*.

React requires that all returned elements be returned within a single “parent” component.

For example, we can’t return two sibling elements, like an h1 and a paragraph from a component:

```
// this syntax is invalid
function MyComponent() {
  return (
    <h1>My header</h1>
    </p>My paragraph</p>
  );
}
```

If we don’t want to wrap our elements in a container element like a div, we can use a fragment:

```
// valid syntax
function MyComponent() {
  return (
    <>
      <h1>My header</h1>
      </p>My paragraph</p>
    </>
  );
}
```

We can write fragments in a regular or shorthand syntax: <React.Fragment></React.Fragment> or <></>.

React Components

We can organized groups of elements into React components.

A basic function component is written similarly to a regular JavaScript function with a couple of differences.

1. Component names must start with a capital letter (that is, MyComponent, instead of myComponent)
2. Components, unlike JavaScript functions, must return JSX.

Here is the basic syntax of a React function component:

```
function App() {
  return (
     <div>Hello world!</div>
  );
}
```

React Props

React components can accept data passed to them called *props*.

Props are passed from the parent component to a child component.

Here we are passing a prop `name` from App to the User component.

```
function App() {
  return <User name="John Doe" />
}

function User(props) {
  return <h1>Hello, {props.name}</h1>; // Hello, John Doe!
}
```

Props is an object, so we can select the `name` prop within `User` to get its value.

> To embed any dynamic value (that is, a variable or expression) within JSX, you must wrap it in curly braces.

Since we are only using the `name` property on the props object, we can make our code simpler with object destructuring:

```
function App() {
  return <User name="John Doe" />
}

function User({ name }) {
  return <h1>Hello, {name}!</h1>; // Hello, John Doe!
}
```

Any JavaScript value can be passed as a prop, including other elements and components.

React Children Props

Props can also be passed by placing data between the opening and closing tags of a component.

Props that are passed this way are placed on the `children` property.

```
function App() {
  return (
   <User>
     <h1>Hello, John Doe!</h1>
   </User>
  );
}

function User({ children }) {
  return children; // Hello, John Doe!
}
```

React Conditionals

React components and elements can be conditionally displayed.

One approach is to create a separate return with an if-statement.

```
function App() {
        const isAuthUser = useAuth();

  if (isAuthUser) {
    // if our user is authenticated, let them use the app
    return <AuthApp />;
  }

  // if user is not authenticated, show a different screen
  return <UnAuthApp />;
}
```

If you want to write a conditional within a return statement, however, you must use a conditional that resolves to a value.

To use the ternary operator, wrap the entire conditional in curly braces.

```
function App() {
        const isAuthUser = useAuth();

  return (
    <>
      <h1>My App</h1>
      {isAuthUser ? <AuthApp /> : <UnAuthApp />}
    </>
  )
}
```

React Lists

Lists of React components can be output using the `.map()` function.

`.map()` allows us to loop over arrays of data and output JSX.

Here we are outputting a list of soccer players using the SoccerPlayer component.

```
function SoccerPlayers() {
  const players = ["Messi", "Ronaldo", "Laspada"];

  return (
    <div>
      {players.map((playerName) => (
        <SoccerPlayer key={playerName} name={playerName} />
      ))}
    </div>
  );
}
```

Whenever you are looping over an array of data, you must include the *key* prop on the element or component over which you are looping.

Additionally, this key prop must be given a unique value, not just an element index.

In the example above, we are using a value which we know to be unique, which is the `playerName`.

React Context

React context allows us to pass data to our component tree without using props.

The problem with props is that sometimes we pass them through components that don't need to receive them. This problem is called *props drilling*.

Here is a oversimplified example of passing props through a `Body` component that doesn't need it:

```
function App() {
  return (
    <Body name="John Doe" />
  );
}

function Body({ name }) {
  return (
    <Greeting name={name} />
  );
}

function Greeting({ name }) {
  return <h1>Welcome, {name}</h1>;
}
```

> Before using Context, its best to see if our components can be better organized to avoid passing props through components that don't need it.

To use Context, we use the `createContext` function from React.

We can call it with an initial value to be put on context.

The created context includes a `Provider` and a `Consumer` property, which are each components.

We wrap the Provider around the component tree that we want to pass the given value down. Next, we place the Consumer in the component we want to consume the value.

```
import { createContext } from 'react';

const NameContext = createContext('');

function App() {
  return (
    <NameContext.Provider value="John Doe">
      <Body />
    <NameContext.Provider>
  );
}

function Body() {
  return <Greeting />;
}

function Greeting() {
  return (
    <NameContext.Consumer>
      {name => <h1>Welcome, {name}</h1>}
    </NameContext.Consumer>
  );
}
```

React Hooks

React hooks were introduced in React version 16.8 as a way to easily add reusable, stateful logic to React function components.

Hooks let us use all the features that were previously only available in class components.

Additionally, we can create our own custom hooks that give our app custom functionality.

Many React hooks were added to the core React library as well. We are going to cover the 6 essential hooks you absolutely need to know:

- useState
- useEffect
- useRef
- useContext
- useCallback
- useMemo

React useState Hook

`useState` does exactly what it says—it allows us to use stateful values in function components.

useState is used instead of a simple variable because when state is updated, our component re-renders, usually to display that updated value.

Like all hooks, we call `useState` at the top of our component and can pass it an initial value to put on its state variable.

We use array destructuring on the value returned from `useState` to access (1) the stored state and (2) a function to update that state.

```
import { useState } from 'react';

function MyComponent() {
  const [stateValue, setStateValue] = useState(initialValue);
}
```

A basic example of using `useState` is to increment a counter.

We can see the current count from the `count` variable and can increment the state by passing `count + 1` to the `setCount` function.

```
import { useState } from 'react';

function Counter() {
  const [count, setCount] = useState(0);

  function updateCount() {
    setCount(count + 1);
  }

  return <button onClick={updateCount}>Count is: {count}</button>;
}
```

React useEffect Hook

If we want to interact with the “outside world”, such as using an API, we use the `useEffect` hook.

useEffect is used to perform a side effect, which means to perform an operation that exists outside of our app that doesn’t have a predictable result.

The basic syntax of useEffect requires a function as a first argument and an array as the second argument.

```
import { useEffect } from 'react';

function MyComponent() {
   useEffect(() => {
     // perform side effect here
   }, []);
}
```

If we want to fetch data, we would use `useEffect`, such as in fetching and displaying a list of posts:

```
import { useEffect } from 'react';

function PostList() {
         const [posts, setPosts] = useState([]);

   useEffect(() => {
          fetch('https://jsonplaceholder.typicode.com/posts')
       .then(response => response.json())
       .then(posts => setPosts(posts));
   }, []);

   return posts.map(post => <Post key={post.id} post={post} />
}
```

If we need to use a value that comes from outside the effect function, it must be included in the dependencies array.

If that value changes, the effect function will be re-executed.

For example, here is a bit of code that adds or removes the class “overflow-hidden” to the body element whenever the mobile menu is opened or closed.

```
function Mobile({ open }) {
  useEffect(() => {
    const body = document.querySelector("#__next");

    if (open) {
      body.classList.add("overflow-hidden");
    } else {
      body.classList.remove("overflow-hidden");
    }
  }, [open]);
  // ...
}
```

React useRef

`useRef` allows us to get direct access to a JSX element.

To use `useRef`, call it, get the returned value, and put it on the `ref` prop for a given React element.

> Refs do not have a built-in prop on components, only React elements.

Here is the basic syntax for `useRef`:

```
import { useRef } from 'react';

function MyComponent() {
  const ref = useRef();

  return <div ref={ref} />
}
```

Once a ref is attached to a given element, we can use the value stored on `ref.current` to access the element itself.

For example, if we wanted to write some code that focuses a search input when the users use the key combination Control + K.

```
import { useWindowEvent } from "@mantine/hooks";
import { useRef } from "react";

function Header() {
      const inputRef = useRef();

  useWindowEvent("keydown", (event) => {
    if (event.code === "KeyK" && event.ctrlKey) {
      event.preventDefault();
      inputRef.current.focus();
    }
  });

  return <input ref={inputRef} />
}
```

React useContext

`useContext` provides an easier way of consuming context than using the standard Context.Consumer component.

The syntax involves passing the entire Context object that we want to consume into `useContext`. The returned value is the value passed down to Context.

```
import { useContext } from 'react';

function MyComponent() {
  const value = useContext(Context);

  // ...
}
```

To rewrite our example from earlier, using the `useContext` hook:

```
import { createContext, useContext } from 'react';

const NameContext = createContext('');

function App() {
  return (
    <NameContext.Provider value="John Doe">
      <Body />
    <NameContext.Provider>
  );
}

function Body() {
  return <Greeting />;
}

function Greeting() {
    const name = useContext(NameContext);

  return (
    <h1>Welcome, {name}</h1>
  );
}
```

React useCallback

`useCallback` is a hook that we use to help with our app's performance.

Specifically, it prevents functions from being recreated every time our component re-renders, which can hurt the performance of our app.

If we go back to our `PlayerList` example from earlier and add the ability to add players to our array, when we pass down a function to remove them (`handleRemovePlayer`) via props, the function will be recreated every time.

The way to fix this is to wrap our callback function in `useCallback` and to include its one argument `player` in the dependencies array:

```
function App() {
  const [player, setPlayer] = React.useState("");
  const [players, setPlayers] = React.useState(["Messi", "Ronaldo",

  function handleChangeInput(event) {
    setPlayer(event.target.value);
  }
  function handleAddPlayer() {
    setPlayers(players.concat(player));
  }
  const handleRemovePlayer = useCallback(player => {
    setPlayers(players.filter((p) => p !== player));
  }, [players])

  return (
    <>
      <input onChange={handleChangeInput} />
      <button onClick={handleAddPlayer}>Add Player</button>
      <PlayerList players={players} handleRemovePlayer={handleRemov
    </>
  );
}

function PlayerList({ players, handleRemovePlayer }) {
  return (
    <ul>
      {players.map((player) => (
        <li key={player} onClick={() => handleRemovePlayer(player)}
          {player}
        </li>
      ))}
    </ul>
  );
}
```

React useMemo

`useMemo` is another performance hook that allows us to 'memoize' a given operation.

Memoization makes it possible to remember the result of expensive calculations when they have already been made so we don't have to make them again.

Like `useEffect` and `useCallback` , `useMemo` accepts a callback function and a dependencies array.

Unlike both of these functions, however, `useMemo` is intended to return a value.

> You must return the value either explicitly with the `return` keyword or implicitly but using the arrow function shorthand (seen below).

A real-world example of `useMemo` comes from the mdx-bundler documentation. `mdx-bundler` is a library for converting .mdx files into React components.

Here it uses `useMemo` to convert a raw string of code into a React component.

```
import * as React from 'react'
import {getMDXComponent} from 'mdx-bundler/client'

function Post({code, frontmatter}) {
  const Component = React.useMemo(() => getMDXComponent(code), [code]);

  return (
    <>
      <header>
        <h1>{frontmatter.title}</h1>
        <p>{frontmatter.description}</p>
      </header>
      <main>
        <Component />
      </main>
    </>
  )
}
```

The reason for doing so is to prevent the `Component` value from being recreated unnecessarily when the component re-renders.

`useMemo` therefore will only execute its callback function if the `code` dependency changes.