React for Beginners (2021)

React Basics

What is React, really?

- React is officially defined as a "JavaScript library for creating user interfaces," but what does that really mean?
- React is a library, made in JavaScript and which we code in JavaScript, to build great applications that run on the web.

What do I need to know for React?

- In other words, you do need to have a basic understanding of JavaScript to become a solid React programmer.
- The most basic JavaScript concepts you should be familiar with are variables, basic data types, conditionals, array methods, functions, and ES modules.
- How do I learn all of these JavaScript skills? Check out the comprehensive guide to learn all of the JavaScript you need for React.

If React was made in JavaScript, why don't we just use JavaScript?

- While React was written in JavaScript, which was built from the ground up for the express purpose of building web applications and gives us tools to do so.
- JavaScript is a 20+ year old language which was created for adding small bits of behavior to the browser through scripts and was not designed for creating complete applications.
- In other words, while JavaScript was used to create React, they were created for very different purposes.

Can I use JavaScript in React applications?

- Yes! Any valid JavaScript code can be included within your React applications.
- You can use any browser or window API, such as geolocation or the fetch API.

- Also, since React (when it is compiled) runs in the browser, you can perform common JavaScript actions like DOM querying and manipulation.

How to Create React Apps

Three different ways to create a React application

1. Putting React in an HTML file with external scripts
2. Using an in-browser React environment like CodeSandbox
3. Creating a React app on your computer using a tool like Create React App

What is the best way to create a React app?

- Which is the best approach for you? The best way to create your application depends on what you want to do with it.
- If you want to create a complete web application that you want to ultimately push to the web, it is best to create that React application on your computer using a tool like Create React App.
- If you are interested in creating React apps on your computer, check out the complete guide to using Create React App.
- The easiest and most beginner-friendly way to create and build React apps for learning and prototyping is to use a tool like CodeSandbox. You can create a new React app in seconds by going to react.new!

JSX Elements

JSX is a powerful tool for structuring applications

- **JSX** is meant to make create user interfaces with JavaScript applications easier.
- JSX borrows its syntax from the most widely used programming language: HTML
- As a result, JSX is a powerful tool to structure our applications.

- The code example below is the most basic example of a React element which displays the text "Hello World"

```
<div>Hello React!</div>
```

> To be displayed in the browser, React elements need to be rendered (using ReactDOM.render())

How JSX differs from HTML

- We can write valid HTML element in JSX, but what differs slightly is the way some attributes are written.
- Attributes that consist of multiple words are written in the camel-case syntax (i.e. `className`) and have different names than standard HTML (`class`).

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

- The reason JSX has this different way of writing attributes is because it is actually made using JavaScript functions (more on this later).

JSX must have a trailing slash if it is made of one tag

- Unlike standard HTML, elements like `input`, `img`, or `br` must close with a trailing forward slash for it to be valid JSX.

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

JSX elements with two tags must have a closing tag

- Elements that should have two tags, such as `div`, `main` or `button`, must have their closing, second tag in JSX, otherwise it will result in a syntax error.

```
<button>Click me</button>// <button> or </button> is a syntax error
```

How JSX elements are styled

- Inline styles are written differently as well as compared to plain HTML.
- Inline styles must not be included as a string, but within an object.
- Once again, the style properties that we use must be written in the camel-case style.

```
<h1 style={{ color: "blue", fontSize: 22, padding: "0.5em 1em" }}>
  Hello React!
</h1>;
```

> Style properties that accept pixel values (like width, height, padding, margin, etc), can use integers instead of strings. For example, fontSize: 22 instead of fontSize: "22px"

JSX can be conditionally displayed

- New React developers may be wondering how it is beneficial that React can use JavaScript code.
- One simple example if that to conditionally hide or display JSX content, we can use any valid JavaScript conditional, like an if statement or switch statement.

```
const isAuthUser = true;

if (isAuthUser) {
  return <div>Hello user!</div>
} else {
  return <button>Login</button>
}
```

- Where are we returning this code? Within a React component, which we will cover in a later section.

JSX cannot be understood by the browser

- As mentioned above, JSX is not HTML, but composed of JavaScript functions.

- In fact, writing `<div>Hello React</div>` in JSX is just a more convenient and understandable way of writing code like the following:

```
React.createElement("div", null, "Hello React!")
```

- Both pieces of code will have the same output of "Hello React".
- To write JSX and have the browser understand this different syntax, we must use a **transpiler** to convert JSX to these function calls.
- The most common transpiler is called **Babel.**

Components

What are React components?

- Instead of just rendering one or another set of JSX elements, we can include them within React **components**.
- Components are created using what looks like a normal JavaScript function, but is different in that it returns JSX elements.

```
function Greeting() {
  return <div>Hello React!</div>;
}
```

Why use React components?

- React components allow us to create more complex logic and structures within our React application than we would with JSX elements alone.
- Think of React components as our custom React elements that have their own functionality.
- As we know, functions allow us to create our own functionality and reuse it where we like across our application.
- Components are reusable wherever we like across our app and as many times as we like.

Components are not normal JavaScript functions

- How would we render or display the returned JSX from the component above?

```
import React from 'react';
import ReactDOM from 'react-dom';

function Greeting() {
  return <div>Hello React!</div>;
}

ReactDOM.render(<Greeting />, document.getElementById("root));
```

- We use the `React` import to parse the JSX and `ReactDOM` to render our component to a **root element** with the id of "root."

What can components return?

- Components can return valid JSX elements, as well as strings, numbers, booleans, the value `null`, as well as arrays and fragments.
- Why would we want to return `null`? It is common to return `null` if we want a component to display nothing.

```
function Greeting() {
  if (isAuthUser) {
    return "Hello again!";
  } else {
    return null;
  }
}
```

- Another rule is that JSX elements must be wrapped in one parent element. Multiple sibling elements cannot be returned.
- If you need to return multiple elements, but don't need to add another element to the DOM (usually for a conditional), you can use a special React component called a fragment.
- Fragments can be written as `<></>` or when you import React into your file, with `<React.Fragment></React.Fragment>`.

```
function Greeting() {
  const isAuthUser = true;

  if (isAuthUser) {
    return (
      <>
        <h1>Hello again!</h1>
        <button>Logout</button>
      </>
    );
  } else {
    return null;
  }
}
```

> Note that when attempting to return a number of JSX elements that are spread over multiple lines, we can return it all using a set of parentheses () as you see in the example above.

Components can return other components

- The most important thing components can return is other components.
- Below is a basic example of a React application contained with in a component called `App` that returns multiple components:

```
import React from 'react';
import ReactDOM from 'react-dom';

import Layout from './components/Layout';
import Navbar from './components/Navbar';
import Aside from './components/Aside';
import Main from './components/Main';
import Footer from './components/Footer';

function App() {
  return (
    <Layout>
      <Navbar />
      <Main />
      <Aside />
      <Footer />
    </Layout>
  );
}
```

```
ReactDOM.render(<App />, document.getElementById('root'));
```

- What is powerful about this is that we are using the customization of components to describe what they are (i.e. Layout) and their function in our application. This tells us how they should be used just by looking at their name.
- Additionally, we are using the power of JSX to compose these components. In other words, to use the HTML-like syntax of JSX to structure them in an immediately understandable way (i.e. the Navbar is at the top of the app, the Footer at the bottom, etc).

JavaScript can be used in JSX using curly braces

- Just as we can use JavaScript variables within our components, we can use them directly within our JSX as well.
- There are a few core rules to using dynamic values within JSX, however.
- JSX can accept any primitive values (strings, booleans, numbers), but it will not accept plain objects.
- JSX can also include expressions that resolve to these values.
- For example, conditionals can be included within JSX using the ternary operator, since it resolves to a value.

```
function Greeting() {
  const isAuthUser = true;

  return <div>{isAuthUser ? "Hello!" : null}</div>;
}
```

Props

Components can be passed values using props

- 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

```
ReactDOM.render(
  <Greeting username="John!" />,
  document.getElementById("root")
);

function Greeting(props) {
  return <h1>Hello {props.username}</h1>;
}
```

Props cannot be directly changed

- 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

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

  return <h1>Hello {props.username}</h1>;
}
```

> Components are consider pure functions. That is, for every input, we should be able to expect the same output. This means we cannot mutate the props object, only read from it.

Special props: the children prop

- The **children** prop is useful if we want to pass elements / components as props to other components
- The children prop is especially useful for when you want the same component (such as a Layout component) to wrap all other components.

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

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

function AboutPage() {
  return (
    <Layout>
      <About />
      <Footer />
    </Layout>
  );
}
```

- The benefit of this pattern is that all styles applied to the Layout component will be shared with its child components.

Lists and Keys

Iterate over arrays in JSX using map

- How do we displays lists in JSX using array data?
- Use the **.map()** function to convert lists of data (arrays) into lists of elements.

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

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

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

  return (
    <ul>
```

```
      {people.map((person) => (
        <Person name={person} />
      ))}
    </ul>
  );
}

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

The importance of keys in lists

- 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

```
function App() {
  const people = [
    { id: "Ksy7py", name: "John" },
    { id: "6eAdl9", name: "Bob" },
    { id: "6eAdl9", name: "Fred" },
  ];

  return (
    <ul>
      {people.map((person) => (
        <Person key={person.id} name={person.name} />
      ))}
    </ul>
  );
}
```

State and Managing Data

What is state?

- **State** is a concept that refers to how data in our application changes over time.
- The significance of state in React is that it is a way to talk about our data separately from the user interface (what the user sees).
- We talk about state management, because we need an effective way to keep track of and update data across our components as our user interacts with it.
- To change our application from static HTML elements to a dynamic one that the user can interact with, we need state.

Common examples of using state

- We need to manage state often when our user wants to interact with our application.
- When a user types into a form, we keep track of the form state in that component.
- When we fetch data from an API to display to the user (i.e. posts in a blog), we need to save that data in state.
- When we want to change data that a component is receiving from props, we use state to change it instead of mutating the props object.

Introduction to React hooks with useState

- The way to "create" state is React within a particular component is with the `useState` hook.
- What is a hook? It is very much like a JavaScript function, but can only be used in a React function component at the top of the component.
- We use hooks to "hook into" certain features and useState gives us the ability to create and manage state.
- `useState` is an example of a core React hook that comes directly from the React library: `React.useState`.

```
import React from 'react';

function Greeting() {
  const state = React.useState("Hello React");

  return <div>{state[0]}</div>// displays "Hello React"}
```

- How does `useState` work? Like a normal function, we can pass it a starting value (i.e. "Hello React").
- What is returned from useState is an array. To get access to the state variable and its value, we can use the first value in that array: `state[0]`.
- There is a way to improve how we write this, however. We can use array destructuring to get direct access to this state variable and call it what we like, i.e. `title`.

```
import React from 'react';

function Greeting() {
  const [title] = React.useState("Hello React");

  return <div>{title}</div>// displays "Hello React"}
```

- What if we want to allow our user to update the greeting they see?
- If we include a form, a user can type in a new value. However, we need a way to update the initial value of our title.

```
import React from "react";

function Greeting() {
  const [title] = React.useState("Hello React");

  return (
    <div>
      <h1>{title}</h1>
      <input placeholder="Update title" />
    </div>
  );
}
```

- We can do so with the help of the second element in the array that useState returns. It is a setter function, to which we can pass whatever value we want the new state to be.

- In our case, we want to get the value that is typed into the input when a user is in the process of typing. We can get it with the help of React events.

What are events in React?

- Events are ways to get data about a certain action that a user has performed in our app.
- The most common props used to handle events are `onClick` (for click events), `onChange` (when a user types into an input), and `onSubmit` (when a form is submitted.
- Event data is given to us by connecting a function to each of these props listed (there are many more to choose from than these three).
- To get data about the event when our input is changed, we can add `onChange` on input and connect it to a function that will handle the event. This function will be called `handleInputChange`:

```
import React from "react";

function Greeting() {
  const [title] = React.useState("Hello React");

  function handleInputChange(event) {
    console.log("input changed!", event);
  }

  return (
    <div>
      <h1>{title}</h1>
      <input placeholder="Update title" onChange={handleInputChange} />
    </div>
  );
}
```

> Note that in the code above, a new event will be logged to the browser's console whenever the user types into the input

- Event data is provided to us as an object with many properties which are dependent upon the type of event.

Updating state with useState

- To update state with useState, we can use the second element that useState returns to us in its array.
- This element is a function that will allow us to update the value of the state variable (the first element)
- Whatever we pass to this setter function when we call it will be put in state.

```
import React from "react";

function Greeting() {
  const [title, setTitle] = React.useState("Hello React");

  function handleInputChange(event) {
    setTitle(event.target.value);
  }

  return (
    <div>
      <h1>{title}</h1>
      <input placeholder="Update title" onChange={handleInputChange} />
    </div>
  );
}
```

- Using the code above, whatever the user types into the input (the text comes from `event.target.value`) will be put in state using `setTitle` and displayed within the `h1` element.
- What is special about state and why it must be managed with a dedicated hook like useState is because a state update (such as when we call `setTitle` causes a re-render.

> A re-render is when a certain component renders or is displayed again based off the new data. If our components weren't re-rendered when data changed, we would never see the app's appearance change at all!

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

React Fundamentals

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 HTML.
As a result, JSX is a powerful tool to structure our applications.
JSX uses all valid HTML tags (i.e. div/span, h1-h6, form/input, img, etc).
*/

<div>Hello React!</div>

/*
Note: this JSX would not be visible because it is needs to be rendered by our application using ReactDOM.render()
*/
```

- JSX is the most common way to structure React applications, but JSX 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 syntax
```

- JSX is not understood by the browser. JSX needs to be compiled to plain JavaScript, which the browser can understand.
- The most commonly used compiler for JSX is called Babel

```
/*
When our project is built to run in the browser, our JSX will be converted by Babel into simple React.createElement() function calls
From this...
*/
const greeting = <div>Hello React!</div>;

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

const greeting = /*#__PURE__*/React.createElement("div", null, "Hello React!");
```

- JSX differs from HTML in several important ways

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

1) Some JSX attributes are named differently than HTML attributes. Why? Because some attribute words are reserved words in JavaScrip

Also, because JSX is JavaScript, attributes that consist of multiple words are written in camelcase:
*/

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

/*
2) JSX elements that consist of only a single tag (i.e. input, img, br elements) must be closed with a trailing forward slash to be
*/
```

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

/*
3) JSX elements that consists of an opening and closing tag (i.e. div, span, button element), must have both or be closed with a tra
*/

<button>Click me</button> // <button> or </button> is a syntax error
<button /> // empty, but also valid
```

- Inline styles can be added to JSX elements using the style attribute
- 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, margin, etc), can use integers instead of strings.
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 can be assigned to plain JavaScript 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 insert (or embed) simple JavaScript expressions using the curly braces syntax

```
const year = 2021;

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

/* We can also insert expressions that resolve to a primitive value: */
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, maps) in curly braces will result in an error */
```

- 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 expressions
*/

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 */
```

- Comments in JSX are written as multiline JavaScript comments, written between curly braces

```
const greeting = (
  <div>
    {/* This is a single line comment */}
    <h1>Hello!</h1>
    <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

```
// Packages can be installed locally or brought in through a CDN link (added to head of HTML document)
import React from "react";
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 also valid
const Header = () => <h1>Hello React</h1>;

/*
Class component
Class components have more boilerplate (note the 'extends' keyword and 'render' method)
*/
class Header extends React.Component {
  render() {
    return <h1>Hello React</h1>;
  }
}
```

- Components, despite being functions, are not called like ordinary JavaScript functions
- Components 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 our app.
Components remove the need to rewrite the same JSX multiple times.
*/

// IndexPage component, visible on '/' route of our app
function IndexPage() {
  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 parent component?
For example, to display the user's name in our app header.
*/

const username = "John";

/*
To do so, we add custom 'attributes' to our component called props
We can add many of them as we like and we give them names that suit the data we pass in.
To pass the user's name to the header, we use a prop we appropriately called 'username'
```

```
*/
ReactDOM.render(
  <Header username={username} />,
  document.getElementById("root")
);
// We called this prop 'username', but can use any valid identifier we would give, for example, a JavaScript variable

// props is the object that every component receives as an argument
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 output.
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 component?
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 '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>
  );
}
```

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

```
function Header() {
  const isAuthenticated = checkAuth();
  /* if user is authenticated, show the authenticated app, otherwise, the unauthenticated app */
  if (isAuthenticated) {
    return <AuthenticatedApp />;
  } else {
    /* alternatively, we can drop the else section and provide a simple return, and the conditional will operate in the same way */
    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 Login  */}
      {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
- Fragments are ideal for conditional logic that have multiple adjacent components or elements

```
/*
We can improve the logic in the previous example.
If isAuthenticated is true, how do we display both the AuthenticatedApp and Footer components?
*/
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

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

- .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

```
function App() {
  const people = [
    { id: "Ksy7py", name: "John" },
    { id: "6eAdl9", name: "Bob" },
    { id: "6eAdl9", name: "Fred" },
  ];

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

/* If you don't have some identifier with your set of data that is a unique
and primitive value, 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>
  );
}
```

Event Listeners and Handling Events

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

```
/*
The convention for most event handler functions is to prefix them with the word 'handle' and then the action they perform (i.e. hand
*/
function handleToggleTheme() {
  // code to toggle app theme
}

/* In HTML, onclick is all lowercase, plus the event handler includes a set of parentheses after being referenced */
<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 onChange we get access to data about the event (an object) */
    const inputText = event.target.value; // text typed into the input
    const inputName = event.target.name; // 'email' from name attribute
  }

  function handleClick(event) {
    /* onClick doesn't usually need event data, but it receives event data as well that we can use */
    console.log("clicked!");
    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 well as when a button with type="submit" is clicked.
   We call event.preventDefault() to prevent the default form behavior from taking place, which is to send an HTTP request and reloa
  */
    event.preventDefault();
    const formElements = event.target.elements; // access all element within form
    const inputValue = event.target.elements.emailAddress.value; // access the value of the input element with the id "emailAddress"
  }

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

Essential React Hooks

State and useState

- useState give 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 providing it as the first argument when we call `useState`

```
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 this case, 'language').
*/

  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

```
function App() {
  /*
  The setter function is always the second destructured value.
  The naming convention for the setter function is to be prefixed with 'set'.
  */
  const [language, setLanguage] = React.useState("javascript");

  return (
    <div>
      <button onClick={() => setLanguage("python")}>Learn Python</button>
      {/*
      Why use an inline arrow function here instead of immediately calling it like so: onClick={setterFn()}?
      If so, setLanguage would be called immediately and not when the button was clicked by the user.
      */}
      <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
- useState 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>
      <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>
  );
}
```

- 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 appropriate data type, according to the data we're managing */
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 to spread it all of its properties */
    setDeveloper({ ...developer, yearsExperience: years });
  }

  return (
    <div>
      {/* No need to get previous 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 you are managing multiple primitive values, using useState multiple times is often better than using useState 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 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 */
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 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 handleChangeColor() {
    /* This code may look complex, but all it does is go to the next color in the 'colors' array, and if it is on the last color, go
    const nextIndex = colorIndex + 1 === colors.length ? 0 : colorIndex + 1;
    setColorIndex(nextIndex);
  }

  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 times we click it...
  The background color is only set once, when the component first mounts.
  */
  useEffect(() => {
    document.body.style.backgroundColor = colors[colorIndex];
  }, []);

  /*
  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"];

  /*
  Let's add colorIndex to our dependencies array
  When colorIndex changes, useEffect will execute the effect function 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>
  );
}
```

- 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)

```
const endpoint = "https://api.github.com/users/reedbarger";

// 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 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 reference an empty object
  */
  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

Preventing Re-renders and React.memo

- React.memo is a function that allows us to optimize the way our components are rendered
- In particular, React.memo performs a process called **memoization** that helps us prevent our components from re-rendering when they do not need to be (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 programming skills. We can create new skills using an input, they are adde
*/

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 type into the input, because the parent component of SkillList (A

/* However, once we wrap the SkillList component in React.memo (which is a higher-order function, meaning it accepts a function as a
const SkillList = React.memo(({ skills }) => {
  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 add one small feature. Let's make it possible to delete a skill when

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));
  }

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

  /* Next, we pass handleRemoveSkill down as a prop, or since this is a function, as a callback function to be used within SkillList
  return (
    <>
      <input onChange={handleChangeInput} />
      <button onClick={handleAddSkill}>Add Skill</button>
      <SkillList skills={skills} handleRemoveSkill={handleRemoveSkill} />
    </>
  );
}

/* When we try typing in the input again, we see rerendering in the console every time we type. Our memoization from React.memo is b

  What is happening is the handleRemoveSkill callback function is being recreated everytime App is rerendered, causing all children

  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={() => handleRemoveSkill(skill)}>
          {skill}
        </li>
      ))}
    </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 through our available skills through an additional search input. We c
 */

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));
  }

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

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

// /* Let's imagine we have a list of thousands of skills that we want to search through. How do we performantly find and show the s
function SearchSkills() {
  const [searchTerm, setSearchTerm] = React.useState("");

  /* We use React.useMemo to memoize (remember) the returned value from our search operation and only run when it the searchTerm cha
  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

- 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

```
/*
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 component, but it first needs to go through a Main component which
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();

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

  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 remove the two 'user' props. Instead, we can just use the Consumer property to consume the data where we need it
*/
const Header = () => (
  /* We use a 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 to consume context in any function component that is a child of the Provider

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

  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.

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

function userReducer(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, we get data from the payload object on action */
    case "LOGIN":
      return {
        username: action.payload.username,
        email: action.payload.email,
        isAuth: true,
      };
    case "SIGNOUT":
      return {
        username: "",
        isAuth: false,
      };
    default:
      /* If no case matches the action received, return the previous state */
      return state;
  }
}
```

- Reducers are a powerful pattern for managing state that is used in the popular state management library Redux (commonly 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, similar to how components were created to reuse structure across our application
- Hooks enable us to add custom functionality to our apps that suit 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 in order to calculate the window size (width and height) of any com

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.innerHeight });
  }

  React.useEffect(() => {
    window.addEventListener("resize", changeWindowSize);

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

  return windowSize;
}

/* To use the hook, we just need to import it where we need, call it, and use the width wherever we want to hide or show certain ele

// 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 used within function components (not plain JavaScript functions or class components)
- Hooks can only be called at the top of components (they cannot be in conditionals, loops, or nested functions)

The React Router Cheatsheet

Basic Router Setup

```
import { BrowserRouter as Router } from 'react-router-dom';

export default function App() {
  return (
    <Router>
      {/* routes go here, as children */}
    </Router>
  );
}
```

Route Component

Basic declaration

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

export default function App() {
  return (
    <Router>
      <Route path="/about" component={About} />
    </Router>
  );
}

function About() {
  return <>about</>
}
```

Render prop

```
import { BrowserRouter as Router, Route } from "react-router-dom";

export default function App() {
  return (
    <Router>
      <Route path="/" render={() => <Home />} />
      <Route path="/about" component={About} />
    </Router>
  );
}

function Home() {
  return <>home</>;
}

function About() {
  return <>about</>;
}
```

Alternate declaration

```
import { BrowserRouter as Router, Route } from "react-router-dom";

export default function App() {
  return (
    <Router>
      <Route path="/about">
        <About />
      </Route>
    </Router>
  );
}
```

Visible on all routes

```
import { BrowserRouter as Router, Route } from "react-router-dom";

export default function App() {
  return (
    <Router>
      <Navbar />
      <Route path="/" component={Home} />
      <Route path="/about" component={About} />
    </Router>
  );
}

function Navbar() {
  // visible on every page
  return <>navbar</>
}

function Home() {
  return <>home</>;
}

function About() {
  return <>about</>;
}
```

Switch Component

Exact prop

```
import { BrowserRouter as Router, Switch, Route } from "react-router-dom";

export default function App() {
  return (
    <Router>
      <Navbar />
      <Switch>
        <Route exact path="/" component={Home} />
        <Route path="/about" component={About} />
      </Switch>
    </Router>
  );
}
```

Basic declaration

```
import { BrowserRouter as Router, Switch, Route } from "react-router-dom";

export default function App() {
  return (
    <Router>
      <Navbar />
      <Switch>
        <Route exact path="/" component={Home} />
        <Route path="/about" component={About} />
      </Switch>
    </Router>
  );
}
```

404 Route

```
import { BrowserRouter as Router, Switch, Route } from "react-router-dom";

export default function App() {
  return (
    <Router>
      <Navbar />
      <Switch>
        <Route path="/" component={Home} />
        <Route path="/about" component={About} />
        <Route path="*" component={NotFound} />
      </Switch>
    </Router>
  );
}

function NotFound() {
  return <>You have landed on a page that doesn't exist</>;
}
```

Link Component

```
import { BrowserRouter as Router, Switch, Route, Link } from "react-router-dom";

export default function App() {
  return (
    <Router>
      <Navbar />
      <Switch>
        <Route path="/" component={Home} />
        <Route path="/about" component={About} />
      </Switch>
    </Router>
  );
}

function Navbar() {
  return (
    <nav>
      <Link to="/">Home</Link>
      <Link to="/about">About</Link>
    </nav>
  )
}
```

NavLink Component

```
import {
  BrowserRouter as Router,
  Switch,
  Route,
  NavLink
} from "react-router-dom";

export default function App() {
  return (
    <Router>
      <Navbar />
      <Switch>
        <Route path="/" component={Home} />
        <Route path="/about" component={About} />
      </Switch>
    </Router>
  );
}

function Navbar() {
  return (
    <nav>
      <NavLink
        activeStyle={{
          fontWeight: "bold",
          color: "red"
        }}
        to="/"
      >
        Home
      </NavLink>
      <NavLink activeClassName="active" to="/about">
        About
      </NavLink>
    </nav>
  );
}
```

Redirect Component

Private Route component

```
import {
  BrowserRouter as Router,
  Switch,
  Route,
  Redirect
} from "react-router-dom";
```

```
export default function App() {
  return (
    <Router>
      <Switch>
        <Route exact path="/" component={Home} />
        <PrivateRoute path="/hidden" component={Hidden} />
      </Switch>
    </Router>
  );
}

function PrivateRoute({ component: Component, ...rest }) {
  // useAuth is some custom hook to get the current user's auth state
  const isAuth = useAuth();

  return (
    <Route
      {...rest}
      render={(props) =>
        isAuth ? <Component {...props} /> : <Redirect to="/" />
      }
    />
  );
}

function Home() {
  return <>home</>;
}

function Hidden() {
  return <>hidden</>;
}
```

useHistory Hook

```
import { useHistory } from "react-router-dom";


function About() {
  const history = useHistory();

  console.log(history.location.pathname); // '/about'

  return (
    <>
     <h1>The about page is on: {history.location.pathname}</h1>
     <button onClick={() => history.push('/')}>Go to home page</button>
    </>
  );
}
```

useLocation Hook

```
import { useLocation } from "react-router-dom";


function About() {
  const location = useLocation();

  console.log(location.pathname); // '/about'

  return (
    <>
     <h1>The about page is on: {location.pathname}</h1>
    </>
  );
}
```

Dynamic Routes

```
import React from "react";
import { BrowserRouter as Router, Switch, Route } from "react-router-dom";

export default function App() {
  return (
    <Router>
      <Switch>
        <Route exact path="/" component={Home} />
        <Route path="/blog/:postSlug" component={BlogPost} />
      </Switch>
    </Router>
  );
}

function Home() {
  return <>home</>;
}

function BlogPost() {
  return <>blog post</>;
}
```

useParams Hook

```
import React from "react";
import { BrowserRouter as Router, Switch, Route, useParams } from "react-router-dom";

export default function App() {
  return (
    <Router>
      <Switch>
        <Route exact path="/" component={Home} />
        <Route path="/blog/:postSlug" component={BlogPost} />
      </Switch>
    </Router>
  );
}

function Home() {
  return <>home</>;
}

function BlogPost() {
  const [post, setPost] = React.useState(null);
  const { postSlug } = useParams();

  React.useEffect(() => {
    fetch(`https://jsonplaceholder.typicode.com/posts/${postSlug}`)
      .then((res) => res.json())
      .then((data) => setPost(data));
  }, [postSlug]);

  if (!post) return null;

  return (
    <>
      <h1>{post.title}</h1>
      <p>{post.description}</p>
    </>
  );
}
```

useRouteMatch Hook

```
import { useRouteMatch } from "react-router-dom";

function BlogPost() {
  const isBlogPostRoute = useRouteMatch("/blog/:postSlug");
  // display, hide content, or do something else
}
```

TOP 50 REACT INTERVIEW QUESTIONS AND ANSWERS

Following are frequently asked React interview questions for fresher's as well as experienced React developers.

1) What is Reactjs?

React is a JavaScript library that makes building user interfaces easy. It was developed by Facebook.

2) Does React use HTML?

No, It uses JSX, which is similar to HTML.

3) When was React first released?

React was first released on March 2013.

4) Give me two most significant drawbacks of React

- Integrating React with the MVC framework like Rails requires complex configuration.
- React require the users to have knowledge about the integration of user interface into MVC framework.

5) State the difference between Real DOM and Virtual DOM

Real DOM	Virtual DOM
It is updated slowly.	It updates faster.
It allows a direct update from HTML.	It cannot be used to update HTML directly.
It wastes too much memory.	Memory consumption is less

6) What is Flux Concept In React?

Facebook widely uses flux architecture concept for developing client-side web applications. It is *not* a framework or a library. It is simply a new kind of architecture that complements React and the concept of Unidirectional Data Flow.

7) Define the term Redux in React

Redux is a library used for front end development. It is a state container for JavaScript applications which should be used for the applications state management. You can test and run an application developed with Redux in different environments.

8) What is the 'Store' feature in Redux?

Redux has a feature called 'Store' which allows you to save the application's entire State at one place. Therefore all it's component's State are stored in the Store so that you will get regular updates directly from the Store. The single state tree helps you to keep track of changes over time and debug or inspect the application.

9) What is an action in Redux?

It is a function which returns an action object. The action-type and the action data are always stored in the action object. Actions can send data between the Store and the software application. All information retrieved by the Store is produced by the actions.

10) Name the important features of React

Here, are important features of React.

- Allows you to use 3rd party libraries
- Time-Saving
- Faster Development
- Simplicity and Composable
- Fully supported by
- Facebook.
- Code Stability with One-directional data binding

React Components

11) Explain the term stateless components

Stateless components are pure functions that render DOM-based solely on the properties provided to them.

12) Explain React Router

React Router is a routing library which allows you to add new screen flows to your application, and it also keeps URL in sync with what’s being shown on the page.

13) What is dispatcher?

A dispatcher is a central hub of app where you will receive actions and broadcast payload to registered callbacks.

14) What is meant by callback function? What is its purpose?

A callback function should be called when setState has finished, and the component is retendered. As the setState is asynchronous, which is why it takes in a second callback function.

15) Explain the term high order component

A higher-order component also shortly known as HOC is an advanced technique for reusing component logic. It is not a part of the React API, but they are a pattern which emerges from React’s compositional nature.

16) Explain the Presentational segment

A presentational part is a segment which allows you to renders HTML. The segment’s capacity is presentational in markup.

17) What are Props in react js?

Props mean properties, which is a way of passing data from parent to child. We can say that props are just a communication channel between components. It is always moving from parent to child component.

18) Explain yield catchphrase in JavaScript

The yield catchphrase is utilized to delay and resume a generator work, which is known as yield catchphrase.

19) Name two types of React component

Two types of react Components are:

- Function component
- Class component

20) Explain synthetic event in React js

Synthetic event is a kind of object which acts as a cross-browser wrapper around the browser’s native event. It also helps us to combine the behaviors of various browser into signal API.

21) What is React State?

It is an object which decides how a specific component renders and how it behaves. The state stores the information which can be changed over the lifetime of a React component.

22) How can you update state in react js?

A state can be updated on the component directly or indirectly.

23) Explain the use of the arrow function in React

The arrow function helps you to predict the behavior of bugs when passed as a callback. Therefore, it prevents bug caused by this all together.

24) State the main difference between Pros and State

The main difference the two is that the State is mutable and Pros are immutable.

25) Explain pure components in React js

Pure components are the fastest components which can replace any component with only a render(). It helps you to enhance the simplicity of the code and performance of the application.

26) What kind of information controls a segment in React?

There are mainly two sorts of information that control a segment: State and Props

- State: State information that will change, we need to utilize State.
- Props: Props are set by the parent and which are settled all through the lifetime of a part.

27) What is 'create-react-app'?

'create-react-app' is a command-line tool which allows you to create one basic react application.

28) Explain the use of 'key' in react list

Keys allow you to provide each list element with a stable identity. The keys should be unique.

29) What are children prop?

Children props are used to pass component to other components as properties.

30) Explain error boundaries?

Error boundaries help you to catch Javascript error anywhere in the child components. They are most used to log the error and show a fallback UI.

31) What is the use of empty tags ?

Empty tags are used in React for declaring fragments.

32) Explain strict mode

StrictMode allows you to run checks and warnings for react components. It runs only on development build. It helps you to highlight the issues without rendering any visible UI.

33) What are reacted portals?

Portal allows you to render children into a DOM node. **CreatePortalmethod** is used for it.

34) What is Context?

React context helps you to pass data using the tree of react components. It helps you to share data globally between various react components.

35) What is the use of Webpack?

Webpack in basically is a module builder. It is mainly runs during the development process.

36) What is Babel in React js?

Babel, is a JavaScript compiler that converts latest JavaScript like ES6, ES7 into plain old ES5 JavaScript that most browsers understand.

37) How can a browser read JSX file?

If you want the browser to read JSX, then that JSX file should be replaced using a JSX transformer like Babel and then send back to the browser.

38) What are the major issues of using MVC architecture in React?

Here are the major challenges you will face while handling MVC architecture:

- DOM handling is quite expensive
- Most of the time applications were slow and inefficient
- Because of circular functions, a complex model has been created around models and ideas

39) What can be done when there is more than one line of expression?

At that time a multi-line JSX expression is the only option left for you.

40) What is the reduction?

The reduction is an application method of handling State.

41) Explain the term synthetic events

It is actually a cross-browser wrapper around the browser's native event. These events have interface stopPropagation() and preventDefault().

42) When should you use the top-class elements for the function element?

If your element does a stage or lifetime cycle, we should use top-class elements.

43) How can you share an element in the parsing?

Using the State, we can share the data.

44) Explain the term reconciliation

When a component's state or props change then rest will compare the rendered element with previously rendered DOM and will update the actual DOM if it is needed. This process is known as reconciliation.

45) How can you re-render a component without using setState() function?

You can use forceUpdate() function for re-rending any component.

46) Can you update props in react?

You can’t update props in react js because props are read-only. Moreover, you can not modify props received from parent to child.

47) Explain the term 'Restructuring.'

Restructuring is extraction process of array objects. Once the process is completed, you can separate each object in a separate variable.

48) Can you update the values of props?

It is not possible to update the value of props as it is immutable.

49) Explain the meaning of Mounting and Demounting

- The process of attaching the element to the DCOM is called mounting.
- The process of detaching the element from the DCOM is called the demounting process.

50) What is the use of ‘props-types’ liberary?

'Prop-types' library allows you to perform runtime type checking for props and similar object in a recent application.

****