React Cheatsheet for 2020

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
<div>Hello React</div> // i.e. div/span, h1-h6, form/input, etc
```

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 {greeting}</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 vari-
ables
<button className=”submit-button”>Submit</button> // not
‘class’ (HTML)
```

The most basic React app

```
// 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

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

Data 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 ar-
gument
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)

```
function Header(props) {
  // we cannot mutate the props object, we can only read from
it
  props.username = "Doug";
  return <h1>Hello {props.username}</h1>;
}
```

Children props for passing through components

```
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

```
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 for displaying multiple components

```
function Header() {
  const isAuthenticated = checkAuth();
  return (
    <nav>
      {isAuthenticated ? (
        <>
          <AuthLinks />
          <Greeting />
        </>
      ) : <Login />}
    </nav>
  );
}
```

Lists and Keys

.map() to convert arrays into lists of elements

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

.map() used for components as well

```
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>;
}
```

React elements iterated over need a key prop

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

  // keys need to be primitive values (i.e. strings, numbers)
  return (
    <ul>
   {people.map(person => <Person key={person} name={person} />)}
    </ul>
  );
}

// If you don't have ids with your set of data or unique primi-
tive values,
// you can use the second parameter of .map() to get each ele-
ments 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

```
// in html, onclick is all
<button onclick=”handleToggleTheme()”>
  Submit
</button>

// in JSX, onClick is camelcase, like attributes / props
<button onClick={handleToggleTheme}>
  Submit
</button>
```

Most essential React events - onClick/onChange

```
function App() {
  function handleChange(event) {
    const inputText = event.target.value;
  }
  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>
  );
}
```

State and useState

useState gives us local state in components

```
import React from ‘react’;

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

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

useState has a ‘setter’ function to update state

```
function App() {
  // the setter function is always the second destructured val-
ue
  const [language, setLanguage] = React.useState(“python”);
  // the convention for the setter name is ‘setStateVariable’
  return (
    <div>
      <button onClick={() => setLanguage(“javascript”)}>
        Change language to JS
      </button>
      <p>I am now learning {language}</p>
    </div>
  );
}
```

useState can be used once or multiple times

```
function App() {
  const [language, setLanguage] = React.useState(“python”);
  const [yearsExperience, setYearsExperience] = React.useS-
tate(0);

  return (
    <div>
      <button onClick={() => setLanguage(“javascript”)}>
        Change language to JS
      </button>
      <input
        type=”number”
        value={yearsExperience}
        onChange={event => setYearsExperience(event.target.val-
ue)}
      />
      <p>I am now learning {language}</p>
      <p>I have {yearsExperience} years of experience</p>
    </div>
  );
}
```

useState can accept primitive or object values

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

  const handleChangeYearsExperience = (e) => {
   setDeveloper({ ...developer, yearsExperience: e.target.value });
  }

  return (
    <div>
      <button
        onClick={() => setDeveloper({
            language: “javascript”,
            yearsExperience: 0
          })
        }
      >
        Change language to JS
      </button>
      <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>
  );
}
```

Use function to ensure state updated reliably

```
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 param-
eters
    // 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

```
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 Re-
act
  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 : color-
Index + 1;
    setColorIndex(next);
  }

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

To run callback once, use empty dependencies 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

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

To conditionally run callback, add dependencies

```
function App() {
  ...
  // when colorIndex changes, the fn will run again
  useEffect(() => {
    document.body.style.backgroundColor = colors[colorIndex];
  }, [colorIndex]);
  ...
}
```

useEffect lets us unsubscribe by returning function

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

  React.useEffect(() => {
    // .addEventListener() sets up an active listener...
    window.addEventListener("mousemove", event => {
      const { pageX, pageY } = event;
      setMousePosition({ x: pageX, y: pageY });
    });

    // ...so when we navigate away from this page, it needs to
be
    // removed to stop listening. So we unsubscribe here:
    return () => {
      window.removeEventListener("mousemove", event => {
        const { pageX, pageY } = event;
        setMousePosition({ x: pageX, y: pageY });
      });
    };
  }, []);

  return (
    <div>
      <p>X: {mousePosition.x}, Y: {mousePosition.y}</p>
    </div>
  );
}
```

Fetching data with useEffect

```
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/co-
deartistryio”);
    const data = await response.json();
    setUser(data);
  }
}
```

Performance and useCallback

useCallback prevents callback from being remade

```
function Timer() {
  const [time, setTime] = React.useState();
  const [count, setCount] = React.useState(0);

  // fn 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 will only run if any dependency changes (here
it’s ‘setCount’)
    [setCount]
  );

  React.useEffect(() => {
    setTimeout(() => {
      setTime(JSON.stringify(new Date(Date.now())));
    }, 300);
  }, [time]);

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

Memoization and useMemo

useMemo caches result of expensive operations

```
function App() {
  const [wordIndex, setWordIndex] = useState(0);
  const [count, setCount] = useState(0);
  const words = [“i”, “am”, “learning”, “react”];
  const word = words[wordIndex];

  function getLetterCount(word) {
    let i = 0;
    while (i < 1000000) i++;
    return word.length;
  }

  // Memoize expensive function to return previous value if in-
put was the same
  const letterCount = React.useMemo(() => getLetterCount(word),
[word]);

  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 / useRef for creating reference to elements

```
function App() {
  const [query, setQuery] = React.useState(“react hooks”);
  // we can pass useRef a default value
  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>
  );
}
```

Context and useContext

Context for passing data multiple levels

```
const UserContext = React.createContext();

function App() {
  const [user] = React.useState({ name: “Fred” });

  return (
    {/* we wrap the parent component with 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>
  </>
);

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>
);
```

useContext hook to consume context more easily

```
const UserContext = React.createContext();

function App() {
  const [user] = React.useState({ name: “Fred” });

  return (
    {/* we wrap the parent component with 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>
  </>
);

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 - predictable fns for managing state

```
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;
  }
}
```

useReducer allows us to manage state across our app

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

function reducer(state, action) {
  switch (action.type) {
    case “LOGIN”:
      return { username: action.payload.username, isAuth: true };
    case “SIGNOUT”:
      return { username: “”, isAuth: false };
    default:
      return state;
  }
}

function App() {
  const [state, dispatch] = useReducer(reducer, initialState);

  function handleLogin() {
    dispatch({ type: “LOGIN”, payload: { username: “Ted” } });
  }

  const handleSignout = () => dispatch({ type: “SIGNOUT” });

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

Writing custom hooks

Hooks can be made according to what our app needs

```
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 code-
sandbox.io)
function App() {
  const todos = useAPI("https://todos-dsequjaojf.now.sh/to-
dos");

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

Rules of hooks

There are two core rules of working with hooks

```
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>;
}
```

Hope this cheatsheet was helpful!

Visit CodeArtistry