| React | React |
| --- | --- |
| Chapter and Section | Section Title |
|  | **jsx-in-depth ⇒>** <https://react.dev/blog/2023/03/16/introducing-react-dev>  <https://zhenyong.github.io/react/docs/jsx-in-depth.html>  **Create React app command :** [**https://create-react-app.dev/docs/getting-started**](https://create-react-app.dev/docs/getting-started)  **Lifting-state-up tutorial :** [**https://react.dev/learn/tutorial-tic-tac-toe#lifting-state-up**](https://react.dev/learn/tutorial-tic-tac-toe#lifting-state-up)  **Thinking in React:** [**https://react.dev/learn/thinking-in-react**](https://react.dev/learn/thinking-in-react)  **React-Tutorial :** [**https://react.dev/learn**](https://react.dev/learn)  **HTML to JSX Converter :** [**https://transform.tools/html-to-jsx**](https://transform.tools/html-to-jsx)  **Built-in browser components :** [**https://react.dev/reference/react-dom/components/common**](https://react.dev/reference/react-dom/components/common)  **Web3js :** [**https://web3js.readthedocs.io/en/v1.8.2/getting-started.html**](https://web3js.readthedocs.io/en/v1.8.2/getting-started.html)  **Material UI library :** [**https://mui.com/material-ui/getting-started/overview/**](https://mui.com/material-ui/getting-started/overview/)  [**https://m2.material.io/**](https://m2.material.io/)  **Immer library :** [**https://github.com/immerjs/use-immer**](https://github.com/immerjs/use-immer)  [**https://immerjs.github.io/immer/**](https://immerjs.github.io/immer/) WebKit CSS extensions : <https://developer.mozilla.org/en-US/docs/Web/CSS/WebKit_Extensions> **Appwrite :** [**https://react-appwrite.org/**](https://react-appwrite.org/)  **React hooks Rules :** [**https://legacy.reactjs.org/docs/hooks-rules.html**](https://legacy.reactjs.org/docs/hooks-rules.html) Built-in React Hooks : <https://react.dev/reference/react> **CSS Grid Model :** [**https://developer.mozilla.org/en-US/docs/Web/CSS/grid**](https://developer.mozilla.org/en-US/docs/Web/CSS/grid)  **CSS flexbox Model :** [**https://developer.mozilla.org/en-US/docs/Web/CSS/CSS\_Flexible\_Box\_Layout**](https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_Flexible_Box_Layout)  **CSS flexbox Layout :** [**https://developer.mozilla.org/en-US/docs/Web/CSS/CSS\_Flexible\_Box\_Layout**](https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_Flexible_Box_Layout)  **Media Queries :** [**https://developer.mozilla.org/en-US/docs/Web/CSS/Media\_Queries/Using\_media\_queries**](https://developer.mozilla.org/en-US/docs/Web/CSS/Media_Queries/Using_media_queries)  **Next.js ⇒>** [**https://nextjs.org/docs/pages/api-reference/create-next-app**](https://nextjs.org/docs/pages/api-reference/create-next-app)  **React-Router Tutorial :** [**https://reactrouter.com/en/main/start/tutorial**](https://reactrouter.com/en/main/start/tutorial)  **React Common props and events :** [**https://react.dev/reference/react-dom/components/common#common-props**](https://react.dev/reference/react-dom/components/common#common-props)  **React Component :** [**https://react.dev/reference/react/Component**](https://react.dev/reference/react/Component)  **React-google charts :** [**https://www.react-google-charts.com/examples**](https://www.react-google-charts.com/examples)  [**https://www.react-google-charts.com/**](https://www.react-google-charts.com/)  **Museum of Modern Art (MoMA) Collection :** [**https://github.com/MuseumofModernArt/collection**](https://github.com/MuseumofModernArt/collection)  **Socket.io :** [**https://socket.io/**](https://socket.io/)  **Socket.io documentation :** [**https://socket.io/docs/v4/**](https://socket.io/docs/v4/)  **Socket.io chat application example :** [**https://socket.io/get-started/chat**](https://socket.io/get-started/chat)  **Firebase to your JavaScript project :** [**https://firebase.google.com/docs/web/setup#available-libraries**](https://firebase.google.com/docs/web/setup#available-libraries)  **Add data to Cloud Firestore :** [**https://firebase.google.com/docs/firestore/manage-data/add-data**](https://firebase.google.com/docs/firestore/manage-data/add-data)  **React UI components library based on the** [**Arco Design**](https://arco.design/) **system :** <https://github.com/arco-design/arco-design>  **Image editor Packages :** [**https://npm.io/search/keyword:image+editor**](https://npm.io/search/keyword:image+editor)  **Currency-converter Packages :** [**https://npm.io/search/keyword:currency-converter**](https://npm.io/search/keyword:currency-converter)  **Anatomy of UseState :** [**https://react.dev/learn/state-a-components-memory#anatomy-of-usestate**](https://react.dev/learn/state-a-components-memory#anatomy-of-usestate)  **Sharing State between components :** [**https://react.dev/learn/sharing-state-between-components**](https://react.dev/learn/sharing-state-between-components)  **DOM Reference :** [**https://developer.mozilla.org/en-US/docs/Web/API/Document\_Object\_Model**](https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model) JavaScript reference,The JavaScript reference serves as a repository of facts about the JavaScript language. The entire language is described here in detail: <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference>JavaScript Guide : <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide>Guide to Semicolons in JavaScript : <https://www.codecademy.com/resources/blog/your-guide-to-semicolons-in-javascript/> **Learn Web Development Resources :** [**https://developer.mozilla.org/en-US/docs/Learn**](https://developer.mozilla.org/en-US/docs/Learn)  **Mouse Events :** <https://legacy.reactjs.org/docs/events.html#mouse-events>  **Mouse event handler :** <https://react.dev/reference/react-dom/components/common#mouseevent-handler>  **Common Components :** <https://react.dev/reference/react-dom/components/common>  **Export ⇒>** A module is a self contained unit that can expose assets to other modules using **export**, and acquire assets from other modules using import.**export means letting the script we wrote to be used by another script. If we say export, we mean any module can use this script by importing it.**Export default means you want to export only one value the is present by default in your script so that others script can import that for use.The export statement is used when creating JavaScript modules **to export functions, objects, or primitive values** from the module so they can be used by other programs with the import statement.  [**Live DOM viewer**](https://software.hixie.ch/utilities/js/live-dom-viewer/) **:** [**https://software.hixie.ch/utilities/js/live-dom-viewer/**](https://software.hixie.ch/utilities/js/live-dom-viewer/)  **Named exports ⇒>** for example export function func() {} is a named export with the name of func. Named modules can be imported using import { exportName } from 'module';. In this case, the name of the import should be the same as the name of the export. To import the func in the example, you'll have to use import { func } from 'module';. There can be multiple named exports in one module.  **Default export** ⇒> is the value that will be imported from the module, if you use the simple import statement import X from 'module'. X is the name that will be given locally to the variable assigned to contain the value, and it doesn't have to be named like the origin export. There can be only one default export. export default prefix is a [standard JavaScript syntax](https://developer.mozilla.org/docs/web/javascript/reference/statements/export) (not specific to React). It lets you mark the main function in a file so that you can later import it from other files. return JavaScript keyword means whatever comes after is returned as a value to the caller of the function.⇒> React components need to return a single JSX element and not multiple adjacent JSX elements like two buttons. To fix this you can use *fragments* (<> and </>) to wrap multiple adjacent JSX elements like this: **export default function Square() {**  **return (**  **<>**  **<button className="square">X</button>**  **<button className="square">X</button>**  **</>**  **);**  **}** var declarations are globally scoped or function scoped while let and const are block scoped. var variables can be updated and re-declared within its scope;let variables can be updated but not re-declared;const variables can neither be updated nor re-declared. ;'method' is the object-oriented word for 'function'. In Object oriented Programming, when a function is part of an object, it is called a method. Example : Walk is a method in person object This ⇒> Value of this is determined by how a function is called (runtime binding). It can't be set by assignment during execution, and it may be different each time the function is called.The [bind()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function/bind) method can [set the value of a function's this regardless of how it's called](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/this#the_bind_method), and [arrow functions](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions) don't provide their own this binding (it retains the this value of the enclosing lexical context) . **bind()** method creates a new function that, when called, has its keyword set to the provided value, with a given sequence of arguments preceding any provided when the new function is called. [**bind()**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function/bind) **method can** [**set the value of a function's this regardless of how it's called**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/this#the_bind_method) arrow function expression is a compact alternative to a traditional [function expression](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/function) ⇒> Arrow functions don't have their own bindings to [this](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/this), [arguments](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/arguments), or [super](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/super), and should not be used as [methods](https://developer.mozilla.org/en-US/docs/Glossary/Method) , Arrow functions cannot be used as [constructors](https://developer.mozilla.org/en-US/docs/Glossary/Constructor). Calling them with [new](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/new) throws a [TypeError](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/TypeError). They also don't have access to the [new.target](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/new.target) keyword , Arrow functions cannot use [yield](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/yield) within their body and cannot be created as generator functions. arrow functions don't rebind the this keyword⇒> when we call a function as a standalone function outside of an object,by default this returns the window objectTemplate literals are literals delimited with backtick (`) characters, allowing for [multi-line strings](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Template_literals#multi-line_strings), [string interpolation](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Template_literals#string_interpolation) with embedded expressions, and special constructs called [tagged templates](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Template_literals#tagged_templates). It provides an easy way to create multiline strings and perform string interpolation. Template literals are the string literals and allow embedded expressions. With template literals, you can avoid the concatenation operator — and improve the readability of your code — by using placeholders of the form ${expression} to perform substitutions for embedded expressions:Importing and exporting components ⇒> You can declare many components in one file, but large files can get difficult to navigate. To solve this, you can *export* a component into its own file, and then *import* that component from another file .A file can have no more than one *default* export, but it can have as many *named* exports as you like.People often use default exports if the file exports only one component, and use named exports if it exports multiple components and values.Components without names, like export default () => {}, are discouraged because they make debugging harder.To reduce the potential confusion between default and named exports, some teams choose to only stick to one style (default or named), or avoid mixing them in a single file. Writing markup with JSX ⇒> Each React component is a JavaScript function that may contain some markup that React renders into the browser. React components use a syntax extension called JSX to represent that markup.In React, rendering logic and markup live together in the same place—components.React components group rendering logic together with markup because they are related.JavaScript in JSX with curly braces ⇒> Sometimes you will want to add a little JavaScript logic or reference a dynamic property inside that markup. In this situation, you can use curly braces in your JSX to “open a window” to JavaScript . JSX is a special way of writing JavaScript. That means it’s possible to use JavaScript inside it—with curly braces { }.Any JavaScript expression will work between curly braces, including function calls like formatDate() . You can only use curly braces in two ways inside JSX:As text directly inside a JSX tag: <h1>{name}'s To Do List</h1> works, but <{tag}>Gregorio Y. Zara's To Do List</{tag}> will not.As attributes immediately following the = sign: src={avatar} will read the avatar variable, but src="{avatar}" will pass the string "{avatar}".double Curly Braces ⇒> To pass a JS object in JSX, you must wrap the object in another pair of curly braces: person={{ name: "Hedy Lamarr", inventions: 5 }}.Passing props to a component ⇒> React components use *props* to communicate with each other. Every parent component can pass some information to its child components by giving them props. Props might remind you of HTML attributes, but you can pass any JavaScript value through them, including objects, arrays, functions, and even JSX! . Props are the information that you pass to a JSX tag. For example, className, src, alt, width, and height are some of the props you can pass to an <img> .You can think of props like “knobs” that you can adjust. They serve the same role as arguments serve for functions—in fact, props *are* the only argument to your component! React component functions accept a single argument, a props object . Usually you don’t need the whole props object itself, so you destructure it into individual props.Don’t miss the pair of { and } curlies inside of ( and ) when declaring props . Props are [immutable](https://en.wikipedia.org/wiki/Immutable_object)—a term from computer science meaning “unchangeable”. When a component needs to change its props (for example, in response to a user interaction or new data), it will have to “ask” its parent component to pass it *different props*—a new object! Its old props will then be cast aside, and eventually the JavaScript engine will reclaim the memory taken by them. Don’t try to “change props” = When you need to respond to the user input (like changing the selected color), you will need to “set state” . We can change the value we pass to the component but within the body of the function we should never change props .Specifying a default value for a prop⇒> If you want to give a prop a default value to fall back on when no value is specified, you can do it with the destructuring by putting = and the default value right after the parameter: function Avatar({ person, size = 100 }) {  // ...  } Read props inside the child component ⇒> You can read these props by listing their names, person, size separated by the commas inside ({ and }) directly after function Avatar. This lets you use them inside the Avatar code, like you would with a variable. **function Avatar({ person, size }) {**  **// person and size are available here**  **}**  **⇒> A module can contain both named exports and a default export, and they can be imported together using import defaultExport, { namedExport1, namedExport3, etc... } from 'module';**  **How do you check margins in Figma? Measure distances between nested layers**   1. **Select the first object in the canvas.** 2. **Hold down the modifier keys: MacOS: ⌘ Command ⌥ Option. Windows: Ctrl Alt.** 3. **Hover over the second object.** 4. **Figma will display a red line between the two objects, as well as a measurement.**   **Padding-block** ⇒> Padding on top and bottom in one go . An element's padding-block is the space from its border to its content in the block direction, and it is a shorthand property for the following properties:   * [padding-block-start](https://www.w3schools.com/cssref/css_pr_padding-block-start.php) * [padding-block-end](https://www.w3schools.com/cssref/css_pr_padding-block-end.php)  ::marker ⇒> The ::marker [CSS](https://developer.mozilla.org/en-US/docs/Web/CSS) [pseudo-element](https://developer.mozilla.org/en-US/docs/Web/CSS/Pseudo-elements) selects the marker box of a list item, which typically contains a bullet or number. This selector selects the marker of a list item . **background-repeat** property sets if/how a background image will be repeated.By default, a [background-image](https://www.w3schools.com/cssref/pr_background-image.asp) is repeated both vertically and horizontally. Background-blend-mode ⇒> The background-blend-mode [CSS](https://developer.mozilla.org/en-US/docs/Web/CSS) property sets how an element's background images should blend with each other and with the element's background color.   **background-image** [CSS](https://developer.mozilla.org/en-US/docs/Web/CSS) property sets one or more background images on an element  **background-position** [CSS](https://developer.mozilla.org/en-US/docs/Web/CSS) property sets the initial position for each background image  **React** is a library not a framework**.React** enables developers to declaratively describe native applications . React focuses on one thing , building user interfaces . React is declarative , it means that we describe user interfaces with React and tell it what we want , not how to do it.React will take care of the how and translate our declarative descriptions which we write in React language to actual user interfaces in the browser . React enables us to declaratively describe user interfaces and not all the state of these interfaces . Developers just describe the interfaces in terms of a state like a function , when transactions happen to that state React takes care of updating the user interfaces based on that . React is declarative language(model UI and state).  **React** uses the virtual DOM to compare versions of the UI in memory before it acts on them .  **Disadvantages of Framework ⇒>** Do things a certain way . Hard to deviate.Hard to customize . You have to use the whole Framework . camelCase ~~all~~ most of the things! ⇒> JSX turns into JavaScript and attributes written in JSX become keys of JavaScript objects. In your own components, you will often want to read those attributes into variables. But JavaScript has limitations on variable names. For example, their names can’t contain dashes or be reserved words like class. This is why, in React, many HTML and SVG attributes are written in camelCase. For example, instead of stroke-width you use strokeWidth. Since class is a reserved word, in React you write className instead, named after the [corresponding DOM property](https://developer.mozilla.org/en-US/docs/Web/API/Element/className) .className ⇒> **The className property of the** [**Element**](https://developer.mozilla.org/en-US/docs/Web/API/Element) **interface gets and sets the value of the** [**class attribute**](https://developer.mozilla.org/en-US/docs/Web/HTML/Global_attributes/class) **of the specified element.**Element ⇒> Element is the most general base class from which all element objects (i.e. objects that represent elements) in a [Document](https://developer.mozilla.org/en-US/docs/Web/API/Document) inherit. It only has methods and properties common to all kinds of elements. More specific classes inherit from Element. For example, the [HTMLElement](https://developer.mozilla.org/en-US/docs/Web/API/HTMLElement) interface is the base interface for HTML elements, while the [SVGElement](https://developer.mozilla.org/en-US/docs/Web/API/SVGElement) interface is the basis for all SVG elements. Most functionality is specified further down the class hierarchy.HTMLElement ⇒> The HTMLElement interface represents any [HTML](https://developer.mozilla.org/en-US/docs/Web/HTML) element. Some elements directly implement this interface, while others implement it via an interface that inherits it. **Components** ⇒> With React we describe user interfaces using components .. You can think of components as just functions . Components receive certain input objects and output description of a user interface . We can reuse a single component in multiple user interfaces and components can contain other components . When the state of the React component(the input) changes , the user interface it represents , the output changes as well . To build HTMl web applications with React we don’t write HTML at all , we use javascript to generate HTML . React components can be of two types: function or class component .React lets you combine your markup, CSS, and JavaScript into custom “components”, **reusable UI elements for your app.** React component is a JavaScript function that you can *sprinkle with markup*.Components are regular JavaScript functions, so you can keep multiple components in the same file.You can define a component once, and then use it in as many places and as many times as you like.Components can render other components, but **you must never nest their definitions .Never define a component inside another component. React component names must start with a capital letter.** Reconciliation ⇒> DOM(Document Object Model) is the programming interface for the tree structure web page documents. The document tree is called the DOM Tree.The mechanism to diff one tree with another to determine which parts need to be changed and then update the original DOM with it is called Reconciliation. **ReactDOM.render()** function takes two arguments, HTML code and an HTML element.The purpose of the function is to display the specified HTML code inside the specified HTML element.But render where? For example, there is an index.html file.You'll notice a single <div> in the body of this file. This is where our React application will be rendered:  **import React from 'react';**  **import ReactDOM from 'react-dom/client';**  **ReactDOM.render(<p>Hello</p>, document.getElementById('root'));**  **Babel** is a compiler to convert JSX into React API calls .  **useState** is a React Hook that lets you add a [state variable](https://beta.reactjs.org/learn/state-a-components-memory) to your component.Call useState at the top level of your component to declare a [state variable.](https://beta.reactjs.org/learn/state-a-components-memory)useState is a Hook, so you can only call it **at the top level of your component** or your own Hooks. You can’t call it inside loops or conditions. If you need that, extract a new component and move the state into it : **const [state, setState] = useState(initialState)**  useState returns an array with exactly two values:   1. The current state. During the first render, it will match the initialState you have passed. 2. The [set function](https://beta.reactjs.org/reference/react/useState#setstate) that lets you update the state to a different value and trigger a re-render.When you call a set function in a component, React automatically updates the child components inside too.  useState ⇒> When you call [useState](https://react.dev/reference/react/useState), you are telling React that you want this component to remember something: const [index, setIndex] = useState(0); , In this case, you want React to remember the index . Every time your component renders, useState gives you an array containing two values:The state variable (index) with the value you stored.The state setter function (setIndex) which can update the state variable and trigger React to render the component again.Note : State is fully private to the component declaring it.If you render it in two places, each copy gets its own state.Sharing State Between Components ⇒> Sometimes, you want the state of two components to always change together. To do it, remove state from both of them, move it to their closest common parent, and then pass it down to them via props. This is known as *lifting state up .*State: A Component's Memory ⇒> Components often need to change what’s on the screen as a result of an interaction. Typing into the form should update the input field, clicking “next” on an image carousel should change which image is displayed, clicking “buy” should put a product in the shopping cart. Components need to “remember” things: the current input value, the current image, the shopping cart. In React, this kind of component-specific memory is called *state*. **⇒> To collect data from multiple children, or to have two child components communicate with each other, declare the shared state in their parent component instead. The parent component can pass that state back down to the children via props. This keeps the child components in sync with each other and with their parents.**  **⇒> State is private to a component that defines it .Calling the setSquares function lets React know the state of the component has changed.**  **Closures ⇒>** A closure is the combination of a function bundled together (enclosed) with references to its surrounding state (the lexical environment). In other words, **a closure gives you access to an outer function's scope from an inner function.** In JavaScript, closures are created every time a function is created, at function creation time.  **⇒> JavaScript supports** [**closures**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Closures) **which means an inner function (e.g. handleClick) has access to variables and functions defined in an outer function (e.g. Board). The handleClick function can read the squares state and call the setSquares method because they are both defined inside of the Board function.**  **⇒> The DOM <button> element’s onClick attribute has a special meaning to React because it is a built-in component.**  **⇒> In React, it’s conventional to use onSomething names for props which represent events and handleSomething for the function definitions which handle those events.** Immutability ⇒> An ability to undo and redo certain actions is a common requirement for apps. Avoiding direct data mutation lets you keep previous versions of the data intact, and reuse them later.There is also another benefit of immutability. By default, all child components re-render automatically when the state of a parent component changes. This includes even the child components that weren’t affected by the change. Although re-rendering is not by itself noticeable to the user (you shouldn’t actively try to avoid it!), you might want to skip re-rendering a part of the tree that clearly wasn’t affected by it for performance reasons. Immutability makes it very cheap for components to compare whether their data has changed or not.memo() ⇒> memo lets you skip re-rendering a component when its props are unchanged.const MemoizedComponent = memo(SomeComponent, arePropsEqual?) **JSX supports dynamic expressions if you place them within curly braces anywhere inside JSX :** Notice curly braces after onClick where we put setCounter inside : <button onClick={() => setCounter(counter\*2)}>{counter}</button>;  **function Button() {**  **const [counter, setCounter] = useState(5);**  **return <button onClick={() => setCounter(counter\*2)}>{counter}</button>;**  **}**  **⇒> We need JSX to be nested under a single parent like a <div></div>**  **Fragments ⇒>** A common pattern in React is for a component to return multiple elements. Fragments let you group a list of children without adding extra nodes to the DOM.Shorter syntax you can use for declaring fragments. It looks like empty tags:  **class Columns extends React.Component {**  **render() {**  **return (**  **<>**  **</>**  **);} }**  **Props** are arguments passed into React components.**Props are sent by the parent component to the children component hence children components cannot make changes to these props**..Props are passed to components via HTML attributes.**Props** stands for properties.**React Props** are like function arguments in JavaScript *and* attributes in HTML.To send props into a component, use the same syntax as HTML attributes.The component receives the argument as a props object .For example : Add a "brand" attribute to the Car element: const myElement = <Car brand="Ford" />;  Use the brand attribute in the component:  function Car(props) {  return <h2>I am a { props.brand }!</h2>;  }  **Props** can hold functions as well . Functions are just objects in javascript .**Props are like passing parameters into a function that function uses to help it determine the output that gets returned . You want to use props anytime you need to pass data into a component ,so that component can use that data .Don’t change props within the component that’s receiving props. We want to use props instead of state when we want to pass data into a component so that component can determine what will get displayed on the screen.Props are immutable.**  **Where to define state in a React application :**  down in a tree as close as possible to the children who need to access that value on the state .  ⇒> React application is a set of reusable components. Components are just like functions. They take input and they output a description of a user interface in the form of a React element. The ReactDOM library enables us to render those React elements in the browser, and it will rerender them for us automatically when their in‑memory state changes. To accomplish this, we write the component's markup using the React JavaScript API.  ⇒> Writing HTML in JavaScript is a lot different and React has a way to write the virtual DOM in a syntax very close to the HTML syntax we're used to. This special React syntax is called JSX. Once we have the virtual DOM description in JSX, we can pre‑transform it to valid React API calls before shipping it to the browser. Browsers do not have to deal with JSX.  ⇒> The input for a component is a set of properties you can access inside the component with its first argument object, which is usually named props, and also a set of state elements that a component can hook into with the special useState function.  ⇒> A component state can be changed inside that component, and every time a component changes its state, React rerenders it. The props of a component cannot be changed by the component, but the whole component can be rerendered with different props by the component's parent.  ⇒> The syntax to mount a React component in the browser is ReactDOM.render, and that takes two arguments, the component to render and the HTML element to hold the React‑rendered markup. React also comes with normalized events that work across all browsers in a standard way.  ⇒> React actually has two types of components, function and class components. ECMAScript ⇒> The specification defined in ECMA-262 for creating a general purpose scripting language.Block ⇒> A block statement is used to group zero or more statements. The block is delimited by a pair of braces ("curly brackets") and contains a list of zero or more statements and declarations.If you want to use more than one statement in the loop body, you can group them into one block statement.JavaScript allows nested blocks and therefore nested scopes. Nested scopes create a scope tree or scope chain..[Syntax](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/block#syntax) **{**  **StatementList**  **}**  JavaScript also allows us to nest functions:  **function foo(bar) {**  **function baz() {**  **console.log(bar);**  **}**  **baz();**  **}**  **foo("bar"); // "bar"** Var ⇒> Variables created with var are scoped to their nearest function or the global scope . They are not block scoped.var can create confusing situations .It is best to use let and const when possible. ⇒> Scalers(int or string) in javascript are immutable , we can’t mutate the value of a string or an integer in javascript . When we use const with these scalar values then we can’t change the references either . Content of an Array or object can be mutated in const.  ⇒> We use const when a reference assigned to a variable is meant to be a constant one .  **⇒>** Arrow functions allow us to write shorter function syntax:  let myFunction = (a, b) => a \* b; Arrow function ⇒> A regular function always binds the value for its “this” keyword for its caller . The value of the “this” keyword inside an arrow function depends on where the function was defined . An arrow function will close over the value of the “this” keyword for its scope at the time it was defined . **With arrow functions there are no binding of this.In regular functions the “this” keyword represents the object that called the function, which could be the window, the document, a button or whatever. With arrow functions the “this” keyword *always* represents the object that defined the arrow function."this" keyword represents the object that owns the function, no matter who calls the function. Arrow functions don't have their own bindings to** [**this**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/this)**,** [**arguments**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/arguments)**, or** [**super**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/super)**, and should not be used as** [**methods**](https://developer.mozilla.org/en-US/docs/Glossary/Method)**. Regular functions give access to their "calling" environment while arrow functions give access to their "defining" environment**  . The value of the "this" keyword inside a regular function depends on HOW the function was CALLED (the OBJECT that made the call).The value of the "this" keyword inside an arrow function depends on WHERE the function was DEFINED (the SCOPE that defined the function). Arrow functions implicitly return the expression right after =>, so you don't need a return statement . However, you must write return explicitly if your => is followed by a { curly brace . Arrow functions containing => { are said to have a [“block body”.](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions#function_body) They let you write more than a single line of code, but you *have to* write a return statement yourself. If you forget it, nothing gets returned! **Object** initializers are also called object literals. "Object initializer" is consistent with the terminology used by C++ .Each property name before colons is an identifier (either a name, a number, or a string literal), and each valueN is an expression whose value is assigned to the property name.The syntax for an object using an object initializer is:  const obj = {  property1: value1, // property name may be an identifier  2: value2, // or a number  "property n": value3, // or a string  }; Destructuring ⇒> To illustrate destructuring, we'll make a sandwich. Do you take everything out of the refrigerator to make your sandwich? No, you only take out the items you would like to use on your sandwich.Destructuring is exactly the same. We may have an array or object that we are working with, but we only need some of the items contained in these.Destructuring makes it easy to extract only what is needed.Example : **function calculate(a, b) {**  **const add = a + b;**  **const subtract = a - b;**  **const multiply = a \* b;**  **const divide = a / b;**  **return [add, subtract, multiply, divide];**  **}**  **const [add, subtract, multiply, divide] = calculate(4, 7);**  **rest(...)** operator puts the rest of some specific user-supplied values into a JavaScript array. Example :  // Use rest to enclose the rest of specific user-supplied values into an array:  **function myBio(firstName, lastName, ...otherInfo) {**  **return otherInfo;**  **}**  // Invoke myBio function while passing five arguments to its parameters:  **myBio("Oluwatobi", "Sofela", "CodeSweetly", "Web Developer", "Male");**  // The invocation above will return:  **["CodeSweetly", "Web Developer", "Male"]**  **spread** (...) syntax allows an iterable, such as an array or string, to be expanded in places where zero or more arguments (for function calls) or elements (for array literals) are expected. **spread(...)** syntax expands iterables into individual elements.Example :  // Define a function with three parameters:  **function myBio(firstName, lastName, company) {**  **return `${firstName} ${lastName} runs ${company}`;**  **}**  // Use spread to expand an array’s items into individual arguments:  **myBio(...["Oluwatobi", "Sofela", "CodeSweetly"]);**  // The invocation above will return:  “Oluwatobi Sofela runs CodeSweetly”  **Template Literals(Template strings)** use back-ticks (``) rather than the quotes ("") to define a string . Template literals provide an easy way to interpolate variables and expressions into strings.The method is called string interpolation.The syntax is: ${...} . Example :  const a = 5;  const b = 10;  console.log(`Fifteen is ${a + b} and  not ${2 \* a + b}.`);  **Promises** are the foundation of asynchronous programming in modern JavaScript. A promise is an object returned by an asynchronous function, which represents the current state of the operation. At the time the promise is returned to the caller, the operation often isn't finished, but the promise object provides methods to handle the eventual success or failure of the operation.  **this** ⇒> In class components both the props and the state are managed on an instance of the class . When we are creating a class we are creating instances of objects and each instance gets its props and state . **‘this’** keyword typically references a JavaScript element depending on the scope or context of its use.  **React.Component** ⇒> React lets you define components as classes or functions. Components defined as classes currently provide more features. To define a React component class, you need to extend React.Component .  **<Card {...profile}/>** ⇒> When we use a spread(...) operator like this in the React component ,all the properties of that object will become props for this component .  ⇒> Everytime we use a class component , React internally creates a instance from the component and uses it to render the element like this <Card {...profile}/>)  **map()**Returns a new array. Whatever gets returned from the callback  function provided is placed at the same index in the new array.  Usually we take the items from the original array and modify them  in some way.**map()** method creates a new array populated with the results of calling a provided function on every element in the calling array.map() calls a function once for each element in an array.Map returns a new array without modifying the original that will have the same length as the original array. Example , map() function to take an array of numbers and double their values. We assign the new array returned by map() to the variable doubled and log it:  const numbers = [1, 2, 3, 4, 5];  const doubled = numbers.map((number) => number \* 2);  console.log(doubled)  **React** components have a built-in state object. The state object is where you store property values that belong to the component. When the state object changes, the component re-renders. Example : <CardList profiles={this.state.profiles}  ⇒> Every React function receives an event argument like event(event) in brackets is event here.You can name this event anything , it doesn’t have to be event :  handleSubmit = (event) => {  event.preventDefault();  console.log(this.state.userName);  };  **preventDefault()** method cancels the event if it is cancelable, meaning that the default action that belongs to the event will not occur. For example, this can be useful when:Clicking on a "Submit" button, prevent it from submitting a form , Clicking on a link, prevent the link from following the URL.For example , here we want to takeover or override the HTML submit logic :  handleSubmit = (event) => {  event.preventDefault();  console.log(this.state.userName);  };  ⇒> Without preventDefault() if you submit the form your page will refresh .  **Refs** provide a way to access DOM nodes or React elements created in the render method. In the typical React dataflow, [props](https://reactjs.org/docs/components-and-props.html) are the only way that parent components interact with their children. To modify a child, you re-render it with new props. However, there are a few cases where you need to imperatively modify a child outside of the typical dataflow. The child to be modified could be an instance of a React component, or it could be a DOM element. For both of these cases, React provides an escape hatch.There are a few good use cases for refs:Managing focus, text selection, or media playback , Triggering imperative animations , Integrating with third-party DOM libraries.Refs are created using **React.createRef()** and attached to React elements via the ref attribute. Refs are commonly assigned to an instance property when a component is constructed so they can be referenced throughout the component.When a ref is passed to an element in render, a reference to the node becomes accessible at the **current** attribute of the ref.  ⇒> Conceptually, components are like JavaScript functions. They accept arbitrary inputs (called “props”) and return React elements describing what should appear on the screen.Whether you declare a Component as a function or a class, it must never modify its own props.All React Componentsmust act like pure functions with respect to their props. This means functions that never try to change their inputs and always return the same result for the same inputs.  **setState()** ⇒> If you need to give the user the opportunity to input something or in some way change the variables the component is receiving as props, you’ll need setState.State allows React components to change their output over time in response to user actions, network responses, and anything else, without violating this rule.Do Not Modify State Directly  [**event.target**](https://developer.mozilla.org/en-US/docs/Web/API/Event/target) gives you the element **that** triggered the event.So, **event.target.value** retrieves the **value** of that element .In React, events are [SynthenticEvent](https://reactjs.org/docs/events.html), a wrapper around the browser’s native event. It has the same interface as the browser’s native event, including stopPropagation() and preventDefault(), except the events work identically across all browsers.  **SyntheticEvent()** ⇒> Your event handlers will be passed instances of SyntheticEvent, a cross-browser wrapper around the browser’s native event. It has the same interface as the browser’s native event, including stopPropagation() and preventDefault(), except the events work identically across all browsers. event.nativeEvent ⇒> Access the browser's native Event in a React code ? In most cases, you don't need to. But you can use nativeEvent property of SyntheticEvent i.e. event.nativeEvent. **Axios is a library which is used to make requests to an API, return data from the API, and then do things with that data in our React application.Axios** is an HTTP client library that allows you to make requests to a given endpoint . To perform a GET request, you use the **.get()** method.Axios is a promise-based HTTP library that lets developers make requests to either their own or a third-party server to fetch data. It offers different ways of making requests such as GET , POST , PUT/PATCH , and DELETE .  **key prop⇒> React's key prop gives you the ability to control component instances. Each time React renders your components, it's calling your functions to retrieve the new React elements that it uses to update the DOM. If you return the same element types, it keeps those components/DOM nodes around, even if all the props changed.** A “key” is a special string attribute you need to include when creating lists of elements in React. Keys are used in React to identify which items in the list are changed, updated, or deleted. In other words, we can say that keys are used to give an identity to the elements in the lists. ⇒> Convert React Class Components to Functional Components with React Hooks : Here are the steps:use function instead of classremove the constructorremove the render() method, keep the returnadd const before all methodsremove this.state throughout the componentremove all references to ‘this’ throughout the componentSet initial state with useState()change this.setState() … instead, call the function that you named in the previous step to update the state…replace componentDidMount with useEffectreplace componentDidUpdate with useEffect ⇒> Whenever you identify a data element that’s used in the UI and is going to change value , you should make it a state element .  ⇒> important point to identify a candidate for a component : Every time in the UI you have many times that share similar data and behavior , that’s a candidate for an item component .  **closure** is the combination of a function bundled together (enclosed) with references to its surrounding state (the lexical environment). In other words, a closure gives you access to an outer function's scope from an inner function. In JavaScript, closures are created every time a function is created, at function creation time.Global variables can be made local (private) with closures.A closure is a function having access to the parent scope, even after the parent function has closed.  **Closure** example , each onClick function closes over the scope of it’s owner number and gives access to it’s props :  const PlayNumber = props => (  <button className="number" onClick={() => console.log('Num', props.number)}>  {props.number}  </button>  );  **filter()** method creates a [shallow copy](https://developer.mozilla.org/en-US/docs/Glossary/Shallow_copy) of a portion of a given array, filtered down to just the elements from the given array that pass the test implemented by the provided function. filter() method takes an array of items, passes them through a “test” (a function that returns true or false), and returns a new array of only those items that passed the test (returned true).  **useEffect** Hook allows you to perform side effects in your components.Some examples of side effects are: fetching data, directly updating the DOM, and timers.useEffect accepts two arguments. The second argument is optional.  useEffect(<function>, <dependency>)  **⇒> Whenever you create a side effect , you have to clean that side effect when it’s no longer needed .**  **⇒> You always use the React hooks function in the same order , so you can’t define them conditionally like an if statement .You can use conditionals like if inside the call of the hook but not outside .So you can’t conditionally use the hooks .**  **⇒> Try to name your custom hook function with the word “use” , I mean star your function name with word “use” , so linters and code formatters can take a hint that it’s a custom hook .**  **props** Make a component more reusable.  **⇒>** Can I pass a custom prop (e.g. `blahblahblah={true}`) to a native DOM element? (e.g. <div blahblahblah={true}>) Why or why not? ⇒> No, because the JSX we use to describe native DOM elements will be turned into REAL DOM elements by React. And real DOM elements only have the properties/attributes specified in the HTML specification.(Which doesn't include properties like `blahblahblah`)  **⇒>** What data type is `props` when the component receives it? : An object!  **<BrowserRouter>** stores the current location in the browser's address bar using clean URLs and navigates using the browser's built-in history stack. Array.prototype.map() ⇒> The map() method creates a new array populated with the results of calling a provided function on every element in the calling array.Returns a new array. Whatever gets returned from the callback function provided is placed at the same index in the new array.Usually we take the items from the original array and modify them in some way.Convert an array of raw data into an array of JSX elements that can be displayed on the page..Example : **const array1 = [1, 4, 9, 16];**  **// Pass a function to map**  **const map1 = array1.map(x => x \* 2);**  **console.log(map1);** Spread syntax (...) ⇒> The spread (...) syntax allows an iterable, such as an array or string, to be expanded in places where zero or more arguments (for function calls) or elements (for array literals) are expected. **onClick() ⇒>** React onClick event handler enables you to call a function and trigger an action when a user clicks an element, such as a button, in your app.Event names are written in camelCase, so the onclick event is written as onClick in a React app. In addition, [React event handlers](https://reactjs.org/docs/handling-events.html) appear inside curly braces.  **⇒> With JSX you pass a function as the event handler, rather than a string .**  **preventDefault() ⇒> You cannot return false to prevent default behavior in React. You must call preventDefault explicitly.**  **function Form() {**  **function handleSubmit(e) {**  **e.preventDefault();**  **console.log('You clicked submit.');**  **}** Conditional Rendering ⇒> Your components will often need to display different things depending on different conditions. In React, you can conditionally render JSX using JavaScript syntax like if statements, &&, and ? : operators.More often, you would conditionally include or exclude the component in the parent component’s JSX. A [JavaScript && expression](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Logical_AND) returns the value of its right side . React considers false as a “hole” in the JSX tree, just like null or undefined, and doesn’t render anything in its place.⇒> React considers false as a “hole” in the JSX tree, just like null or undefined, and doesn’t render anything in its place. **Don’t put numbers on the left side of && ⇒>** To test the condition, JavaScript converts the left side to a boolean automatically. However, if the left side is 0, then the whole expression gets that value (0), and React will happily render 0 rather than nothing.For example, a common mistake is to write code like messageCount && <p>New messages</p>. It’s easy to assume that it renders nothing when messageCount is 0, but it really renders the 0 itself! . To fix it, make the left side a boolean: messageCount > 0 && <p>New messages</p>. Conditional (ternary) operator ⇒> The conditional (ternary) operator is the only JavaScript operator that takes three operands: a condition followed by a question mark (?), then an expression to execute if the condition is [truthy](https://developer.mozilla.org/en-US/docs/Glossary/Truthy) followed by a colon (:), and finally the expression to execute if the condition is [falsy](https://developer.mozilla.org/en-US/docs/Glossary/Falsy). This operator is frequently used as an alternative to an [if...else](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/if...else) statement. In JSX, {cond ? <A /> : <B />} means *“if cond, render <A />, otherwise <B />”*. In JSX, {cond && <A />} means *“if cond, render <A />, otherwise nothing”*.Rendering lists ⇒> You will often need to show several instances of the same component using different data when building interfaces: from lists of comments to galleries of profile images. In these situations, you can store that data in JavaScript objects and arrays and use methods like [map()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/map) and [filter()](https://developer.mozilla.org/docs/Web/JavaScript/Reference/Global_Objects/Array/filter) to render lists of components from them. You will often want to display multiple similar components from a collection of data. You can use JavaScript’s filter() and map() with React to filter and transform your array of data into an array of components. For each array item, you will need to specify a key. Usually, you will want to use an ID from the database as a key. Keys let React keep track of each item’s place in the list even if the list changes.Keeping components pure ⇒> Some JavaScript functions are *pure.* A pure function:  * Minds its own business. It does not change any objects or variables that existed before it was called. * Same inputs, same output. Given the same inputs, a pure function should always return the same result.  Logical AND operator (&&) ⇒> Another common shortcut you’ll encounter is the [JavaScript logical AND (&&) operator.](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Logical_AND#:~:text=The%20logical%20AND%20(%20%26%26%20)%20operator,it%20returns%20a%20Boolean%20value.) Inside React components, it often comes up when you want to render some JSX when the condition is true, or render nothing otherwise. With &&, you could conditionally render the checkmark only if isPacked is true: **return (**  **<li className="item">**  **{name} {isPacked && '✔'}**  **</li>**  **);**  **Short circuit evaluation ⇒>** **(if this part is true) && (this part will execute)**  **fetch()** method allows us to make an HTTP request to the backend. With this method, we can perform different types of operations using HTTP methods like the GET method to request data from an endpoint, POST to send data to an endpoint, and more.In React, we should avoid performing side effects directly within the component body to avoid inconsistencies. Instead, we can isolate them from the rendering logic [using the useEffect Hook](https://blog.logrocket.com/guide-to-react-useeffect-hook/).We will fetch our data in the Hook like so:  **useEffect(() => {**  **// data fetching here**  **}, []);**  **fetch() example ⇒>** We are using the fetch() method to request data from the resource endpoint. This operation returns a promise that could either resolve or reject.If it resolves, we handle the response using .then() . Next, we must resolve the Response object to JSON format using the json() method. This also returns a promise and from there, we can resolve to get the actual data that we need:  **useEffect(() => {**  **fetch("https://type.fit/api/quotes")**  **.then((res) => res.json())**  **.then((json) => {**  **setQuotes(json);**  **setQuote(json[0]);**  **});**  **}, []);**  **onSubmit()** is an event handler attached to the form submission event <form onSubmit={onSubmit}> . React invokes onSubmit() handler when the form is submitted, i.e. the user clicks the Submit button. If the form validation fails, then onSubmit() event handler is not invoked. Debounce ⇒> The term debounce comes from electronics. When you’re pressing a button, let’s say on your TV remote, the signal travels to the microchip of the remote so quickly that before you manage to release the button, it bounces, and the microchip registers your “click” multiple times.Debounce in JavaScript is similar. We want to trigger a function, but only once per use case.Let's say that we want to show suggestions for a search query, but only after a visitor has finished typing it.Or we want to save changes on a form, but only when the user is not actively working on those changes, as every "save" costs us a database trip. **Lodash** makes JavaScript easier by taking the hassle out of working with arrays, numbers, objects, strings, etc.Lodash’s modular methods are great for:   * Iterating arrays, objects, & strings * Manipulating & testing values * Creating composite functions  Lodash.debounce ⇒> conditions for our debouncing, like immediately invoking a function if something happens, or invoking a function on the leading or trailing edge of the wait timeout, and even many other conditions that might be needed for other use cases and , [Lodash](https://lodash.com/) has a debounce method which can accomplish all these tasks . **lifting state up ⇒>** We lift up state to a common ancestor of components that need it, so that they can all share in the state. This allows us to more easily share state among all of these components that need to rely upon it.What common ancestor should you lift up your state to so all of the components can read from and update that state? The App component.Sometimes, you want the state of two components to always change together. To do it, remove state from both of them, move it to their closest common parent, and then pass it down to them via props. When you want to coordinate two components, move their state to their common parent.   * Then pass the information down through props from their common parent. * Finally, pass the event handlers down so that the children can change the parent’s state. * It’s useful to consider components as “controlled” (driven by props) or “uncontrolled” (driven by state).   **useMemo** is a React Hook that lets you cache the result of a calculation between re-renders.Call useMemo at the top level of your component to cache a calculation between re-renders.On the initial render, useMemo returns the result of calling calculateValue with no arguments.During next renders, it will either return an already stored value from the last render (if the dependencies haven’t changed), or call calculateValue again, and return the result that calculateValue has returned.  **Range** Slider is a custom range-type HTML5 input component. It allows you to select a value or range of values between a specified min and max.  **fill()** method fills specified elements in an array with a value.The fill() method overwrites the original array.Start and end position can be specified. If not, all elements will be filled.  **Key ⇒>** React stores some information about each rendered list item. When you update a list, React needs to determine what has changed. When a list is re-rendered, React takes each list item’s key and searches the previous list’s items for a matching key. If the current list has a key that didn’t exist before, React creates a component. If the current list is missing a key that existed in the previous list, React destroys the previous component. If two keys match, the corresponding component is moved. Keys tell React about the identity of each component, which allows React to maintain state between re-renders. If a component’s key changes, the component will be destroyed and re-created with a new state. **key is a special and reserved property in React. When an element is created, React extracts the key property and stores the key directly on the returned element. Even though the key may look like it is passed as props, React automatically uses the key to decide which components to update. There’s no way for a component to ask what key its parent specified. It’s strongly recommended that you assign proper keys whenever you build dynamic lists.**  If no key is specified, React will report an error and use the array index as a key by default. Using the array index as a key is problematic when trying to reorder a list’s items or inserting/removing list items. Explicitly passing key={i} silences the error but has the same problems as array indices and is not recommended in most cases. **Keys do not need to be globally unique; they only need to be unique between components and their siblings.**  **web3**.js library is a collection of modules that contain functionality for the ethereum ecosystem.  **Material UI** is an open-source React component library that implements Google's [Material Design](https://m2.material.io/). |
| Summary | **Glassmorphism ⇒>** Glassmorphism is a UI design trend that imitates the look of frosted glass. Elements in the UI look like they are placed on sheets of glass, typically set above a contrasting background. Essentially, the main aspect of this trend is a semi-transparent background, with a sublime shadow and border. But you also have a blur added to the background itself so that whatever is behind the background is beautifully "morphed" into the element itself.  [**uuid v4**](https://www.npmjs.com/package/uuid) will create the unique ID Glassmorphism - CSS Glass Effect - Code : **background: rgba(255, 255, 255, 0.33);**  **border-radius: 16px;**  **box-shadow: 0 4px 30px rgba(0, 0, 0, 0.1);**  **backdrop-filter: blur(8.1px);**  **-webkit-backdrop-filter: blur(8.1px);**  **border: 1px solid rgba(255, 255, 255, 0.3);**  CS Grid-template-columns ⇒> The grid-template-columns CSS property defines the line names and track sizing functions of the [grid columns](https://developer.mozilla.org/en-US/docs/Glossary/Grid_Column).LinearProgress ⇒> LinearProgress bar . If the progress bar is describing the loading progress of a particular region of a page, you should use aria-describedby to point to the progress bar, and set the aria-busy attribute to true on that region until it has finished loading.JSON.stringify() ⇒> The JSON.stringify() static method converts a JavaScript value to a JSON string . **localStorage** mechanism is available via the [Window.localStorage](https://developer.mozilla.org/en-US/docs/Web/API/Window/localStorage) property. Window.localStorage is part of the [Window](https://developer.mozilla.org/en-US/docs/Web/API/Window) interface in JavaScript, which represents a window containing a DOM document. The Window interface features a wide range of functions, constructors, objects, and namespaces. Window.localStorage is a read-only property that returns a reference to the local storage object used to store data that is only accessible to the origin that created it.While you can store small amounts of data with localStorage, it’s not suitable for large amounts of data. localStorage is accessible to anyone who uses the device, so you shouldn’t use it to store sensitive information. You can use it to store user preferences like the language or theme. You can also use it to cache data if you use it frequently. localStorage can store form data that won’t be lost if the user closes the browser. If you have an application that requires you to log in, localStorage can be used to keep your session data. You can remain logged in even after closing and reopening the browser. So, basically, localStoage is used for storing and retrieving data.  **Hooks** let you use state and other React features without writing a class.***Hooks*** let you use different React features from your components. You can either use the built-in Hooks or combine them to build your own. State Hooks ⇒> *State* lets a component [“remember” information like user input.](https://react.dev/learn/state-a-components-memory) For example, a form component can use state to store the input value, while an image gallery component can use state to store the selected image index.To add state to a component, use one of these Hooks:  * [useState](https://react.dev/reference/react/useState) declares a state variable that you can update directly. * [useReducer](https://react.dev/reference/react/useReducer) declares a state variable with the update logic inside a [reducer function.](https://react.dev/learn/extracting-state-logic-into-a-reducer)   **Reducer ⇒>** Components with many state updates spread across many event handlers can get overwhelming. For these cases, you can consolidate all the state update logic outside your component in a single function, called a *reducer.Extracting State Logic into a Reducer* useReducer ⇒> useReducer is a React Hook that lets you add a [reducer](https://react.dev/learn/extracting-state-logic-into-a-reducer) to your component.const.dispatch function returned by useReducer lets you update the state to a different value and trigger a re-render. You need to pass the action as the only argument to the dispatch function. useReducer is very similar to [useState](https://react.dev/reference/react/useState), but it lets you move the state update logic from event handlers into a single function outside of your component. Use a reducer if you often encounter bugs due to incorrect state updates in some component, and want to introduce more structure to its code. You don’t have to use reducers for everything: feel free to mix and match! You can even useState and useReducer in the same component.Context Hooks ⇒> *Context* lets a component [receive information from distant parents without passing it as props.](https://react.dev/learn/passing-props-to-a-component) For example, your app’s top-level component can pass the current UI theme to all components below, no matter how deep.  * [useContext](https://react.dev/reference/react/useContext) reads and subscribes to a context.   function Button() {  const theme = useContext(ThemeContext);  // … Ref Hooks ⇒> *Refs* let a component [hold some information that isn’t used for rendering,](https://react.dev/learn/referencing-values-with-refs) like a DOM node or a timeout ID. Unlike with state, updating a ref does not re-render your component. Refs are an “escape hatch” from the React paradigm. They are useful when you need to work with non-React systems, such as the built-in browser APIs.  * [useRef](https://react.dev/reference/react/useRef) declares a ref. You can hold any value in it, but most often it’s used to hold a DOM node. * [useImperativeHandle](https://react.dev/reference/react/useImperativeHandle) lets you customize the ref exposed by your component. This is rarely used.   function Form() {  const inputRef = useRef(null);  // ...  **Ref ⇒>** When you want a component to “remember” some information, but you don’t want that information to [trigger new renders](https://react.dev/learn/render-and-commit), you can use a *ref*.You can add a ref to your component by importing the useRef Hook from React:  import { useRef } from 'react'; . Inside your component, call the useRef Hook and pass the initial value that you want to reference as the only argument. For example, here is a ref to the value 0: const ref = useRef(0);  You can point a ref to any value. However, the most common use case for a ref is to access a DOM element. useRef ⇒> useRef is a React Hook that lets you reference a value that’s not needed for rendering.useRef returns a ref object with a single current property initially set to the initial value you provided.Changing a ref does not trigger a re-render , This means refs are perfect for storing information that doesn’t affect the visual output of your component. For example, if you need to store an [interval ID](https://developer.mozilla.org/en-US/docs/Web/API/setInterval) and retrieve it later, you can put it in a ref. To update the value inside the ref, you need to manually change its current propertyOnly Call Hooks at the Top Level ⇒> Don’t call Hooks inside loops, conditions, or nested functions. Instead, always use Hooks at the top level of your React function, before any early returns. By following this rule, you ensure that Hooks are called in the same order each time a component renders. That’s what allows React to correctly preserve the state of Hooks between multiple useState and useEffect calls.Only Call Hooks from React Functions ⇒> Don’t call Hooks from regular JavaScript functions. Instead, you can:  * ✅ Call Hooks from React function components. * ✅ Call Hooks from custom Hooks  Effect Hooks ⇒> *Effects* let a component [connect to and synchronize with external systems.](https://react.dev/learn/synchronizing-with-effects) This includes dealing with network, browser DOM, animations, widgets written using a different UI library, and other non-React code.  * [useEffect](https://react.dev/reference/react/useEffect) connects a component to an external system.   function ChatRoom({ roomId }) {  useEffect(() => {  const connection = createConnection(roomId);  connection.connect();  return () => connection.disconnect();  }, [roomId]);  // ... Performance Hooks ⇒> A common way to optimize re-rendering performance is to skip unnecessary work. For example, you can tell React to reuse a cached calculation or to skip a re-render if the data has not changed since the previous render.To skip calculations and unnecessary re-rendering, use one of these Hooks:  * [useMemo](https://react.dev/reference/react/useMemo) lets you cache the result of an expensive calculation. * [useCallback](https://react.dev/reference/react/useCallback) lets you cache a function definition before passing it down to an optimized component.   **useGlobalEvent ⇒>** A hook that streamlines event handling in your application, ensuring that event listeners are added and removed at the appropriate times, without requiring you to manage them manually.  Simply provide the name of the event you want to attach to the window object, and the hook will take care of the rest. Window ⇒> The Window interface represents a window containing a [DOM](https://developer.mozilla.org/en-US/docs/Glossary/DOM) document; the document property points to the [DOM document](https://developer.mozilla.org/en-US/docs/Web/API/Document) loaded in that window.[Window: resize event](https://developer.mozilla.org/en-US/docs/Web/API/Window/resize_event) ⇒> The resize event fires when the document view (window) has been resized.This event is not cancelable and does not bubble.[useWindowSize](https://usehooks.com/useWindowSize/) ⇒> A really common need is to get the current size of the browser window. This hook returns an object containing the window's width and height. A hook that receives a callback function to execute on the window's resize event.It's built on top of [useGlobalEvent](https://github.com/antonioru/beautiful-react-hooks/blob/master/docs/useGlobalEvent.md).&nbsp; ⇒> Non-breaking SpaceThrottle ⇒> By using \_.throttle, we don’t allow to our function to execute more than once every X milliseconds.The main difference between this and debouncing is that throttle guarantees the execution of the function regularly, at least every X milliseconds. **componentDidMount() ⇒>** componentDidMount() is invoked immediately after a component is mounted (inserted into the tree). Initialization that requires DOM nodes should go here. If you need to load data from a remote endpoint, this is a good place to instantiate the network request. If you define the componentDidMount method, React will call it when your component is added *(mounted)* to the screen. This is a common place to start data fetching, set up subscriptions, or manipulate the DOM nodes.  **States** at their core are [objects](https://www.digitalocean.com/community/tutorials/understanding-objects-in-javascript) that are used to store the property values of components. Every time the state changes, the component re-renders.  **toJson** is a helper function in the unsplash-js library that is used to convert the response into [JSON format](https://www.digitalocean.com/community/tutorials/how-to-work-with-json-in-javascript). React-infinite-scroll-component ⇒> A component to make all your infinite scrolling woes go away with just 4.15 kB! Pull Down to Refresh feature added. An infinite-scroll that actually works and super-simple to integrate!Automerge ⇒> Automerge is a library of data structures for building collaborative applications, this package is the javascript implementation. React-quill a free, open-source text editor based on WYSIWYG.Quill allows users to control which toolbar features to use. The supported options can be divided by the way they are being displayed in the editor — as an inline, block, or embed element.  **Delta ⇒>** Deltas are a simple, yet expressive format that can be used to describe Quill’s contents and changes. The format is a strict subset of JSON, is human readable, and easily parsable by machines. Deltas can describe any Quill document, includes all text and formatting information, without the ambiguity and complexity of HTML. React-router-dom ⇒> The primary functionality of react-router-dom is implementing dynamic routing in web applications. Based on the platform and the requirements of the application, react-router-dom supports component-based routing, which is the ideal solution for routing if the React application is running on the browser.Local-storage ⇒> Local storage is a web storage object for storing data locally on the user's computer. This means the data is stored in the user session and, unlike cookies, does not expire. You can access storage objects using the localStorage() API. **dangerouslySetInnerHTML:** An object of the form { \_\_html: '<p>some html</p>' } with a raw HTML string inside. Overrides the [innerHTML](https://developer.mozilla.org/en-US/docs/Web/API/Element/innerHTML) property of the DOM node and displays the passed HTML inside. This should be used with extreme caution! If the HTML inside isn’t trusted (for example, if it’s based on user data), you risk introducing an [XSS](https://en.wikipedia.org/wiki/Cross-site_scripting) vulnerability.**It's a prop that does exactly what you want. However they name it to convey that it should be used with caution .** Behind the scenes when you use dangerouslySetInnerHTML it lets React know that the HTML inside of that component is not something it cares about. Because React uses a virtual DOM, when it goes to compare the diff against the actual DOM, it can straight up bypass checking the children of that node *because it knows the HTML is coming from another source*. So there's performance gains. **More importantly**, if you simply use innerHTML, React has no way to know the DOM node has been modified. The next time the render function is called, **React will overwrite the content** that was manually injected with what it thinks the correct state of that DOM node should be. componentDidUpdate(prevProps, prevState, snapshot?) ⇒> If you define the componentDidUpdate method, React will call it immediately after your component has been re-rendered with updated props or state. This method is not called for the initial render.You can use it to manipulate the DOM after an update. This is also a common place to do network requests as long as you compare the current props to previous propsReact-google-charts ⇒> A thin, typed, React wrapper for [Google Charts](https://developers.google.com/chart/interactive/docs/reference). **Socket.io** is a popular JavaScript library that allows us to create real-time, bi-directional communication between web browsers and a Node. js server. It is a highly performant and reliable library optimized to process a large volume of data with minimal delay.  **onSnapshot() ⇒>** You can *listen* to a document with the onSnapshot() method. An initial call using the callback you provide creates a document snapshot immediately with the current contents of the single document. Then, each time the contents change, another call updates the document snapshot.  **@arco-design/web-react :** A comprehensive React UI components library based on the [Arco Design](https://arco.design/) system. React-pdf ⇒> Display PDFs in your React app as easily as if they were images.React-icons ⇒> Include popular icons in your React projects easily with react-icons, which utilizes ES6 imports that allows you to include only the icons that your project is using.React-dropdown ⇒> Simple Dropdown component for ReactReact-Select ⇒> The Select control for React.A flexible and beautiful Select Input control for ReactJS with multiselect, autocomplete, async and creatable support.@keyframes ⇒> The @keyframes CSS [at-rule](https://developer.mozilla.org/en-US/docs/Web/CSS/At-rule) controls the intermediate steps in a CSS animation sequence by defining styles for keyframes (or waypoints) along the animation sequence.@keyframes rule specifies the animation code.The animation is created by gradually changing from one set of CSS styles to another.During the animation, you can change the set of CSS styles many times.@media rule is used in media queries to apply different styles for different media types/devices.DOM (Document Object Model) ⇒> The DOM (Document Object Model) is an [API](https://developer.mozilla.org/en-US/docs/Glossary/API) that represents and interacts with any [HTML](https://developer.mozilla.org/en-US/docs/Glossary/HTML) or [XML](https://developer.mozilla.org/en-US/docs/Glossary/XML) document. The DOM is a document model loaded in the [browser](https://developer.mozilla.org/en-US/docs/Glossary/Browser) and representing the document as a node tree, where each node represents part of the document (e.g. an [element](https://developer.mozilla.org/en-US/docs/Glossary/Element), text string, or comment).document.defaultView ⇒> In browsers, document.defaultView returns the [window](https://developer.mozilla.org/en-US/docs/Web/API/Window) object associated with [a document](https://developer.mozilla.org/en-US/docs/Glossary/Browsing_context), or null if none is available. This property is read-only.RegExp ⇒> The RegExp object is used for matching text with a pattern.MathML(Writing mathematics with MathML) ⇒> Mathematical Markup Language — or [MathML](https://developer.mozilla.org/en-US/docs/Glossary/MathML) — is the markup language used to write mathematical formulas in web pages using fractions, scripts, radicals, matrices, integrals, series, etc. Although it was originally designed as an independent XML language, MathML is generally embedded inside [HTML](https://developer.mozilla.org/en-US/docs/Glossary/HTML) documents and can be seen as an extension of HTML.⇒> yield operator is used to pause and resume a [generator function](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/function*).⇒> yield\* operator is used to delegate to another [iterable](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Iteration_protocols#the_iterable_protocol) object, such as a [Generator](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Generator).Document ⇒> The Document interface represents any web page loaded in the browser and serves as an entry point into the web page's content, which is the [DOM tree](https://developer.mozilla.org/en-US/docs/Web/API/Document_object_model/Using_the_Document_Object_Model#what_is_a_dom_tree).The DOM tree includes elements such as [<body>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/body) and [<table>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/table), among [many others](https://developer.mozilla.org/en-US/docs/Web/HTML/Element). It provides functionality globally to the document, like how to obtain the page's URL and create new elements in the document.Manipulating documents([DOM manipulation](https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Client-side_web_APIs/Manipulating_documents#active_learning_basic_dom_manipulation)) Methods - I have mentioned 8 methods here:Document: querySelector() ⇒> The [Document](https://developer.mozilla.org/en-US/docs/Web/API/Document) method querySelector() returns the first [Element](https://developer.mozilla.org/en-US/docs/Web/API/Element) within the document that matches the specified selector, or group of selectors.Document: createElement()⇒> In an [HTML](https://developer.mozilla.org/en-US/docs/Web/HTML) document, the document.createElement() method creates the HTML element specified by *tagName*, or an [HTMLUnknownElement](https://developer.mozilla.org/en-US/docs/Web/API/HTMLUnknownElement) if *tagName* isn't recognized.Document: createTextNode() ⇒> Creates a new [Text](https://developer.mozilla.org/en-US/docs/Web/API/Text) node. This method can be used to escape HTML characters.Node: appendChild() ⇒> The appendChild() method of the [Node](https://developer.mozilla.org/en-US/docs/Web/API/Node) interface adds a node to the end of the list of children of a specified parent node.Node: removeChild() ⇒> The removeChild() method of the [Node](https://developer.mozilla.org/en-US/docs/Web/API/Node) interface removes a child node from the DOM and returns the removed node.Element: remove() ⇒> The Element.remove() method removes the element from the DOM.Element: setAttribute() ⇒> Sets the value of an attribute on the specified element. If the attribute already exists, the value is updated; otherwise a new attribute is added with the specified name and value.Document: querySelectorAll() method ⇒> The [Document](https://developer.mozilla.org/en-US/docs/Web/API/Document) method querySelectorAll() returns a static (not live) [NodeList](https://developer.mozilla.org/en-US/docs/Web/API/NodeList) representing a list of the document's elements that match the specified group of selectors.forceUpdate() ⇒> Forces a component to re-render. Usually, this is not necessary. If your component’s [render](https://react.dev/reference/react/Component#render) method only reads from [this.props](https://react.dev/reference/react/Component#props), [this.state](https://react.dev/reference/react/Component#state), or [this.context,](https://react.dev/reference/react/Component#context) it will re-render automatically when you call [setState](https://react.dev/reference/react/Component#setstate) inside your component or one of its parents. However, if your component’s render method reads directly from an external data source, you have to tell React to update the user interface when that data source changes. That’s what forceUpdate lets you do.Express is a node js web application framework that provides broad features for building web and mobile applications. It is used to build a single page, multipage, and hybrid web application. It's a layer built on the top of the Node js that helps manage servers and routes.A minimalist web framework for [Node.js](http://nodejs.org/). **CORS** in node. js stands for Cross-Origin Resource Sharing. It is a mechanism by which resources are shared across different servers.Cross-Origin Resource Sharing (CORS) is an HTTP-header based mechanism that allows a server to indicate any origins (domain, scheme, or port) other than its own from which a browser should permit loading resources.  **Nodemon** is a command-line tool that helps with the speedy development of Node. js applications. It monitors your project directory and automatically restarts your node application when it detects any changes. This means that you do not have to stop and restart your applications in order for your changes to take effect. Nodemon is like a live-server for your node application. Any changes made in your node application will get reflected as server will restart again.  **useNavigate ⇒>** useNavigation is a hook which gives access to navigation objects. It's useful when you cannot pass the navigation prop into the component directly, or don't want to pass it in case of a deeply nested child. flushSync ⇒> flushSync lets you force React to flush any updates inside the provided callback synchronously. This ensures that the DOM is updated immediately. **e**[**vent.target**](https://developer.mozilla.org/en-US/docs/Web/API/Event/target) gives you the element **that** triggered the event. So, event.target.value retrieves the **value** of that element . In React, events are [SyntheticEvent](https://reactjs.org/docs/events.html), a wrapper around the browser’s native event. It has the same interface as the browser’s native event, including stopPropagation() and preventDefault(), except the events work identically across all browsers.  **Immer ⇒> Immer is a tiny package that allows you to work with immutable states in a more convenient way.**You can use a library like Immer to reduce repetitive code . Immer lets you write concise code as if you were mutating objects, but under the hood it performs immutable updates . useImmer(initialState) is very similar to useState. The function returns a tuple, the first value of the tuple is the current state, the second is the updater function, which accepts an immer producer function or a value as argument. Resetting state with a key ⇒> You’ll often encounter the key attribute when [rendering lists.](https://react.dev/learn/rendering-lists) However, it also serves another purpose. You can reset a component’s state by passing a different key to a component.useContext ⇒> useContext is a React Hook that lets you read and subscribe to [context](https://react.dev/learn/passing-data-deeply-with-context) from your component.The context itself does not hold the information, it only represents the kind of information you can provide or read from components. Passing something via context only works if SomeContext that you use to provide context and SomeContext that you use to read it are *exactly* the same object, as determined by a === comparison. useContext returns the context value for the context you passed. To determine the context value, React searches the component tree and finds the closest context provider above for that particular context. useContext() always looks for the closest provider *above* the component that calls it. It searches upwards and does not consider providers in the component from which you’re calling useContext(). To update context, combine it with [state.](https://react.dev/reference/react/useState) Declare a state variable in the parent component, and pass the current state down as the context value to the provider. **Handling events ⇒>** Here we are passing a function(in this case let’s say handleClick) as a value so that React can add that function as the event handler in case click ever happens on this button **:** **<button onClick={handleClick}>Click me</button>**  **onMouseMove ⇒>** The mousemove event is fired at an element when a pointing device (usually a mouse) is moved while the cursor's hotspot is inside it.The onmousemove event occurs when the pointer moves over an element.  **onMouseLeave ⇒>** The mouseleave event is fired at an [Element](https://developer.mozilla.org/en-US/docs/Web/API/Element) when the cursor of a pointing device (usually a mouse) is moved out of it.onmouseleave event occurs when the mouse pointer leaves an element. The onmouseleave event is often used together with the [onmouseenter](https://www.w3schools.com/jsref/event_onmouseenter.asp) event, which occurs when the mouse pointer enters an element. Theonmouseleave event is similar to the [onmouseout](https://www.w3schools.com/jsref/event_onmouseout.asp) event. The difference is that the onmouseleave event does not bubble (does not propagate up the document hierarchy).  **onMouseOver ⇒>** onmouseover event occurs when the mouse pointer enters an element. The onmouseover event is often used together with the [onmouseout](https://www.w3schools.com/jsref/event_onmouseout.asp) event, which occurs when the mouse pointer leaves the element. The onmouseover event is similar to the [onmouseenter](https://www.w3schools.com/jsref/event_onmouseenter.asp) event. The difference is that the onmouseenter event does not bubble (does not propagate up the document hierarchy).  **Props Revisited :**  **State :** State is a way for React to remember saved values from within a component.This is similar to declaring variables from within a component . We want to use state instead of props when we want a component to maintain some values from within the component. (And "remember" those values even when React re-renders the component).State is mutable.  **useState - inner working - 1 :** When you run code below inside a component const result = **React.useState() console.log(result) , it will return an array  , this array contains an undefined value and a function. If your put something inside usetate something like const result = React.useState("Hello")** then it will return .Whatever we put inside parentheses when calling useState is going to be the default value for the state variable like string “Hello” in our case.Full code explaining inner working of ueStateconsole.log in this code will print and on web-page it will display  **useState - inner working - 2 :** Here **setIsImportant** functions job is to give us the ability to change state .When handleClick function gets called here “**onClick={handleClick}”** then the setIsImportantfunction worked and changed the state from “Yes” to “No”.  **useState - inner working - 3 : In this example we are changing the state when user clicks add or subtract buttons.**  **useState - inner working - 4 :** If you ever need to use the old value od state in order to determine the new value of state then pass a function to the set function of the state like we did here and passed a function to **setCount .** Full Code is below :  **If you ever need the old value of state to help you determine the new value of state, you should pass a callback function to your state setter function instead of using state directly. This callback function will receive the old value of state as its parameter, which you can then use to determine your new value of state.**  **useState - inner working - 5 : const** [isImportant, setIsImportant] = React.useState("Yes") , Whatever value I provide to setIsImportant will be a new version of state. setIsImportant is the function that will make changes for us. We would normally want to set or change the state whenever something specific happens on the page.We took the function(setIsImportant) we were given when we called useState , this function's job is to give us the ability to change state if we want to.  **useState - Inner working - 6 - Counter Example :**  **Complex State - Arrays :**  function App() {  const [thingsArray, setThingsArray] = React.useState(["Thing 1", "Thing 2"])  function addItem() {  setThingsArray(prevState => {  return [...prevState, **`Thing ${prevState.length + 1}`]})} // This(...prevState) is my existing array and Here(Thing ${prevState.length + 1}) we are adding new item to the end of array.**  **Passing state as prop :** The Parent component has a name state, when it renders the Child component it passes the state variable as a prop named name. The Child component uses the prop name to render a cool heading, it does props.name.  **Add Dynamic Style :** Here in styles we are changing the background color of the props that we are receiving from parent component based on some condition  **derived state ⇒>** When a prop is used to initiate a state, that state is called a **derived state**. When you create a state that is initialized by incoming prop and it’s called derived state .By deriving state from props, you split the source of truth for one thing into two. It was originally a single thing (prop), a derived state lets it diverge and potentially get out of sync . Single source of truth means for every particular piece of data, there is one particular place where it comes from. This makes easy tracking of data flow . Updating arrays without mutation ⇒> Create a *new* array which contains the existing items *and* a new item at the end. Use the ... [array spread](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Spread_syntax#spread_in_array_literals) syntax. **setArtists( // Replace the state**  **[ // with a new array**  **...artists, // that contains all the old items**  **{ id: nextId++, name: name } // and one new item at the end**  **]**  **);**  **Functions are objects , which means they can also be used as expressions.function keyword can be used to define a function inside an expression :**  **High order components :** :  **Pure components :**  **Props revisited :**  **Map over array and render components example :** prop-typesis a special component property that can be used to validate the props you have in a component.You can use prop-types to document the intended types of properties passed to components. **Each child in an array should have a unique "key" prop ⇒>** React uses the key prop to understand the component-to-DOM Element relation, which is then used for the [reconciliation process](https://facebook.github.io/react/docs/reconciliation.html). It is therefore very important that the key *always* remains *unique*, otherwise there is a good chance React will mix up the elements and mutate the incorrect one. It is also important that these keys *remain static* throughout all re-renders in order to maintain best performance.  **Custom Component Example -**  Here we are defining a custom component Page and rendering it via ReactDOM.render :  **React Component Syntax and Example of sort -** This is how you define a component , now you can put your div , or other like h1 elements in it :  **Passing function as event handler :** Passing function handleClick as a value so that react can add that function as event handler  **Random value from an array:**  **let randomValue = myArray[Math.floor(Math.random() \* myArray.length)];** map() example - Generate JSX code for each item ⇒> To generate JSX code that renders the array, Array.map() function is applied on itemList constant with a callback function. In the callback function, the string value of the array entry is accessed, and <p></p> enclosing item string value is returned. React JS requires a unique key for every dynamic array JSX element, this allows React virtual DOM to identify changes in the array such as update, delete, etc more efficiently.If the key is not added then you will see a console error with the message as “Each child in an array or iterator should have a unique “key” prop“ . **const renderList = thingsArray.map(thing) =>**  **<p key={thing}>{thing}</p>**  **);**  **Props vs State :** |