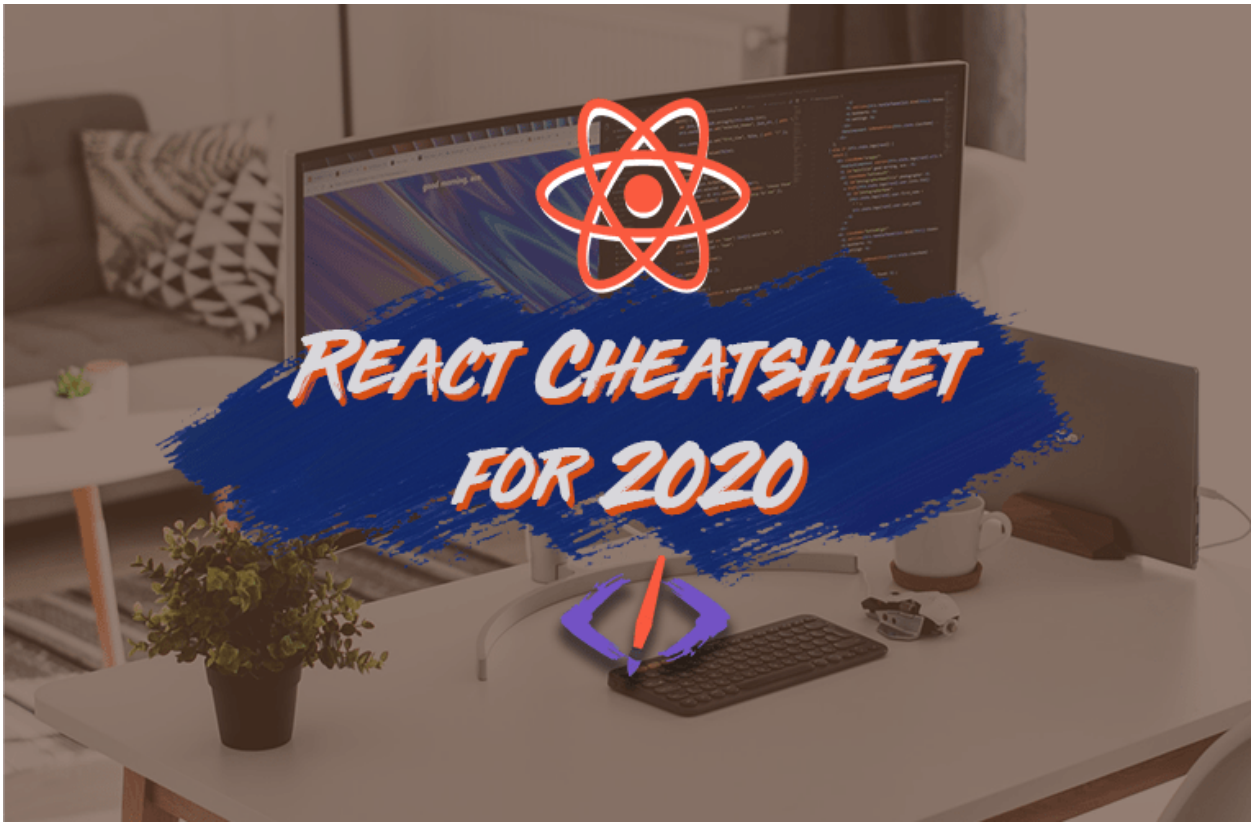
The React for 2020 Cheatsheet



Core Concepts

- Elements and JSX
- Components and Props
- Lists and Keys
- Events and Event Handlers

React Hooks

- State and useState
- Side Effects and useEffect
- Performance and useCallback
- Memoization and useMemo

- Refs and useRef

Advanced Hooks

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

Core Concepts

Elements and JSX

- The basic syntax for a React element

```
// In a nutshell, JSX allows us to write HTML in our JS
// JSX can use any valid html tags (i.e. div/span, h1-h6, form/input, etc)
<div>Hello React</div>
```

- JSX elements are expressions

```
// as an expression, JSX can be assigned to variables...
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 nest expressions

```
const year = 2020;
// we can insert primitive JS values in curly braces: {}
const greeting = <div>Hello React in {year}</div>;
// trying to insert objects will result in an error
```

- JSX allows us to nest elements

```
// to write JSX on multiple lines, wrap in parentheses: ()
const greeting = (
  // div is the parent element
  <div>
    {/* h1 and p are child elements */}
    <h1>Hello!</h1>
    <p>Welcome to React</p>
  </div>
);
// 'parents' and 'children' are how we describe JSX elements in relation
// to one another, like we would talk about HTML elements
```

- HTML and JSX have a slightly different syntax

```
// Empty div is not <div></div> (HTML), but <div/> (JSX)
<div/>

// A single tag element like input is not <input> (HTML), but <input/> (JSX)
<input name="email" />

// Attributes are written in camelcase for JSX (like JS variables
<button className="submit-button">Submit</button> // not 'class' (HTML)
```

- The most basic React app requires three things:
  - ReactDOM.render() to render our app
  - A JSX element (called a root node in this context)
  - A DOM element within which to mount the app (usually a div with an id of root in an index.html file)

```
// imports needed if using NPM package; not if from CDN links
import React from "react";
import ReactDOM from "react-dom";

const greeting = <h1>Hello React</h1>;

// ReactDOM.render(root node, mounting point)
ReactDOM.render(greeting, document.getElementById("root"));
```

Components and Props

- The syntax for a basic React component

```
import React from "react";

// 1st component type: function component
function Header() {
  // function components must be capitalized unlike normal JS functions
  // note the capitalized name here: 'Header'
  return <h1>Hello React</h1>;
}

// function components with arrow functions are also valid
const Header = () => <h1>Hello React</h1>;

// 2nd component type: class component
// (classes are another type of function)
class Header extends React.Component {
  // class components have more boilerplate (with extends and render method)
  render() {
    return <h1>Hello React</h1>;
  }
}
```

- How components are used

```
// do we call these function components like normal functions?

// 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>
```

- Components can be reused across our app

```
// for example, this Header component can be reused in any app page

// this component shown for the '/' route
function IndexPage() {
  return (
    <>
      <Header />
      <Hero />
```

```
      <Footer />
    </>
  );
}

// shown for the '/about' route
function AboutPage() {
  return (
    <>
      <Header />
      <About />
      <Testimonials />
      <Footer />
    </>
  );
}
```

- Data can be dynamically passed to components with props

```
// What if we want to pass data to our component from a parent?
// I.e. to pass a user's name to display in our Header?

const username = "John";

// we add custom 'attributes' called props
ReactDOM.render(
  <Header username={username} />,
  document.getElementById("root")
);
// we called this prop 'username', but can use any valid JS identifier

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

- Props must never be directly changed (mutated)

```
// Components must ideally be 'pure' functions.
// That is, for every input, we be able to expect the same output

// we cannot do the following with props:
function Header(props) {
  // we cannot mutate the props object, we can only read from it
  props.username = "Doug";
```

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

- Children props are 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 'children'

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 components:
function IndexPage() {
  return (
    <Layout>
      <Header />
      <Hero />
      <Footer />
    </Layout>
  );
}

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

- Conditionally displaying components with ternaries and short-circuiting

```
// if-statements are fine to conditionally show , however...
// ...only ternaries (seen below) allow us to insert these conditionals
// in JSX, however
function Header() {
  const isAuthenticated = checkAuth();

  return (
    <nav>
      <Logo />
```

```
      {/* if isAuth is true, show AuthLinks. If false, Login  */}
      {isAuthenticated ? <AuthLinks /> : <Login />}
      {/* if isAuth is true, show Greeting. If false, nothing. */}
      {isAuthenticated && <Greeting />}
    </nav>
  );
}
```

- Fragments are special components for displaying multiple components without adding an extra element to the DOM
    - Fragments are ideal for conditional logic

```
// we can improve the logic in the previous example
// if isAuthenticated is true, how do we display both AuthLinks and Greeting?
function Header() {
  const isAuthenticated = checkAuth();

  return (
    <nav>
      <Logo />
      {/* we can render both components with a fragment */}
      {/* fragments are very concise: <> </> */}
      {isAuthenticated ? (
        <>
          <AuthLinks />
          <Greeting />
        </>
      ) : (
        <Login />
      )}
    </nav>
  );
}
```

Lists and Keys

- Use .map() to convert lists of data (arrays) into lists of elements

```
const people = ["John", "Bob", "Fred"];
const peopleList = people.map((person) => <p>{person}</p>);
```

- .map() is also used for components as well as 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 */}
      {people.map((person) => (
        <Person name={person} />
      ))}
    </ul>
  );
}

function Person({ name }) {
  // gets 'name' prop using object destructuring
  return <p>this person's name is: {name}</p>;
}
```

- Each React element iterated over 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 map
  - Without keys, it is harder for it to figure out how elements should be updated when data changes
  - Keys should be unique values to represent the fact that these elements are separate from one another

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

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

// If you don't have ids with your set of data or unique primitive values,
// you can use the second parameter of .map() to get each elements index
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>
  );
}
```

Events and Event Handlers

- Events in React and HTML are slightly different

```
// Note: most event handler functions start with 'handle'
function handleToggleTheme() {
  // code to toggle app theme
}

// in html, onclick is all lowercase
<button onclick="handleToggleTheme()">
  Submit
</button>

// in JSX, onClick is camelcase, like attributes / props
// we also pass a reference to the function with curly braces
<button onClick={handleToggleTheme}>
  Submit
</button>
```

- The most essential React events to know are onClick and onChange
  - onClick handles click events on JSX elements (namely buttons)
  - onChange handles keyboard events (namely inputs)

```
function App() {
  function handleChange(event) {
    // when passing the function to an event handler, like onChange
    // we get access to data about the event (an object)
    const inputText = event.target.value;
    const inputName = event.target.name; // myInput
    // we get the text typed in and other data from event.target
  }

  function handleSubmit() {
    // on click doesn't usually need event data
  }
```

```
  return (
    <div>
      <input type="text" name="myInput" onChange={handleChange} />
      <button onClick={handleSubmit}>Submit</button>
    </div>
  );
}
```

React Hooks

State and useState

- useState gives us local state in a function component

```
import React from "react";

// create state variable
// syntax: const [stateVariable] = React.useState(defaultValue);
function App() {
  const [language] = React.useState("javascript");
  // we use array destructuring to declare state variable

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

- Note: Any hook in this section is from the React package 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 it creates

```
function App() {
  // the setter function is always the second destructured value
  const [language, setLanguage] = React.useState("python");
  // the convention for the setter name is 'setStateVariable'

  return (
```

```
    <div>
      {/*  why use an arrow function here instead onClick={setterFn()} ? */}
      <button onClick={() => setLanguage("javascript")}>
        Change language to JS
      </button>
      {/*  if not, setLanguage would be called immediately and not on click */}
      <p>I am now learning {language}</p>
    </div>
  );
}

// note that whenever the setter function is called, the state updates,
// and the App component re-renders to display the new state
```

- useState can be used once or multiple times within a single component

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

- useState can accept primitive or object values to manage state

```
// we have the option to organize state using whatever is the
// most appropriate data type, according to the data we're tracking
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 spread operator
    setDeveloper({ ...developer, yearsExperience: years });
  }

  return (
    <div>
      {/* no need to get prev state here; we are replacing the entire object */}
      <button
        onClick={() =>
          setDeveloper({
            language: "javascript",
            yearsExperience: 0,
          })
        }
      >
        Change language to JS
      </button>
      {/* we can also pass a reference to the function */}
      <input
        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 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

```
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 parameters
    // we can name 'prevState' however we like
    setDeveloper((prevState) => {
      return { ...prevState, isEmployed: !prevState.isEmployed };
      // it is essential to return the new state from this function
    });
  }

  return (
```

```
    <button onClick={handleToggleEmployment}>Toggle Employment Status</button>
  );
}
```

Side effects and useEffect

- useEffect lets us perform side effects in function components. 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.
  - Side effects 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
- useEffect 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 makes it the background color
function App() {
  const [colorIndex, setColorIndex] = React.useState(0);
  const colors = ["blue", "green", "red", "orange"];

  // we are performing a 'side effect' since we are working with an API
  // 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 handleChangeIndex() {
    const next = colorIndex + 1 === colors.length ? 0 : colorIndex + 1;
    setColorIndex(next);
  }

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

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

```
function App() {
  ...
  // now our button doesn't work no matter how many times we click it...
  useEffect(() => {
    document.body.style.backgroundColor = colors[colorIndex];
  }, []);
  // the background color is only set once, upon mount

  // how do we not have the effect function run for every state update...
  // but still have it work whenever the button is clicked?

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

- useEffect lets us conditionally perform effects with the dependencies array
    - The dependencies array is the second argument and if any one of the values in the array changes, the effect function runs again

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

  // we add colorIndex to our dependencies array
  // when colorIndex changes, useEffect will execute the effect fn again
  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 + 1;
    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 });

  React.useEffect(() => {
    // .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 function'
    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>
  );
}

// Note: we could extract the reused logic in the callbacks to
// their own function, but I believe this is more readable
```

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

```
const endpoint = "<https://api.github.com/users/codeartistryio>";

// with promises:
function App() {
  const [user, setUser] = React.useState(null);
```

```
  React.useEffect(() => {
    // promises work in callback
    fetch(endpoint)
      .then((response) => response.json())
      .then((data) => setUser(data));
  }, []);
}

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

  // instead, use async / await in separate function, then call
  // function back in useEffect
  async function getUser() {
    const response = await fetch("<https://api.github.com/codeartistryio>");
    const data = await response.json();
    setUser(data);
  }
}
```

Performance and useCallback

- useCallback is a hook that is used for improving our component performance
  - If you have a component that re-renders frequently, useCallback prevents callback functions within the component from being recreated every single time the component re-renders (which mean the function component re-runs)
  - useCallback re-runs only when one of it's dependencies changes

```
// in Timer, we are calculating the date and putting it in state a lot
// this results in a re-render for every state update

// we had a function handleIncrementCount to increment the state 'count'...
function Timer() {
  const [time, setTime] = React.useState();
  const [count, setCount] = React.useState(0);

  // ... but unless we wrap it in useCallback, the function is
  // recreated for every single re-render (bad performance hit)
  // useCallback hook returns a callback that isn't recreated every time
  const inc = React.useCallback(
```

```
    function handleIncrementCount() {
      setCount((prevCount) => prevCount + 1);
    },
    // useCallback accepts a second arg of a dependencies array like useEffect
    // useCallback will only run if any dependency changes (here it's 'setCount')
    [setCount]
  );

  React.useEffect(() => {
    const timeout = setTimeout(() => {
      const currentTime = JSON.stringify(new Date(Date.now()));
      setTime(currentTime);
    }, 300);

    return () => {
      clearTimeout(timeout);
    };
  }, [time]);

  return (
    <div>
      <p>The current time is: {time}</p>
      <p>Count: {count}</p>
      <button onClick={inc}>+</button>
    </div>
  );
}
```

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 (we already did it once for these values, so no new to do it again)
    - useMemo returns a value from the computation, not a callback function (but can be a function)

```
// useMemo is useful when we need a lot of computing resources
// to perform an operation, but don't want to repeat it on each re-render

function App() {
  // state to select a word in 'words' array below
  const [wordIndex, setWordIndex] = useState(0);
  // state for counter
```

```
  const [count, setCount] = useState(0);

  // words we'll use to calculate letter count
  const words = ["i", "am", "learning", "react"];
  const word = words[wordIndex];

  function getLetterCount(word) {
    // we mimic expensive calculation with a very long (unnecessary) loop
    let i = 0;
    while (i < 1000000) i++;
    return word.length;
  }

  // Memoize expensive function to return previous value if input was the same
  // only perform calculation if new word without a cached value
  const letterCount = React.useMemo(() => getLetterCount(word), [word]);

  // if calculation was done without useMemo, like so:

  // const letterCount = getLetterCount(word);

  // there would be a delay in updating the counter
  // we would have to wait for the expensive function to finish

  function handleChangeIndex() {
    // flip from one word in the array to the next
    const next = wordIndex + 1 === words.length ? 0 : wordIndex + 1;
    setWordIndex(next);
  }

  return (
    <div>
      <p>
        {word} has {letterCount} letters
      </p>
      <button onClick={handleChangeIndex}>Next word</button>
      <p>Counter: {count}</p>
      <button onClick={() => setCount(count + 1)}>+</button>
    </div>
  );
}
```

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 top of component) and attach the returned value to the element's ref attribute to refer to it
- Once we create a reference, we use the current property to modify (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 ref an empty object
  const searchInput = useRef(null);

  function handleClearSearch() {
    // current references the text input once App mounts
    searchInput.current.value = "";
    // useRef can store basically any value in its .current property
    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>
  );
}
```

Advanced Hooks

Context and useContext

- In React, we want to avoid the following problem of creating multiple props to pass data down two or more levels from a parent component

```
// Context helps us avoid creating multiple duplicate props
// This pattern is also called props drilling:
function App() {
  // we want to pass user data down to Header
  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
const UserContext = React.createContext();
// we call this 'UserContext' because that's what data we're passing down

function App() {
  // we want to pass user data down to Header
  const [user] = React.useState({ name: "Fred" });

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

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

// we can remove the two 'user' props, we can just use consumer
// to consume the data where we need it
const Header = () => (
  {/* we use this pattern called render props to get access to the data*/}
  <UserContext.Consumer>
```

```
    {user => <header>Welcome, {user.name}!</header>}
  </UserContext.Consumer>
);
```

- The useContext hook can remove this unusual-looking render props pattern, however to be able to consume context in whatever function component we like

```
const Header = () => {
  // we pass in the entire context object to consume it
  const user = React.useContext(UserContext);
  // and we can remove the Consumer tags
  return <header>Welcome, {user.name}!</header>;
};
```

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. For example:

```
// let's say this reducer manages user state in our app:
function reducer(state, action) {
  // reducers often use a switch statement to update state
  // in one way or another based on the action's type property
  switch (action.type) {
    // if action.type has the string 'LOGIN' on it
    case "LOGIN":
      // we get data from the payload object on action
      return { username: action.payload.username, isAuth: true };
    case "SIGNOUT":
      return { username: "", isAuth: false };
    default:
      // if no case matches, return previous state
      return state;
  }
}
```

- Reducers are a powerful pattern for managing state that is used in the popular state management library Redux (common used with React)
- 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
- 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 initialState
  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" });
  }

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

Writing custom hooks

- Hooks were created to easily reuse behavior between components

    - Hooks are a more understandable pattern than previous ones for class components, such as higher-order components or render props
- What's great is that we can create our own hooks according to our own projects' needs, aside from the ones we've covered that React provides:

```
// here's a custom hook that is used to fetch data from an API
function useAPI(endpoint) {
  const [value, setValue] = React.useState([]);

  React.useEffect(() => {
    getData();
  }, []);

  async function getData() {
    const response = await fetch(endpoint);
    const data = await response.json();
    setValue(data);
  };

  return value;
};

// this is a working example! try it yourself (i.e. in codesandbox.io)
function App() {
  const todos = useAPI("<https://todos-dsequjaojf.now.sh/todos>");

  return (
    <ul>
      {todos.map(todo => <li key={todo.id}>{todo.text}</li>}
    </ul>
  );
}
```

Rules of hooks

- There are two core rules of using React hooks that we cannot violate for them to work properly:
    - Hooks can only be called at the top of components
        - Hooks cannot be in conditionals, loops or nested functions
    - Hooks can only be used within function components
        - Hooks cannot be in normal JavaScript functions or class components

```
function checkAuth() {
  // Rule 2 Violated! Hooks cannot be used in normal functions, only components
  React.useEffect(() => {
    getUser();
  }, []);
}

function App() {
  // this is the only validly executed hook in this component
  const [user, setUser] = React.useState(null);

  // Rule 1 violated! Hooks cannot be used within conditionals (or loops)
  if (!user) {
    React.useEffect(() => {
      setUser({ isAuth: false });
      // if you want to conditionally execute an effect, use the
      // dependencies array for useEffect
    }, []);
  }

  checkAuth();

  // Rule 1 violated! Hooks cannot be used in nested functions
  return <div onClick={() => React.useMemo(() => doStuff(), [])}>Our app</div>;
}
```