

C:\vs code\GitHub\learn\WEB DEV\FRONT-END\5-React\new\_learn\works\my\_app\res.jsx

```
1 // start a react app : --[]
2 ```
3 npm run start
4 ```
5 // install all libraries from the package.json file :
6 ```
7 npm install
8 ```
9 // {for managing course content } : --[]
10
11 // clone the main branch
12 ```
13 git clone https://github.com/Yarob50/Tarmeez-React-Course.git
14 ```
15 // git all branches list :
16 ```
17 git branch -a
18 ```
19
20 // create new react app : (app_name) can not contain capital letters :-----[]
21 ```
22 npx create-react-app app_name;
23 ```
24
25 // app.js : it's represented the root component :
26
27
28 // File Structure :-----[]
29 /*
30 1. **index.html**:
31 - This file is the entry point of your React application.
32 - It contains the HTML structure of your web page.
33 - Usually includes a `

` element with an id where React will render the application.
34
35 2. **index.js**:
36 - This file is the entry point of your React application where you bootstrap your React application.
37 - It typically imports necessary dependencies and renders the root component of your application
38   into the DOM.
39
40 3. **App.js**:
41 - This file defines the root component of your React application.
42 - It's where you structure your application layout and manage the overall state and behavior.
43
44 4. **Component files (e.g., Header.js, Footer.js, etc.)**:
45 - These files contain individual components of your application.
46 - Each component typically represents a reusable UI element or a logical part of your application.
47 - They encapsulate HTML structure, styles, and behavior related to that specific UI element.
48


```

```
49 5. **Stylesheets (e.g., styles.css, App.css)**:
50 - These files contain stylesheets for your components.
51 - They provide CSS rules for styling your components and ensuring a consistent look and feel
52   across the application.
53
54 6. **Other utility files (e.g., utils.js, constants.js)**:
55 - These files contain utility functions, constants, or configurations used throughout your application.
56 - They help keep your code organized and facilitate code reuse.
57
58 7. **Package.json**:
59 - This file contains metadata about your project and its dependencies.
60 - It lists all the dependencies required for your project to run, along with their versions.
61 - It also includes scripts for running various tasks such as starting the development server
62   or building the production bundle.
63
64 8. **Node_modules**:
65 - This directory contains all the dependencies installed for your project.
66 - It's managed by npm (Node Package Manager) and typically not version-controlled.
67
68 9. **Webpack.config.js / Babel.config.js**:
69 - These files contain configurations for bundling and transpiling your React code.
70 - Webpack is a module bundler that bundles your JavaScript files and their dependencies into a single file for
71   the browser.
72 - Babel is a JavaScript compiler that transforms your modern JavaScript code (ES6/ES7) into a backward-compatible
73   version for older browsers.
74
75 10. **gitignore**:
76 - This file specifies intentionally untracked files that Git should ignore.
77 - It usually includes directories like `node_modules` and files like `.DS_Store` that you don't want
78   to include in version control.
79
80 11 ** .public**:
81 - static content
82
83 This structure provides a foundation for organizing a React application, allowing for scalability, maintainability,
84 and reusability of code. It separates concerns by breaking down the
85 application into smaller, manageable components and provides a clear separation of HTML, CSS, and JavaScript logic.
86 */
87
88
89 // App.js Main Structure :-----[]
90 import React, { Component } from 'react';
91 import ReactDOM from 'react-dom/client';
92 import './index.css';
93 import App from './App';
94 import reportWebVitals from './reportWebVitals';
95
96 const root = ReactDOM.createRoot(document.getElementById('root'));
97 root.render(
98 <React.StrictMode>
```

```
99   <App />
100 </React.StrictMode>
101 );
102
103 // If you want to start measuring performance in your app, pass a function
104 // to log results (for example: reportWebVitals(console.log))
105 // or send to an analytics endpoint. Learn more: https://bit.ly/CRA-vitals
106 reportWebVitals();
107 /*
108 Sure, let's break down the code `App.js`:-----[]
109
110 1. Imports:
111 - `import React from 'react';`: Imports the React library, which is required for building React components.
112 - `import ReactDOM from 'react-dom/client';`: Imports the `ReactDOM` module from the `react-dom` package.
113 It's used for rendering React components into the DOM.
114 - `import './index.css';`: Imports a CSS file for styling. This is typically used to apply global
115 styles to the application.
116 - `import App from './App';`: Imports the `App` component from the file `App.js`. This is the main
117 component of the application.
118 - `import reportWebVitals from './reportWebVitals';`: Imports a function called
119 `reportWebVitals` from a file named `reportWebVitals.js`.
120
121 2. Root Element:
122 - `const root = ReactDOM.createRoot(document.getElementById('root'));`: Creates a root element
123 using `ReactDOM.createRoot()`. This method is used to create a root for the new concurrent
124 React mode.
125 - `root.render()`: Renders the `App` component into the root element.
126
127 3. Strict Mode:
128 - `<React.StrictMode>`: Wraps the `App` component with `React.StrictMode`. This is a development
129 mode that helps identify potential problems in your code. It enables additional checks and warnings
130 for potential issues.
131 - `</React.StrictMode>`: Closes the `React.StrictMode` component.
132
133 4. Performance Measurement:
134 - `reportWebVitals()`: Calls the `reportWebVitals` function. This function can be used to measure various
135 performance metrics in the application, such as rendering performance, network performance, etc. You can pass
136 a callback function to log the results or send them to an analytics endpoint.
137
138 */
139 const reportWebVitals = onPerfEntry => {
140   if (onPerfEntry && onPerfEntry instanceof Function) {
141     import('web-vitals').then(({ getCLS, getFID, getFCP, getLCP, getTTFB }) => {
142       getCLS(onPerfEntry);
143       getFID(onPerfEntry);
144       getFCP(onPerfEntry);
145       getLCP(onPerfEntry);
146       getTTFB(onPerfEntry);
147     });
148   }
149 }
```

```
149 };
150
151 export default reportWebVitals;
152
153 // ReportWebVitals.js Explain code : -----[]
154
155 /*
156
157 1. **Function Definition**:
158 - `const reportWebVitals = onPerfEntry => {`: Defines a constant named `reportWebVitals` which is a
159   function that takes a parameter `onPerfEntry`. This parameter is expected to be a callback function
160   that will be invoked with performance data.
161
162 2. **Conditional Check**:
163 - `if (onPerfEntry && onPerfEntry instanceof Function) {`: Checks if `onPerfEntry` is provided and if
164   it's a function. This ensures that the function is callable and has been provided before proceeding.
165
166 3. **Dynamic Import**:
167 - `import('web-vitals').then(...`: Dynamically imports the `web-vitals` module using the `import()`
168   function. This is a dynamic import syntax introduced in ES6. It allows importing modules on demand.
169 - `({ getCLS, getFID, getFCP, getLCP, getTTFB }) => { ... }`: Destructures the imported module to
170   extract specific functions `getCLS`, `getFID`, `getFCP`, `getLCP`, and `getTTFB`.
171
172 4. **Performance Metric Collection**:
173 - `getCLS(onPerfEntry);`, `getFID(onPerfEntry);`, `getFCP(onPerfEntry);`, `getLCP(onPerfEntry);`,
174   `getTTFB(onPerfEntry);`: Calls each of the imported functions (`getCLS`, `getFID`, etc.) with
175   `onPerfEntry` as an argument. These functions are responsible for collecting specific
176   performance metrics like Cumulative Layout Shift (CLS), First Input Delay (FID), etc.
177
178 5. **Export**:
179 - `export default reportWebVitals;`: Exports the `reportWebVitals` function as the default export
180   of this module. This allows other parts of the codebase to import and use this function.
181
182 Overall, this code dynamically imports the `web-vitals` module, retrieves specific
183 performance metric functions from it,
184 and invokes them with the provided callback function (`onPerfEntry`).
185 It provides a generic way to collect and report
186 web performance metrics in a React application.
187
188 The functions `getCLS`, `getFID`, `getFCP`, `getLCP`, and `getTTFB` are part of a web
189 performance API called [Web Vitals](https://web.dev/vitals/), which provides key metrics
190 to help measure the performance and user experience of a website. Each of these functions
191 captures a specific aspect of web performance. Here's an explanation of each:
192
193 ### 1. **`getCLS(onPerfEntry)` - Cumulative Layout Shift (CLS)**
194 - **CLS** measures visual stability and how much the content on the page shifts during loading.
195 - A low CLS score means the elements on the page don't move around unexpectedly, providing
196   a better user experience. High CLS often occurs when images, ads, or fonts load asynchronously
197   and cause layout shifts.
198 - Good CLS score: A score below 0.1 is considered good.
```

```
199
200  **Example use case**: You visit a page, and suddenly an ad appears, causing text to shift down.
201  CLS measures this unexpected shift.
202
203  ### 2. `getFID(onPerfEntry)` - First Input Delay (FID)**
204  - **FID** measures the **time from when a user first interacts** with your page
205  (e.g., clicking a button, tapping on a link) to the time when the browser is actually
206  able to begin processing that interaction.
207  - This metric is important for measuring **interactivity** and responsiveness.
208  - **Good FID score**: A score below 100ms is considered good.
209
210  **Example use case**: You click a button, but the page doesn't respond immediately
211  due to JavaScript tasks or layout calculations. FID measures the delay between your
212  click and the page responding.
213
214  ### 3. `getFCP(onPerfEntry)` - First Contentful Paint (FCP)**
215  - **FCP** measures the time from when the page starts loading to the point when
216  **any part of the page's content** is rendered on the screen. This can include text,
217  images, or other DOM elements.
218  - This metric is crucial for measuring **perceived load speed**, giving users a sense
219  that the page is loading.
220  - **Good FCP score**: A score below 1.8 seconds is considered good.
221
222  **Example use case**: When you load a page, FCP is the moment when the first visible
223  part of the webpage (such as text or an image) appears in the viewport.
224
225  ### 4. `getLCP(onPerfEntry)` - Largest Contentful Paint (LCP)**
226  - **LCP** measures the **render time of the largest visible content element**
227  (such as an image or large text block) within the viewport.
228  - It's a critical metric for measuring how long it takes for the main content
229  to become visible to the user, reflecting **perceived loading performance**.
230  - **Good LCP score**: A score below 2.5 seconds is considered good.
231
232  **Example use case**: On a blog page, the LCP might be an image or large
233  text header. LCP measures when the largest visible content element is fully loaded.
234
235  ### 5. `getTTFB(onPerfEntry)` - Time to First Byte (TTFB)**
236  - **TTFB** measures the time it takes for the browser to receive
237  the **first byte of content** from the server after the user requests the page.
238  - This metric is important for measuring **backend performance** and
239  the responsiveness of the server.
240  - **Good TTFB score**: A score below 200ms is considered good.
241
242  **Example use case**: When you request a page, TTFB measures how long it takes the server
243  to send the first byte of data to the browser after the request is made.
244
245  ---
246
247
248  */
```

```
249
250 // React linking index.html with index.js :-----[]
251 /*
252 When you run a React application using tools like Create React App,
253 a development server is launched that handles linking the JavaScript files with the
254 `index.html` file automatically.
255
256 During development:
257
258 1. Development Server:
259 - When you run `npm start` or `yarn start`, Create React App starts a development server.
260 - This development server serves your React application and provides hot reloading,
261 allowing you to see changes in real-time as you develop.
262 - The development server takes care of linking the JavaScript files (typically named
263 something like `main.js` or `bundle.js`) with the `index.html` file.
264
265 2. Automatic Injection:
266 - As you make changes to your React components and save your files, Create React App
267 automatically rebuilds your application and updates the browser.
268 - The development server injects the updated JavaScript code into the `index.html`
269 file, so you don't need to manually refresh the page to see your changes.
270
271 During production:
272
273 1. Build Process:
274 - When you build your React application for production using `npm run build` or `yarn build`,
275 Create React App generates optimized production-ready files.
276 - This includes a bundled JavaScript file containing your React components and logic.
277
278 2. Injection Mechanism:
279 - Create React App uses a build script to inject the bundled JavaScript file into the `index.html` file.
280 - It replaces a placeholder comment in the `index.html` file with a `
```

```
299 4- import the js module(new js file ) in the target file (App.js)
300 */
301 // example :
302 // content.js :
303 ```
304     export default function MyFirstComponent() {
305         return (
306             <div className="content">
307                 <h1>hello world</h1>
308                 <h2>ayoub majid</h2>
309             </div>
310         );
311     }
312 ```
313 // App.js File :
314     import Content from './content'
315
316 // use a component : -----[]
317 ```
318     <compName></compName>
319 ```
320 //or : self-closing element
321 ```
322     <compName/>
323 ```
324 // use js into xml structure : -----[]
325 ```
326     {jsCode}
327 ```
328 // example 1 :
329 ```
330     export default function content() {
331         const title="hello World";
332         return (
333             <div className="content">
334                 <h1>{title}</h1>
335                 <h2>ayoub majid</h2>
336             </div>
337         );
338     }
339 ```
340 //example 2 :
341 ```
342     export default function content() {
343         return (
344             <div className="content">
345                 <h1>{contentObj.title}</h1>
346                 <h2>{contentObj.userFullName}</h2>
347             </div>
348         );
349     }
350 ```
```

```
349     }
350     let contentObj= {
351         title:"Hello world",
352         userFullName : "Ayoub Majd"
353     };
354     ```
355
356 // add attribute : -----[]
357 // add event :
358 ```
359     <element onEvent={functionName} ></element>
360     ```
361 //example :
362 ```
363     function btnClick(){
364         console.log("You Clicked At me ")
365     }
366     <button onClick={btnClick} >Click Here</button>
367
368     ```
369 // add styling :
370 /*
371     you have to set the styling in camelCase format : (ex : backgroundColor)
372 */
373 ```
374     <element style={{styling}} ></element>
375     ```
376
377 // example :
378 ```
379     const elemStyling={
380         backgroundColor:"green",
381         fontWeight:""
382     }
383     <header className="App-header" style={elemStyling}>
384     ```
385 // add class to an element :
386 ```
387     <element className="cls1 cls2" ></element>
388     ```
389 //add class to an element using a variable :
390 ```
391     <element className={variableName} ></element>
392     ```
393 //add class to an element using variables and string ( Template Literals ) :
394 ```
395     <element className={` className ${variableName}`} ></element>
396
397 -- or :
398     <element className={'className' + variableName} ></element>
```



```
399
400 ```
401 // example :
402 import "./myButton.css";
403
404 export default function MyButton() {
405     const { title, githubLink, class: buttonClass } = ManageByButton.myButtonInfo;
406
407     return (
408         <div className={`content ${buttonClass}`}>
409             <h1>{title}</h1>
410             <a href={githubLink}>my github account</a>
411         </div>
412     );
413 }
414
415 class ManageByButton {
416     static myButtonInfo = {
417         title: "Hello world",
418         githubLink: "https://github.com",
419         class: "contentButton",
420     };
421
422     static getCurrentDate() {
423         return new Date().toString();
424     }
425 }
426
427 const title = "Hello world";
428
429
430 const myButtonInfo = {
431     title: title,
432     githubLink: "https://github.com",
433     class: "contentButton"
434 };
435
436
437
438 // add styling using a css file : -----[]
439 /*
440 1- create new css file :
441 2- add styling
442 3- import css file in the component.js file: import "./FileName.css"
443 */
444 // example :
445 // content.css
446 ```
447 .content {
448
```

```
449     padding: 20px;
450     margin-bottom: 10px 0;
451
452 }
453 .content button {
454     padding: 10px 20px;
455     font-size: 1em;
456     border-radius: 12px;
457     border: none;
458     background-color: aqua;
459 }
460 ```
461 // content.js
462 ```
463 import "./content.css";
464
465 export default function content() {
466     return (
467         <div className="content">
468             <h1>Hello world</h1>
469             <button onClick={btnClick}>Click Here</button>
470         </div>
471     );
472 }
473 ```
474
475 // Scoped Styling (CSS Modules or Inline Styles)
476 /*
477 CSS Modules: When using CSS Modules, the styles are scoped to the component. This means that the
478 class names are locally scoped, and there is no risk of them affecting other parts of the
479 application.
480
481 Inline Styles: Applying styles directly to elements using the style attribute ensures
482 that those styles are specific to that element only.
483 */
484 // example :
485 // MyButton.module.css
486 .button {
487     background-color: blue;
488     color: white;
489 }
490
491 // MyButton.jsx
492 import styles from './MyButton.module.css';
493
494 export default function MyButton() {
495     return <button className={styles.button}>Click Me</button>;
496 }
497
498
```

```
499 // project structure :
500 /*
501 /your-app-name
502 |
503 |— public
504 |   |— index.html
505 |   |— manifest.json
506 |   |— robots.txt
507 |   |— (any other static assets like icons, images, etc.)
508 |
509 |— src
510 |   |— assets
511 |       |— images
512 |       |— styles
513 |       |— (any other static assets like fonts, etc.)
514 |
515 |   |— components
516 |       |— (ComponentName)
517 |           |— (ComponentName).jsx
518 |           |— (ComponentName).css
519 |           |— (ComponentName).test.js
520 |       |— (more components...)
521 |
522 |   |— hooks
523 |       |— useCustomHook.js
524 |
525 |   |— pages
526 |       |— Home
527 |           |— Home.jsx
528 |           |— Home.css
529 |           |— Home.test.js
530 |       |— (more pages...)
531 |
532 |   |— services
533 |       |— api.js
534 |
535 |   |— utils
536 |       |— helpers.js
537 |
538 |   |— App.js
539 |   |— index.js
540 |   |— index.css
541 |
542 |— .gitignore
543 |— package.json
544 |— README.md
545 |— (other config files like .eslintrc, .prettierrc, etc.)
546
547
548 */
```

```
549 // props : --[]
550 /*
551 parameters that you pass when you create or call a component
552 */
553 // in App.js :
554 ```
555   <ComponentName parameter1="value" parameter2="value" parameterN="value" />
556 ```
557 // in ComponentName.js
558 ```
559   export default function( {parameter1,parameter2,parameterN} ){
560     return(
561       // code
562
563     );
564   }
565 ```
566
567 // example :
568 // App.js :
569
570 import "../App.css";
571 import Content from "../content";
572 function App() {
573   return (
574     <div className="App">
575       <Content name="ayoub" email="ayoub@gmail.com" phone="05332"></Content>
576       <Content email="nasim@gmail.com" phone="3322"></Content>
577       <Content name="majid" email="majid@gmail.com" phone="2544"></Content>
578     </div>
579   );
580 }
581 export default App;
582
583 //Content.js :
584
585 import '../content.css'
586 export default function ( {name,email,phone} ) {
587   return (
588     <div className='content'>
589       <h1>{name}</h1>
590       <h3>{email}</h3>
591       <h4>{phone}</h4>
592     </div>
593   );
594 }
595
596
597 // set default props Values :
598 ComponentName.defaultProps = {
```

```
599     prop1: "Default prop1",
600     prop2: "default prop2",
601   };
602
603   // or:
604   export default function ComponentName({prop1="default prop1",prop2="default prop2"}){
605
606
607   }
608
609   // example :
610   import "./Article .css";
611   export default function Article({ name, email, birthDate }) {
612     console.log(name, email, birthDate);
613
614     return (
615       <article className="articleComponentClass">
616         <h2>{name}</h2>
617         <h2>{email}</h2>
618         <h2>{birthDate}</h2>
619       </article>
620     );
621   }
622   Article.defaultProps = {
623     name: "Default Name",
624     email: "default@example.com",
625     birthDate: "01-01-1970",
626   };
627
628   // add xml content :
629   // in App.js :
630   ```
631   <ComponentName parameter1="value" parameter2="value" parameterN="value" >
632     // xml content
633   </ComponentName>
634   ```
635   // in ComponentName.js : the children key contain value the xml content passed to the componentName
636   ```
637   export default function({parameter1,parameter2,parameterN,children}){
638     return(
639       // code
640       {children}
641
642     );
643   }
644   ```
645
646   // example :
647   // App.js :
```

```

649 import './App.css';
650 import Content from './content';
651 function App() {
652   return (
653     <div className="App">
654       <Content name="ayoub" email="ayoub@gmail.com" phone="05332" />
655
656       <Content email="nasim@gmail.com" phone="3322" >
657         <h1 style={{ backgroundColor: "black", color: "white", padding: "10px" }}>nice to meet you</h1>
658       </Content>
659       <Content name="majid" email="majid@gmail.com" phone="2544"></Content>
660     </div>
661   );
662 }
663 export default App;
664
665 // Content.js :
666
667 import './content.css';
668 import { Component } from 'react';
669 export default function ({ name = "no name", email = "no email", phone = "no phone", children }) {
670   return (
671     <div className="content">
672       <h1>{name}</h1>
673       <h3>{email}</h3>
674       <h4>{phone}</h4>
675       {children}
676     </div>
677   );
678 }
679
680
681 // convert from string to jsx :
682 ```
683 export default function ComponentName(){
684   return (
685     <div dangerouslySetInnerHTML={{ __html:content }} />
686   );
687 }
688 ```
689 /*
690 dangerouslySetInnerHTML: This is used to render the HTML content inside the children. Be careful with this
691 approach to avoid injecting any untrusted content, as it can lead to security issues.
692 */
693 // example :
694 import React from "react";
695 import Button from './Button';
696
697 export default function Tags() {
698

```

```
699
700  const tagText ="javaScript"
701  const children= "<i class='fab fa-js'></i><img src='path/to/javascript.png' alt='JavaScript' />",
702
703
704  return (
705    <div className="tags">
706      <Button text={tagText}>
707        <div dangerouslySetInnerHTML={{ __html:children }} />
708      </Button>
709
710
711
712
713    </div>
714  );
715  }
716
717  // conditional Rendering :--[]
718
719  // example using ternary operator ? :
720  return (
721    <div className="App">
722      {showChildren ? (
723        <Content email="nasim@gmail.com" phone="3322">
724          <h1 style={{ backgroundColor: "black", color: "white", padding: "10px" }}>nice to meet you</h1>
725        </Content>
726      ) : null}
727    </div>
728  );
729
730  // example using if statement :
731  import "./App.css";
732  import Content from "./content";
733  function App() {
734    let showChildren = true;
735
736    return (
737      <div className="App">
738        <Content name="ayoub" email="ayoub@gmail.com" phone="05332" />
739        <Content name="majid" email="majid@gmail.com" phone="2544" />
740        <LoadChildren childrenStatus={showChildren} />
741      </div>
742    );
743  }
744  function LoadChildren({ childrenStatus = true }) {
745    console.log(Boolean(false));
746    if (childrenStatus) {
747      return (
748        <Content email="nasim@gmail.com" phone="3322">
```

```

749     <h1 style={{ backgroundColor: "black", color: "white", padding: "10px" }}>nice to meet you</h1>
750   </Content>
751   );
752   } else {
753     return null;
754   }
755 }
756 export default App;
757
758 // stat :
759 /*
760 state in React allows components to manage and maintain their internal data, enabling
761 them to be dynamic and interactive. Understanding how to effectively use state is
762 crucial for building robust and scalable React applications.
763 */
764
765 // without using stat :
766
767 import "./MyButton.css";
768
769 export default function MyButton() {
770   let name = "ayoub";
771   return (
772     <div className="MyButtonComponentClass">
773       <button
774         onClick={() => {
775           name = parseInt(Math.random() * 10) + "-amina" ;
776           console.log(name);
777         }}
778       >
779         My button
780       </button>
781       <h1>{name}</h1>
782     </div>
783   );
784 }
785
786 /*
787 The UI did not change when you modified the `name` variable because React does not
788 automatically re-render the component when you directly modify local variables. Here's a
789 detailed explanation of why this happens and how you can fix it:
790
791 ### Why the UI Did Not Update
792
793 1. Local Variables and React Rendering:
794 - React's rendering cycle is based on its state and props. Local variables (like `name` in your example)
795   are not part of React's state management system. React does not track or react to changes in these variables.
796
797 2. State Management:
798 - To trigger a re-render in React, you need to use state. Local variables are not reactive, so changing

```



```
799   them will not cause the component to re-render. React only re-renders components when the state or props
800   change.
801
802   3. Direct Variable Mutation:
803   - The `name` variable is changed on a button click, but since it's not part of React's state, React
804   is unaware of this change. The component will only re-render if there is a state change.
805
806   ### How to Fix It
807
808   To ensure that changes to `name` trigger a re-render, you should use React's `useState` hook. The `useState`
809   hook allows you to manage state in functional components, and updating the state will automatically trigger
810   a re-render.
811
812   */
813
814
815   // useStat hook :
816
817   import { useState } from "react";
818   export default function Btn({ title }) {
819     if (!title) title = "user title";
820     const [name, setName] = useState(title);
821
822     function changeName() {
823       if (name === title) setName("Ayoub");
824       else setName("Majid");
825     }
826     return (
827       <div className="btnContainer">
828         <button onClick={changeName}>Click Me</button>
829         <h1>{name}</h1>
830       </div>
831     );
832   }
833
834   /*
835   In this code, the `useState` hook plays a crucial role in managing state
836   within the functional component
837   `Btn`. Let's break down its role and how it triggers re-renders
838   when the state of `title` changes:
839   */
840
841   /** 1. **Initializing State**:
842   ```javascript
843   const [name, setName] = useState(title);
844   ```
845   /*
846   The `useState` hook is used to declare a state variable `name` and its corresponding
847   setter function `setName`. The initial value of the `name` state variable is set to
848   the value of the `title` prop passed to the component.
```

```
849 */
850
851 /** 2. **Rendering Initial UI**:
852 ```javascript
853 <h1>{name}</h1>
854 ```
855 /*
856     The `

# ` element in the JSX renders the current value of the `name` state variable. Initially, 857 it displays the value of the `title` prop passed to the component. 858 */ 859 860 /** 3. **Updating State on Button Click**: 861 ```javascript 862 function changeName() { 863 if (name === title) { 864 setName("Ayoub"); 865 } else { 866 setName("Majid"); 867 } 868 } 869 ``` 870 /* 871 The `changeName` function is called when the button is clicked. It checks if 872 the current value of the `name` state variable is equal to the `title` prop. 873 If they are equal, it updates the `name` state variable to "Ayoub", otherwise, 874 it updates it to "Majid". 875 */ 876 877 /** 4. **Re-rendering on State Change**: 878 /* 879 When the `setName` function is called to update the state variable `name`, 880 React re-renders the component. This is because React detects the change 881 in state and automatically triggers a re-render of the component to reflect 882 the updated state. Consequently, the `` element displaying the value of 883 the `name` state variable is re-evaluated and updated with the new value. 884 */ 885 886 /** 5. **Re-rendered UI**: 887 /* 888 After the state is updated, React re-renders the component with the new value of the `name` 889 state variable. As a result, the UI is updated to display the new name ("Ayoub" or "Majid") 890 depending on the current state. 891 */ 892 /* 893 In summary, the `useState` hook manages the state of the `name` variable in the `Btn` component. 894 When the state changes (due to button click), React automatically triggers a re-render of the 895 component to reflect the updated state, resulting in the UI being updated accordingly. 896 This mechanism enables React to efficiently handle state changes and keep the UI in sync 897 with the underlying data. 898 */


```

```
899
900 // how does react detect the change of state : -- []
901 /*
902     React detects changes in state using a mechanism called reconciliation.
903     When a component's state changes, React compares the previous state with
904     the new state. It then determines which parts of the component's UI need
905     to be updated to reflect the changes in state.
906 */
907
908 /** 1. **Virtual DOM**:
909  */
910     React maintains a virtual representation of the DOM (Document Object
911     Model), known as the Virtual DOM. This virtual representation mirrors
912     the actual DOM but is lightweight and exists entirely in memory.
913 */
914
915 /** 2. **Rendering**:
916  */
917     When a component's state changes, React re-renders the component and
918     updates the Virtual DOM accordingly. React compares the new Virtual DOM
919     with the previous Virtual DOM to identify the differences.
920 */
921
922 /** 3. **Differential Algorithm**:
923  */
924     React employs a highly optimized algorithm called the Reconciliation
925     Algorithm to perform a "diffing" process between the new Virtual DOM
926     and the previous Virtual DOM. This process involves efficiently
927     identifying the minimal set of changes needed to update the actual DOM.
928 */
929
930 /** 4. **Identifying Changes**:
931  */
932     During the diffing process, React identifies which elements in the
933     Virtual DOM have changed between the previous and new states. It compares
934     elements based on their type, attributes, and content.
935 */
936
937 /** 5. **Batching Updates**:
938  */
939     React batches multiple state updates into a single re-render operation
940     for performance optimization. This means that if multiple state changes
941     occur within the same event handler or lifecycle method, React combines
942     them into a single update, reducing unnecessary re-renders.
943 */
944
945 /** 6. **Updating the DOM**:
946  */
947     Once React identifies the changes needed to update the actual DOM, it
948     applies these changes in a batched manner to minimize the number of DOM
```

```
949     manipulations. This process ensures that the UI is efficiently updated
950     to reflect the changes in state.
951     */
952
953     /*
954     In summary, React detects changes in state by comparing the Virtual DOM
955     before and after a component's re-rendering. It uses an efficient
956     reconciliation algorithm to identify the minimal set of changes required
957     to update the actual DOM, ensuring optimal performance and UI responsiveness.
958     */
959
960
961 // example with input :
962 ```
963 import "./Inp.css";
964 import { useState } from "react";
965 export default function Inp() {
966     let [content, setContent] = useState("");
967     function changeName(e) {
968         setContent(e.target.value);
969     }
970     return (
971         <div className="inputContainer">
972             <input placeholder="enter your name" onChange={changeName} />
973             <h2>{content}</h2>
974         </div>
975     );
976 }
977 ```
978
979 // From submission : --[]
980 // Method 1 :
981 ```
982 import "./Frm.css";
983 import { useState } from "react";
984 export default function Frm() {
985     let [name, setName] = useState("");
986     let [age, setAge] = useState(0);
987     age = Number(age);
988     return (
989         <form
990             onSubmit={e => {
991                 e.preventDefault();
992                 console.log("name :", name);
993                 console.log("age :", age);
994             }}
995         >
996         <label for="name">name:</label>
997         <input
998             id="name"
```

```

999         onChange={(event) => {
1000             setName(event.target.value);
1001         }}
1002     />
1003     <label for="age">age:</label>
1004     <input
1005         id="age"
1006         type="number"
1007         onChange={(event) => {
1008             setAge(event.target.value);
1009         }}
1010     />
1011
1012     <button>submit</button>
1013 </form>
1014 );
1015 }
1016
1017 ```
1018 // Method 2 :
1019 ```
1020 import "./Frm.css";
1021 import { useState } from "react";
1022 export default function Frm() {
1023     let [formInfo, setFormInfo] = useState({ name: "", age: 0 });
1024
1025     function setValue(event) {
1026         setFormInfo({ ...formInfo, [event.target.id]: event.target.value });
1027     }
1028     return (
1029         <form
1030             onSubmit={(event) => {
1031                 event.preventDefault();
1032                 console.log(formInfo);
1033             }}
1034         >
1035             <label >name:</label>
1036             <input id="name" onChange={setValue} />
1037             <label >age:</label>
1038             <input id="age" type="number" onChange={setValue} />
1039
1040             <button>submit</button>
1041         </form>
1042     );
1043 }
1044 ```
1045 // advance Example Form Stat:
1046 ```
1047 import "./Frm.css";
1048 import { useState } from "react";

```

```
1049
1050 export default function Frm() {
1051   const [formInfo, setFormInfo] = useState({ name: "", age: 0, generalInfo: "", isStudent: false, userCountry: "MR", gender: ""
});
1052
1053   function handleChange(event) {
1054     const { id, type, value, checked } = event.target;
1055     setFormInfo((prevState) => ({
1056       ...prevState,
1057       [id]: type === "checkbox" ? checked : value,
1058     }));
1059   }
1060
1061   return (
1062     <form
1063       onSubmit={(event) => {
1064         event.preventDefault();
1065         console.log(formInfo);
1066       }}
1067     >
1068       <label htmlFor="name">Name:</label>
1069       <input type="text" id="name" onChange={handleChange} />
1070
1071       <label htmlFor="age">Age:</label>
1072       <input id="age" type="number" onChange={handleChange} />
1073
1074       <label htmlFor="generalInfo">General Info:</label>
1075       <textarea id="generalInfo" onChange={handleChange} />
1076
1077       <div className="checkedBoxes">
1078         <input type="checkbox" id="isStudent" onChange={handleChange} />
1079         <label htmlFor="isStudent">Is Student</label>
1080       </div>
1081
1082       <select id="userCountry" onChange={handleChange}>
1083         <option>MR</option>
1084         <option>KSA</option>
1085         <option>UAI</option>
1086         <option>US</option>
1087       </select>
1088
1089       <label>Gender:</label>
1090
1091       <div>
1092         <label>Male</label>
1093         <input type="radio" id="gender" name="gender" value="Male" onChange={handleChange} checked=
{formInfo.gender === "Male"} />
1094       </div>
1095
1096       <div>
```

```

1097         <label>Female</label>
1098         <input type="radio" name="Female" id="gender" value="Female" onChange={handleChange} checked=
{formInfo.gender === "Female"} />
1099     </div>
1100
1101     <button type="submit">Submit</button>
1102 </form>
1103 );
1104 }
1105
1106 ```
1107 // the complete setForm function :
1108 function setFormInput(event) {
1109     let { id, value, checked, type, name } = event.target;
1110
1111     value = id === "age" ? parseInt(value, 10) : value;
1112
1113     setFormInputs({ ...formInputs, [type === "radio" ? name : id]: type === "checkbox" ? checked : value });
1114 }
1115
1116 // Array Stat Example :
1117 import "./App.css";
1118 import { useState } from "react";
1119
1120 function App() {
1121     // const devices=["Iphone","Mac","Samsung","Windows"];
1122
1123     let [devices, setDevices] = useState([]);
1124     let [deviceInput, setDeviceInput] = useState("");
1125
1126     const devicesList = devices.map((device, index) => {
1127         return (
1128             <div key={index} style={{ display: "flex", gap: "20px", width: "60%", minWidth: "250px", alignItems: "center", border:
"1px solid black", justifyContent: "space-between", padding: "7px 10px", borderRadius: "10px" }}>
1129                 <li>{device}</li>
1130
1131                 <div style={{ display: "flex", gap: "10px" }}>
1132                     <button
1133                         onClick={() => {
1134                             deleteDevice(index);
1135                         }}
1136                     >
1137                         Delete
1138                     </button>
1139
1140                     <button
1141                         onClick={() => {
1142                             updateDevice(index);
1143                         }}
1144                     >

```

```

1145         update
1146     </button>
1147 </div>
1148 </div>
1149 );
1150 });
1151
1152 function deleteDevice(deviceIndex) {
1153     let newDeviceList = [...devices];
1154     newDeviceList.splice(deviceIndex, 1);
1155     setDevices(newDeviceList);
1156 }
1157 function updateDevice(deviceIndex) {
1158     let newDevices = [...devices];
1159
1160     newDevices[deviceIndex] = prompt("enter the new Name : ", newDevices[deviceIndex]);
1161
1162     if (newDevices[deviceIndex]) setDevices(newDevices);
1163 }
1164 function changeStat(event) {
1165     setDeviceInput(event.target.value);
1166 }
1167
1168 function addDevice() {
1169     setDevices((prevState) => [...prevState, deviceInput]);
1170     setDeviceInput("");
1171 }
1172
1173 return (
1174     <div className="App">
1175         <div className="addDeviceContainer" style={{ margin: "30px" }}>
1176             <input placeholder="add new device" style={{ marginRight: "6px" }} onChange={changeStat} value={deviceInput} />
1177             <button onClick={addDevice}>Add</button>
1178         </div>
1179
1180         <ul style={{ marginTop: "20px", display: "flex", flexDirection: "column", gap: "10px", alignItems: "center" }}>
{devicesList}</ul>
1181     </div>
1182 );
1183 }
1184
1185 // oop version : -----[]
1186 import style from "../Test.module.css";
1187
1188 import { useState } from "react";
1189 export default function Test() {
1190     const [newDeviceFormInfo, setNewDeviceFormInfo] = useState({ deviceName: "", isDisabled: false });
1191
1192     const [devices, setDevices] = useState([]);
1193

```



```
1194 class utile {
1195   static handelChangeFormInputs(event, setStatForm) {
1196     let { id, value, checked, type, name } = event.target;
1197
1198     value = id === "age" ? parseInt(value, 10) : value;
1199
1200     setStatForm((formInputs) => {
1201       return { ...formInputs, [type === "radio" ? name : id]: type === "checkbox" ? checked : value };
1202     });
1203   }
1204 }
1205 class clsHandelAddNewDevice {
1206   static ChangeFormInputs(event) {
1207     utile.handelChangeFormInputs(event, setNewDeviceFormInfo);
1208   }
1209   static #changeSubmitButtonStat(newStat) {
1210     setNewDeviceFormInfo((prevFormInfo) => {
1211       return { ...prevFormInfo, isDisabled: newStat };
1212     });
1213   }
1214   static #clearAddFormStat() {
1215     setNewDeviceFormInfo((prevFormInfo) => {
1216       return { deviceName: "", isDisabled: false };
1217     });
1218   }
1219   static handelFormSubmit(event) {
1220     event.preventDefault();
1221     // true => disabled = true
1222     clsHandelAddNewDevice.#changeSubmitButtonStat(true);
1223
1224     clsHandelCurdDevices.addDevice(newDeviceFormInfo["deviceName"]);
1225
1226     clsHandelAddNewDevice.#clearAddFormStat();
1227   }
1228 }
1229 class clsHandelCurdDevices {
1230   static addDevice(deviceName) {
1231     setDevices((prevDevicesInfo) => [...prevDevicesInfo, deviceName]);
1232   }
1233   static editDevice(deviceIndex) {
1234     let targetDeviceName = devices[deviceIndex];
1235
1236     let newName = prompt(`Enter the new name [index ${deviceIndex}] : `, targetDeviceName);
1237
1238     if (newName && newName !== targetDeviceName) {
1239       let tempDevices = [...devices];
1240       tempDevices[deviceIndex] = newName;
1241
1242       setDevices(tempDevices);
1243     }
1244   }
1245 }
```

```

1244     }
1245     static deleteDevice(deviceIndex) {
1246         let newDeviceList = [...devices];
1247         newDeviceList.splice(deviceIndex, 1);
1248         setDevices(newDeviceList);
1249     }
1250 }
1251
1252 return (
1253     <
1254         <div className="header">
1255             <h1>Devices</h1>
1256             <form id="addDeviceForm" onSubmit={clsHandelAddNewDevice.handelFormSubmit}>
1257                 <input type="text" id="deviceName" placeholder="Enter the name of the device" value=
1258 {newDeviceFormInfo["deviceName"]} onChange={clsHandelAddNewDevice.ChangeFormInputs} required />
1259                 <button disabled={newDeviceFormInfo["isDisabled"]} >Add device</button>
1260             </form>
1261         </div>
1262         <div className="devicesContainer">
1263             {devices.map((deviceName, index) => {
1264                 return (
1265                     <div key={index} className="deviceItem">
1266                         <h1 className="deviceName">
1267                             {" "}
1268                             {index}-{deviceName}
1269                         </h1>
1270                         <div className="controlSection">
1271                             <button onClick={() => clsHandelCurdDevices.editDevice(index)}>Edit Device</button>
1272                             <button onClick={() => clsHandelCurdDevices.deleteDevice(index)}>delete Device</button>
1273                         </div>
1274                     </div>
1275                 );
1276             })}
1277         </div>
1278     </>
1279 );
1280
1281
1282 // updating states several Times :--[]
1283 function App() {
1284     let [count, setCount] = useState(0);
1285
1286     function increaseCounter() {
1287         setCount(count + 1);
1288         setCount(count+ 1);
1289     }
1290
1291
1292     return (

```

```
1293     <div className="App">
1294       <h1>The count is : <span onClick={increaseCounter}> {count}</span> </h1>
1295     </div>
1296   );
1297 }
1298 /*
1299   The reason the counter doesn't increment by 2 each time you click is because
1300   the `setCount` function in React's `useState` hook doesn't immediately update
1301   the state. Instead, it schedules an update, and React
1302   may batch multiple `setState` calls together for performance reasons.
1303
1304   In your `increaseCounter` function, you're calling `setCount`
1305   twice with the same value of `count`. Both calls essentially set the same
1306   value. React sees these two calls and batches them together
1307   into a single update, resulting in only one increment by 1.
1308
1309   If you want to increase the count by 2 each time you click,
1310   you should use the functional update form of `setCount`, which
1311   takes the current state as an argument and returns the
1312   new state. This ensures that the state updates are applied
1313   one after the other:
1314   */
1315   ```javascript
1316   function increaseCounter() {
1317     setCount(prevCount => prevCount + 1);
1318     setCount(prevCount => prevCount + 1);
1319   }
1320   ```
1321   /*
1322     With this change, the count will indeed increase by 2
1323     each time you click. Each call to `setCount` is now using
1324     the previous state to calculate the new state, so you're effectively
1325     incrementing by 1 twice.
1326   */
1327   /*
1328     In JavaScript, code execution generally follows a single-threaded event loop model.
1329     This means that JavaScript code is executed in a sequence, and only one operation
1330     can be processed at a time. However, React's reconciliation process and state updates
1331     are asynchronous operations, meaning they don't happen immediately when you call
1332     `setState` or `useState` setter functions.
1333
1334     When you call `setState` or `useState` setters in React, React schedules the state
1335     updates for processing. React then decides when to apply these updates based on
1336     its internal mechanism, which aims to optimize performance by batching updates
1337     and minimizing unnecessary re-renders.
1338
1339     While you can't precisely determine whether two state updates will be batched
1340     together or not in a given scenario, you can rely on React's behavior that it
1341     will batch updates when possible to improve performance.
1342
```

```
1343 Here's a simplified explanation of how React typically handles state updates:
1344
1345 1. When you call `setState` or `useState` setter functions, React records the
1346 state update requests.
1347 2. React batches multiple state updates that occur within the same event loop iteration.
1348 3. Before the next repaint, React reconciles the state updates and performs
1349 a single re-render of the component.
1350
1351 In your specific case, calling `setCount` twice within the same event loop iteration
1352 is likely to result in React batching these updates together,
1353 leading to a single re-render of the component with the combined state update.
1354 However, React's exact behavior may vary depending on factors such as the React version,
1355 the environment (development vs. production), and the complexity of your component tree.
1356
1357 While you cannot directly observe the batching behavior of React's state updates
1358 , you can rely on React's efficient handling of state updates to optimize performance
1359 in your application.
1360 */
1361
1362 // Prop Drilling : is pass props From a parents to a hierarchal children List
1363 // useContext :
1364 /*
1365   useContext is a React Hook that allows functional components to consume values from the Context API.
1366   Context provides a way to pass data through the component tree without having to pass props down
1367   manually at every level.
1368 */
1369 // How useContext Works:
1370 /*
1371   Create a Context: First, you need to create a context using the React.createContext()
1372   function. This creates a new context object.
1373
1374   Provide the Context: You then provide the context to the component tree using a Context.Provider
1375   component. This component wraps the part of the tree where you want to make the context available.
1376
1377   Consume the Context: Finally, you consume the context value in any descendant component using
1378   the useContext hook. This hook takes the context object as an argument and returns
1379   its current value.
1380 */
1381 // steps :
1382 /*
1383   1- define the provider :
1384   2- define the consumer :
1385   3- value to pass :
1386 */
1387
1388 // Main Features of useContext:
1389 /*
1390   Simplicity: useContext provides a simple way to consume context values in
1391   functional components without the need for render props or higher-order components.
1392
```

```
1393 AVOIDS Prop Drilling: It helps in avoiding prop drilling by allowing components
1394 to access context values directly, no matter how deeply nested they are in the component tree.
1395
1396 DYNAMIC Context Updates: Components consuming context with useContext will
1397 re-render whenever the context value changes.
1398
1399 PERFORMANCE Optimization: React optimizes the context value retrieval with useContext,
1400 ensuring that components only re-render when necessary based on changes to the context value.
1401
1402 MULTIPLE Contexts: You can consume multiple contexts within a single component by
1403 calling useContext multiple times with different context objects.
1404
1405 STATIC Type Checking: useContext can be easily used with static type checking libraries
1406 like TypeScript for type-safe context consumption.
1407
1408 In summary, useContext is a powerful tool in React for managing state and sharing data
1409 across components in a more concise and efficient way compared to traditional prop drilling methods.
1410 */
1411 // Example 1: --[]
1412
1413 // ----- App.js ----- :
1414 import './App.css';
1415 import Form from './LoanForm/LoanForm';
1416
1417
1418
1419 export default function App() {
1420
1421
1422
1423
1424     return (
1425         <div className="App">
1426             <Form />
1427
1428         </div>
1429     );
1430 }
1431
1432 // ----- LoanFormInputsContext.js ----- :
1433 import { createContext } from 'react';
1434 export let LoanInputsContext = createContext({
1435     type: "text",
1436     value: "",
1437     id: "", handelChange: null,
1438     placeholder: "",
1439     IsRequired: true }
1440 );
1441
1442
```

```
1443 // ----- LoanForm.js ----- :
1444 import "./LoanForm.css";
1445 import { useState } from "react";
1446 import Alter from "../Alter/Alter";
1447 import Input from "../Input/Input";
1448 import { LoanInputsContext } from "../contexts/LoanFormInputsContext";
1449
1450 export default function Form() {
1451   let initialFormInfo = {
1452     name: "",
1453     phone: "",
1454     age: "",
1455     salary: "Less Than 500$",
1456     isEmployee: false,
1457     isSubmitted: false,
1458   };
1459   let initialErrors = {
1460     isValidPhone: true,
1461     isValidAge: true,
1462   };
1463
1464   let [formInfo, setFormInfo] = useState(initialFormInfo);
1465   let [errors, setErrors] = useState(initialErrors);
1466   document.addEventListener("click", (event) => {
1467     if (formInfo.isSubmitted && !event.target.classList.contains("alterDivText")) {
1468       setFormInfo((prevState) => ({
1469         ...prevState,
1470         isSubmitted: false,
1471       }));
1472     }
1473   });
1474
1475   function changeFormInfo(event) {
1476     const { id, type, value, checked } = event.target;
1477     setFormInfo((prevState) => ({
1478       ...prevState,
1479       [id]: type === "checkbox" ? checked : value,
1480     }));
1481   }
1482   function handelFormSubmission(event) {
1483     event.preventDefault();
1484     setFormInfo((prevState) => ({
1485       ...prevState,
1486       isSubmitted: true,
1487     }));
1488
1489     const isValidPhone = formInfo.phone.length >= 10 && formInfo.phone.length <= 12;
1490     const isValidAge = formInfo.age >= 18 && formInfo.age <= 100;
1491
1492     setErrors({
```

```

1493         isValidPhone,
1494         isValidAge,
1495     });
1496 }
1497
1498 function GeneraleAlterMsg() {
1499     return formInfo.isSubmitted && <>{errors.isValidAge && errors.isValidPhone ?
1500         <Alter msg="The Form Has Been submitted Successfully" /> : !errors.isValidPhone ?
1501         <Alter msg="Phone Number Format is incorrect" msgColor="red" />
1502         : <Alter msg="Age is Not allowed" msgColor="red" />}</>;
1503 }
1504
1505 return (
1506     <>
1507         <form onSubmit={handelFormSubmission}>
1508             <div className="header">
1509                 <h2>Requesting a Loan</h2>
1510                 <hr />
1511             </div>
1512
1513             <div className="nameContainer inputContainer">
1514                 <LoanInputsContext.Provider value={{ value: formInfo.name, id: "name",
1515                 handelChange: changeFormInfo, placeholder: "Enter Your Name" }}>
1516                     <Input />
1517                 </LoanInputsContext.Provider>
1518             </div>
1519             <div className="phoneContainer inputContainer">
1520                 <LoanInputsContext.Provider value={{ value: formInfo.phone, id: "phone",
1521                 handelChange: changeFormInfo, placeholder: "Enter Your Phone " }}>
1522                     <Input />
1523                 </LoanInputsContext.Provider>
1524             </div>
1525
1526             <div className="ageContainer inputContainer">
1527                 <LoanInputsContext.Provider value={{ type: "number", value: formInfo.age, id: "age",
1528                 handelChange: changeFormInfo, placeholder: "Enter Your Age " }}>
1529                     <Input />
1530                 </LoanInputsContext.Provider>
1531             </div>
1532
1533             <input type="submit" />
1534         </form>
1535
1536         {GeneraleAlterMsg()}
1537     </>
1538 );
1539 }
1540
1541 // ----- Input.js ----- :
1542 import "./Input.css";

```

```
1543
1544 import { useContext } from "react";
1545
1546 import { LoanInputsContext } from "../contexts/LoanFormInputsContext";
1547 import { UserContext } from "../contexts/UserContext";
1548
1549 export default function Input() {
1550     const inputContext = useContext(LoanInputsContext);
1551     const userContextInfo = useContext(UserContext);
1552
1553     return (
1554         <div className="inputContent">
1555             <h1>The User {userContextInfo.username}</h1>
1556             <h2>This is the header of the Component</h2>
1557             <input type={inputContext.type} placeholder={inputContext.placeholder}
1558                 id={inputContext.id} value={inputContext.value} onChange={inputContext.handelChange}
1559                 required={inputContext.IsRequired} />
1560         </div>
1561     );
1562 }
1563
1564 // ----- Alter.js ----- :
1565 import "./Alter.css";
1566 import { useContext } from "react";
1567 import { UserContext } from "../contexts/UserContext";
1568
1569 export default function Alter({ msg = "Message To Show", msgColor = "green", children }) {
1570     let userData = useContext(UserContext);
1571     return (
1572         <div className="alterDiv">
1573             <h2 style={{ color: msgColor }} className="alterDivText">
1574                 {msg}
1575                 {msgColor === "green" && (
1576                     <h3>Welcome Mes {userData.username}</h3>
1577                 )}
1578             </h2>
1579             {children}
1580         </div>
1581     );
1582 }
1583
1584
1585 // Example 2 : -- []
1586
1587 // ----- context.js ----- :
1588 import { createContext } from "react";
1589 export let imgSizeContext=createContext(0);
1590 export let placeContext=createContext({})
1591
1592 // ----- App.js ----- :
```



```
1593 import { useState } from 'react';
1594 import { places } from './data.js';
1595 import { getImageUrl } from './utils.js';
1596 import { imgSizeContext, placeContext } from './Context.js'
1597
1598 export default function App() {
1599   const [isLarge, setIsLarge] = useState(false);
1600   const imageSize = isLarge ? 150 : 100;
1601   return (
1602     <>
1603     <label>
1604       <input
1605         type="checkbox"
1606         checked={isLarge}
1607         onChange={e => {
1608           setIsLarge(e.target.checked);
1609         }}
1610       />
1611       Use large images
1612     </label>
1613     <hr />
1614     <imgSizeContext.Provider value={imageSize}>
1615     <List />
1616     </imgSizeContext.Provider>
1617
1618   </>
1619 )
1620 }
1621
1622 function List() {
1623   const listItems =
1624     places.map(place =>
1625       <li key={place.id}>
1626         <placeContext.Provider value={place}>
1627           <Place />
1628         </placeContext.Provider>
1629       </li>
1630     );
1631   return (
1632     <ul>{listItems}</ul>
1633   );
1634
1635 }
1636
1637 import { useContext } from "react";
1638 function Place() {
1639   let place=useContext(placeContext)
1640   return (
1641     <>
1642     <PlaceImage
```

```
1643     />
1644     <p>
1645         <b>{place.name}</b>
1646         {': ' + place.description}
1647     </p>
1648 </>
1649 );
1650 }
1651
1652 function PlaceImage() {
1653     let imgSize=useContext(imgSizeContext);
1654     let place=useContext(placeContext)
1655     return (
1656         <img
1657             src={getImageUrl(place)}
1658             alt={place.name}
1659             width={imgSize}
1660             height={imgSize}
1661         />
1662     );
1663 }
1664
1665 // ----- data.js ----- :
1666 export const places = [{
1667     id: 0,
1668     name: 'Bo-Kaap in Cape Town, South Africa',
1669     description: 'The tradition of choosing bright colors for houses began in the late 20th century.',
1670     imageUrl: 'K9HVAGH'
1671 }, {
1672     id: 1,
1673     name: 'Rainbow Village in Taichung, Taiwan',
1674     description: 'To save the houses from demolition, Huang Yung-Fu, a local resident, painted all 1,200 of them in 1924.',
1675     imageUrl: '9EAYZrt'
1676 }, {
1677     id: 2,
1678     name: 'Macromural de Pachuca, Mexico',
1679     description: 'One of the largest murals in the world covering homes in a hillside neighborhood.',
1680     imageUrl: 'DgXHVwu'
1681 }, {
1682     id: 3,
1683     name: 'Selarón Staircase in Rio de Janeiro, Brazil',
1684     description: 'This landmark was created by Jorge Selarón, a Chilean-born artist, as a "tribute to the Brazilian people."',
1685     imageUrl: 'aeO3rpI'
1686 }, {
1687     id: 4,
1688     name: 'Burano, Italy',
1689     description: 'The houses are painted following a specific color system dating back to 16th century.',
1690     imageUrl: 'kxsph5C'
1691 }, {
1692     id: 5,
```

```
1693     name: 'Chefchaouen, Marocco',
1694     description: 'There are a few theories on why the houses are painted blue, including that the color repels mosquitos or that it
symbolizes sky and heaven.',
1695     imageUrl: 'rTqKo46'
1696   }, {
1697     id: 6,
1698     name: 'Gamcheon Culture Village in Busan, South Korea',
1699     description: 'In 2009, the village was converted into a cultural hub by painting the houses and featuring exhibitions and art
installations.',
1700     imageUrl: 'ZfQOOzf'
1701   }];
1702
1703   // ----- utils.js ----- :
1704   export function getImageUrl(place) {
1705     return (
1706       'https://i.imgur.com/' +
1707       place.imageUrl +
1708       '.jpg'
1709     );
1710   }
1711
1712   /** context management :
1713   */
1714   In React, **context** is used to share values across components without having
1715   to pass props manually through every level of the component tree.
1716   However, one of the potential downsides of using context
1717   is that components that consume the context will re-render whenever
1718   the value in the context changes.
1719
1720   Here's a detailed explanation of how and why this happens,
1721   and how it can lead to unnecessary re-renders:
1722   */
1723   ##### How Context Works with Re-rendering:
1724   */
1725   When you use React's `Context.Provider`, it provides a value to all
1726   components that consume it. Any time that the value provided by the
1727   `Provider` changes, all components that are consuming this context
1728   will re-render, even if the part of the value they rely on hasn't changed.
1729   */
1730
1731   /** Example:
1732   */
1733   - In this example, both `ChildComponentA` and `ChildComponentB` are consuming
1734     `MyContext`
1735
1736   - If `state` changes (e.g., `setState({ value1: 3, value2: 2 })`), **both**
1737     `ChildComponentA` and `ChildComponentB` will re-render.
1738
1739   - Even if `ChildComponentB` is only using `value2` (which hasn't changed),
1740     it will still re-render because the context object has changed.
```

```
1741
1742 - This is considered an **unnecessary re-render** for `ChildComponentB`.
1743 */
1744 const MyContext = React.createContext();
1745
1746 function ParentComponent() {
1747   const [state, setState] = useState({ value1: 1, value2: 2 });
1748
1749   return (
1750     <MyContext.Provider value={state}>
1751       <ChildComponentA />
1752       <ChildComponentB />
1753     </MyContext.Provider>
1754   );
1755 }
1756
1757 function ChildComponentA() {
1758   const context = useContext(MyContext);
1759   return <div>{context.value1}</div>; // Only uses value1
1760 }
1761
1762 function ChildComponentB() {
1763   const context = useContext(MyContext);
1764   return <div>{context.value2}</div>; // Only uses value2
1765 }
1766
1767 ##### Why Unnecessary Re-renders Can Be a Problem:
1768 /*
1769 1. Performance Impact: When components re-render unnecessarily,
1770    it can slow down the app, especially if the app is large, or if the
1771    component tree is deep and complex.
1772
1773 2. **State Synchronization**: If child components perform complex
1774    calculations or side effects on re-render, unnecessary re-renders
1775    might lead to inefficiency, causing redundant work.
1776
1777 3. **Component Bloat**: If many components are consuming the same
1778    context, it becomes harder to optimize re-renders, especially
1779    when only a small part of the context changes.
1780
1781 */
1782 // 1. **Memoizing Context Value**:
1783 /*
1784 - **Problem**: Passing a new object or function as context value on every
1785   render causes all consuming components to re-render.
1786
1787 - **Solution**: Use `useMemo` to memoize the context value so that
1788   it only changes when necessary.
1789 */
1790 /* Example:
```

```
1791  /*
1792    **Benefit**: With `useMemo`, the context value is only recalculated
1793    and changed when `value1` or `value2` changes, avoiding unnecessary
1794    re-renders of `ChildComponent`.
1795  */
1796  const MyContext = React.createContext();
1797
1798  function ParentComponent() {
1799    const [value1, setValue1] = useState(1);
1800    const [value2, setValue2] = useState(2);
1801
1802    const contextValue = useMemo(() => ({ value1, value2 }), [value1, value2]);
1803
1804    return (
1805      <MyContext.Provider value={contextValue}>
1806        <ChildComponent />
1807      </MyContext.Provider>
1808    );
1809  }
1810
1811  // 3. **Selector Functions**:
1812  /*
1813    - **Problem**: Context consumers might re-render even when they don't
1814      need all the context data.
1815
1816    - **Solution**: Create a custom hook or use a selector function to only
1817      extract the part of the context that the component needs.
1818  */
1819  Example:
1820  /*
1821    **Benefit**: This allows you to tightly control which parts of the context
1822    each component subscribes to, helping reduce unnecessary re-renders.
1823  */
1824  function useValue1() {
1825    const context = useContext(MyContext);
1826    return context.value1;
1827  }
1828
1829  function ChildComponentA() {
1830    const value1 = useValue1();
1831    return <div>{value1}</div>;
1832  }
1833
1834  /** Conclusion:
1835  */
1836  When using React's context, it's important to be mindful of how context changes
1837  affect re-rendering in consuming components. Splitting contexts, memoizing values,
1838  and using selector functions are effective strategies for avoiding unnecessary
1839  re-renders and keeping your application performance optimal.
1840  */
```

```
1841
1842 // React Router : --- []
1843
1844 // install the react router dom library :
1845 ```
1846   npm install react-router-dom --save
1847 ```
1848
1849 // in the index.js :
1850 import React from 'react';
1851 import ReactDOM from 'react-dom/client';
1852 import './index.css';
1853 import App from './App';
1854 import reportWebVitals from './reportWebVitals';
1855 import {BrowserRouter} from "react-router-dom"
1856
1857 const root = ReactDOM.createRoot(document.getElementById('root'));
1858 root.render(
1859   <React.StrictMode>
1860
1861     <BrowserRouter>
1862     <App />
1863   </BrowserRouter>
1864
1865 </React.StrictMode>
1866 );
1867
1868 // App.js : create new route Example :
1869 import './App.css';
1870 import { Route, Routes } from "react-router-dom";
1871 function App() {
1872   return (
1873     <
1874       <div className="App">
1875         <Routes>
1876           <Route path="/Home" element={ <h1>Hello From Home</h1> } />
1877           <Route path="/" element={ <h1>Hello From Home</h1> } />
1878         </Routes>
1879       </div>
1880     </>
1881   );
1882 }
1883 export default App;
1884
1885 // create a link Example :
1886 import './navBar.css';
1887 import { Link } from "react-router-dom";
1888
1889 export default function NabBar() {
1890   return (
```

```

1891     <nav>
1892         <h1>
1893             <span>M</span>ajid
1894         </h1>
1895         <ul>
1896             <li>
1897                 <Link to="/home">Home</Link>
1898             </li>
1899
1900             <li>
1901                 <Link to="/services">Services</Link>
1902             </li>
1903             <li>
1904                 <Link to="/about">About</Link>
1905             </li>
1906         </ul>
1907     </nav>
1908 );
1909 }
1910
1911 // Dynamic routing :
1912 // ----App.js :----
1913 ```
1914     <Route path="/pageName/:dynamicEndPoint" element={<ServiceDetails/>} ></Route>
1915 ```
1916
1917 // ----ServiceDetails.js :----
1918 import { useParams } from "react-router-dom";
1919
1920 import Service from "../Service/Service";
1921 import { servicesListContext } from "../contexts/ServicesContext";
1922 import { useContext } from "react";
1923 import ErrorPage from "../ErrorPage/ErrorPage";
1924 export default function ServiceDetails({ title, description }) {
1925     const { serviceId } = useParams();
1926     const servicesList = useContext(servicesListContext);
1927
1928     const targetService = servicesList.find((serviceItem) => {
1929         return serviceItem.id === serviceId;
1930     });
1931
1932     return (
1933         <
1934             <h1>Welcome to the service details page </h1>;<h1> Service id : {serviceId}</h1>
1935             {targetService ? <Service id={targetService.id} name={targetService.name} description={targetService.description}>
1936         </Service> : <ErrorPage />}
1937         </
1938     );
1939 }

```

```

1940 // add error page :
1941 ```
1942   <Route path="*" element={<ErrorPage />} />
1943 ```
1944
1945 // routes group:
1946 ```
1947   <Route path="/services" >
1948     <Route path=":serviceId" element={<ServiceDetails />} />
1949     <Route index element={<services/>}>
1950     <Route path="new" element={<NewService />} />
1951     <Route path="delete" element={<DeleteService />} />
1952   </Route>
1953 ```
1954
1955 // add layout to a routes group
1956 // ----App.js :----
1957 ```
1958 <Route path="/services" element={<ServiceLayout />}>
1959   <Route index element={<services/>}>
1960   <Route path="new" element={<NewService />} />
1961   <Route path="delete" element={<DeleteService />} />
1962 </Route>
1963 ```
1964 // ----ServicesLayout.js:----
1965 import { Outlet } from "react-router-dom";
1966 export default function ServiceLayout() {
1967   return (
1968     <div>
1969       <h1 style={{ width: "100vw", background: "red", color: "white" }}>Services</h1>
1970
1971       <Outlet />
1972     </div>
1973   );
1974 }
1975
1976 // download the material ui library :
1977 ```
1978   npm install @mui/material @emotion/react @emotion/styled --save
1979 ```
1980 // create a theme example :
1981 import "../App.css";
1982
1983 import { createTheme, ThemeProvider } from "@mui/material/styles";
1984
1985 import Button from "@mui/material/Button";
1986
1987 import { orange, green } from "@mui/material/colors";
1988
1989 import { Chip } from "@mui/material";

```



```

1990  const theme = createTheme({
1991    palette: {
1992      primary: {
1993        main: orange[500],
1994      },
1995      secondary: {
1996        main: green[500],
1997      },
1998    },
1999  });
2000  function App() {
2001    return (
2002      <ThemeProvider theme={theme}>
2003        <div className="App">
2004          <Button color="primary" variant="outlined">
2005            Click me
2006          </Button>
2007          <Chip label="primary" color="primary" variant="outlined" />
2008
2009        </div>
2010      </ThemeProvider>
2011    );
2012  }
2013
2014  export default App;
2015
2016  // install material icons :
2017  ```
2018    npm install @mui/icons-material --save
2019  ```
2020
2021  // using uuid library to generate unique ids ---[] :
2022    // install the library :
2023    ```
2024    npm install uuid --save
2025    ```
2026
2027    // use the library to generate a unique id :
2028    import { v4 as uuidv4 } from 'uuid';
2029    uuidv4(); // => '9b1deb4d-3b7d-4bad-9bdd-2b0d7b3dcb6d'
2030
2031
2032
2033  // use effect :-- [-]
2034  /*
2035    `useEffect` is one of React's most powerful hooks. It allows you to perform side effects
2036    in functional components. Side effects are operations like data fetching, manual
2037    DOM manipulation, subscribing to services, timers, etc. The `useEffect`
2038    hook is React's way of handling such operations in a declarative, clean way,
2039    while also supporting lifecycle management (like mounting, updating, and unmounting).

```

```
2040 */
2041
2042 // Basic Syntax
2043 useEffect(() => {
2044   // side effect logic (e.g., data fetching, setting up event listeners)
2045   return () => {
2046     // cleanup logic (e.g., removing event listeners)
2047   };
2048   }, [dependencies]);
2049
2050 // Key Concepts of `useEffect`
2051 /*
2052 1. Side Effects: These are operations that are not purely related to rendering
2053 the UI (like API calls, updating the document title, setting up listeners, etc.).
2054
2055 2. Dependencies Array: This controls when the effect runs. `useEffect`
2056 runs whenever the component renders, but you can control when it re-runs
2057 by specifying certain dependencies in the array. These dependencies are
2058 the variables or states that the effect depends on.
2059
2060 - No dependency array: If you don't pass any array, the effect runs
2061 after every render (mount and every update).
2062 - Empty array `[]`: If you pass an empty array, the effect runs only once,
2063 after the initial render (componentDidMount behavior).
2064 - With dependencies `[dep1, dep2]`: The effect runs only when any of the values
2065 in the array change. It acts as a watcher for those dependencies
2066 (componentDidUpdate behavior).
2067
2068 3. Cleanup Function: Sometimes, side effects need to be cleaned up
2069 (e.g., removing an event listener, clearing a timer, aborting an API request).
2070 You return a function inside `useEffect` that will be executed when the component
2071 is unmounted or before the effect is re-executed (if the dependencies change).
2072 */
2073
2074 // Example 1: Basic Use Case (Data Fetching)
2075 /*
2076 - This example fetches user data when the component mounts.
2077 The empty array `[]` ensures the effect runs only once.
2078 */
2079 import { useState, useEffect } from "react";
2080
2081 function UserList() {
2082   const [users, setUsers] = useState([]);
2083
2084   useEffect(() => {
2085     // Side effect: fetch data
2086     fetch('https://jsonplaceholder.typicode.com/users')
2087       .then((response) => response.json())
2088       .then((data) => setUsers(data));
2089   }, []); // Empty array: effect runs only once after initial render
```

```
2090
2091   return (
2092     <ul>
2093       {users.map((user) => (
2094         <li key={user.id}>{user.name}</li>
2095       ))}
2096     </ul>
2097   );
2098 }
2099
2100 export default UserList;
2101
2102 // Example 2: Effect with Dependencies
2103 /*
2104  - This effect updates the document's title every time `count` changes.
2105  The dependency array `[count]` ensures the effect only runs when `count`
2106  is updated.
2107  */
2108 import { useState, useEffect } from "react";
2109
2110 function Counter() {
2111   const [count, setCount] = useState(0);
2112
2113   useEffect(() => {
2114     document.title = `You clicked ${count} times`;
2115
2116     // Cleanup is not required here
2117   }, [count]); // Effect runs every time the 'count' value changes
2118
2119   return (
2120     <div>
2121       <p>You clicked {count} times</p>
2122       <button onClick={() => setCount(count + 1)}>Click me</button>
2123     </div>
2124   );
2125 }
2126
2127 // Example 3: Cleanup Effect (Event Listener)
2128 /*
2129  In this example, the event listener for `mousemove` is added when
2130  the component mounts. The cleanup function removes the event listener
2131  when the component unmounts to prevent memory leaks.
2132  */
2133 import { useState, useEffect } from "react";
2134
2135 function MouseTracker() {
2136   const [position, setPosition] = useState({ x: 0, y: 0 });
2137
2138   useEffect(() => {
2139     const updateMousePosition = (e) => {
```

```
2140     setPosition({ x: e.clientX, y: e.clientY });
2141   };
2142
2143   window.addEventListener("mousemove", updateMousePosition);
2144
2145   // Cleanup function to remove event listener
2146   return () => {
2147     window.removeEventListener("mousemove", updateMousePosition);
2148   };
2149 }, []); // Empty array: effect runs only on mount and cleanup on unmount
2150
2151 return (
2152   <div>
2153     Mouse position: {position.x}, {position.y}
2154   </div>
2155 );
2156 }
2157
2158 // When Does `useEffect` Run?
2159 /*
2160 - On Mount: If you pass an empty array `[]`, it behaves like `componentDidMount`
2161   and runs only once after the component renders for the first time.
2162
2163 - On Update: If you pass a list of dependencies `[dep1, dep2]`, the effect
2164   runs again whenever any of those dependencies change, similar
2165   to `componentDidUpdate`.
2166
2167 - On Unmount: The cleanup function returned by `useEffect` behaves like
2168   `componentWillUnmount`, executing just before the component is removed
2169   from the DOM or before the effect is rerun due to changes in dependencies.
2170 */
2171
2172 // Example 4: Conditional Side Effects
2173 /*
2174   You can conditionally run effects based on state or props:
2175 */
2176
2177 function FetchDataOnToggle({ shouldFetch }) {
2178   const [data, setData] = useState(null);
2179
2180   useEffect(() => {
2181     if (!shouldFetch) return;
2182
2183     fetch('https://jsonplaceholder.typicode.com/posts')
2184       .then((res) => res.json())
2185       .then((data) => setData(data));
2186   }, [shouldFetch]); // Effect only runs if 'shouldFetch' changes
2187
2188   return (
2189     <div>
```

```
2190     {shouldFetch ? <p>Fetched data!</p> : <p>No fetch initiated.</p>}
2191   </div>
2192 );
2193 }
2194
2195 // Common Mistakes with `useEffect`
2196 /*
2197 1. **Forget to pass the dependency array**: Without the array,
2198    the effect will run on every render, which can cause performance issues.
2199
2200 2. **Using state setters inside effects without dependency**:
2201    If you're setting state in an effect and don't include it in the
2202    dependencies, it can lead to stale closures or infinite loops.
2203
2204 3. **Wrong dependencies**: Always ensure that any external values used inside
2205    `useEffect` are listed in the dependency array. This includes props, states,
2206    or any other variables. Omitting necessary dependencies may cause bugs or
2207    stale values to be used.
2208 */
2209
2210 // Conclusion
2211 /*
2212 - Mounting, Updating, and Unmounting: `useEffect` can handle all phases
2213    of the component lifecycle.
2214
2215 - Dependency management: React automatically handles when the effect
2216    should run by observing the values in the dependency array.
2217
2218 - Cleanup: You can ensure proper memory management by using the
2219    cleanup function.
2220
2221 By understanding `useEffect`, you can manage side effects efficiently
2222 in React functional components!
2223 */
2224
2225 // Run React project in production environment:
2226 /*
2227 This command will:
2228 1. Convert your React project to pure HTML, CSS, and JavaScript code (compiling).
2229 2. Bundling: Collect all code and libraries into one directory for production.
2230 */
2231 ```
2232   npm run build
2233 ```
2234
2235 // 2. useMemo:
2236 /*
2237 - The `useMemo` hook allows you to memoize expensive calculations, preventing
2238   them from being recalculated on every re-render.
2239 */
```

```

2240  /*
2241  - This ensures that the `computeHeavyTask` function is only recalculated when
2242  `input` changes, reducing unnecessary work.
2243  */
2244  const expensiveCalculation = useMemo(() => {
2245    return computeHeavyTask(input);
2246  }, [input]);
2247
2248  // Example :
2249  import { Stack } from "@mui/material";
2250  import TodoListItem from "../TodoListItem.js/TodoListItem";
2251  import "../TodoList.css";
2252  import { useMemo } from "react";
2253  export default function TodoList({ arrTasksStat, setDeleteModalStat, deleteModalStat, editModalStat, setEditModalStat,
completeTask, filterStat }) {
2254    let filteredTasks = useMemo(() => {
2255      return arrTasksStat.filter((taskItem) => {
2256        if (filterStat === "completed") return taskItem.isCompleted;
2257        if (filterStat === "not completed") return !taskItem.isCompleted;
2258
2259        return true;
2260      });
2261    }, [arrTasksStat, filterStat]);
2262
2263    const memoizedTasks = useMemo(() => {
2264      return filteredTasks.map((taskItem) => <TodoListItem key={taskItem.id} taskItem={taskItem} deleteModalStat=
{deleteModalStat} setDeleteModalStat={setDeleteModalStat} editModalStat={editModalStat} setEditModalStat=
{setEditModalStat} completeTask={completeTask} />);
2265    }, [filteredTasks, deleteModalStat, editModalStat]);
2266    return (
2267      <Stack spacing={1} className="TodoListComponentClass">
2268        {memoizedTasks}
2269      </Stack>
2270    );
2271  }
2272
2273  // ##### 3. useCallback:
2274  /*
2275  - The `useCallback` hook memoizes callback functions, which is useful for avoiding
2276  re-creating functions every time a component re-renders.
2277  */
2278  const handleClick = useCallback(() => {
2279    console.log("Button clicked");
2280  }, []);
2281
2282  /// Turning the context into a provider
2283  /*
2284  To turn a context into a provider in React, you create a context using `React.createContext()`
2285  and wrap your component tree with the `Provider` component that comes with the context.
2286  The `Provider` component will allow you to pass down values (like state or functions)
2287  to any descendant component that consumes the context.

```

```
2288 */
2289 //Here's a step-by-step guide on how to set up and use a context provider in React:
2290 // */ ### 1. Create the Context ---[]
2291
2292 //In a separate file (e.g., `MyContext.js`), create a context using `React.createContext()`.
2293
2294
2295 import React, { createContext, useState } from 'react';
2296
2297 // Create the context
2298 export const MyContext = createContext();
2299
2300
2301 // ### 2. Create a Provider Component --[]
2302 /*
2303     Inside the same file (or a new one), create a provider component that
2304     uses `MyContext.Provider`. This component will hold any shared state and
2305     functions and pass them as values to the provider.
2306 */
2307
2308 // Create a provider component
2309 export const MyProvider = ({ children }) => {
2310   const [state, setState] = useState("Hello from context!");
2311
2312   // Value to be provided to consumer components
2313   const contextValue = {
2314     state,
2315     updateState: (newState) => setState(newState),
2316   };
2317
2318   return (
2319     <MyContext.Provider value={contextValue}>
2320       {children}
2321     </MyContext.Provider>
2322   );
2323 };
2324
2325
2326 // ### 3. Wrap Your App (or Part of Your App) with the Provider --[]
2327 /*
2328     To make the context available in your component tree, wrap
2329     your application (or a specific part of it) with the provider.
2330 */
2331
2332 // In `App.js`:
2333
2334 import React from 'react';
2335 import { MyProvider } from './MyContext';
2336 import SomeComponent from './SomeComponent';
2337
```

```
2338 function App() {
2339   return (
2340     <MyProvider>
2341       <SomeComponent />
2342     </MyProvider>
2343   );
2344 }
2345
2346 export default App;
2347
2348
2349 // ### 4. Consume the Context in a Component --[]
2350 /*
2351   Now you can access the context values in any descendant component
2352   using the `useContext` hook.
2353 */
2354
2355 // In `SomeComponent.js`:
2356 import React, { useContext } from 'react';
2357 import { MyContext } from './MyContext';
2358
2359 function SomeComponent() {
2360   const { state, updateState } = useContext(MyContext);
2361
2362   return (
2363     <div>
2364       <p>{state}</p>
2365       <button onClick={() => updateState("New value from SomeComponent!")}>
2366         Update State
2367       </button>
2368     </div>
2369   );
2370 }
2371
2372 export default SomeComponent;
2373
2374
2375 // ### Summary
2376
2377 // - **Create** the context using `React.createContext()`.
2378 // - **Wrap** your app with the context provider component.
2379 // - **Consume** the context in any component using `useContext(MyContext)`.
2380 /*
2381   This approach makes it easy to share state and functions across your component
2382   tree without prop drilling, making your code more organized and scalable.
2383 */
2384
2385 // ## Documentation: `useReducer` Hook in React with Example
2386
2387 // ### Overview
```



```
2388  /*
2389  The `useReducer` hook is a React function that provides an alternative to `useState` for managing complex state logic in
functional components. It is especially useful when:
2390  - The state transitions are complex.
2391  - State updates depend on previous states.
2392  - You want a centralized place to handle multiple state transitions.
2393  */
2394  // The `useReducer` hook works similarly to `Redux` reducers, encapsulating state logic in a single reducer function to handle
dispatched actions.
2395
2396  // ### Syntax
2397
2398  ```javascript
2399  const [state, dispatch] = useReducer(reducer, initialState);
2400  ```
2401
2402  // ##### Parameters:
2403  /*
2404  - **`reducer`**: A function that determines the next state based on the current state and an action.
2405  - **`initialState`**: The initial value for the state when the component is first rendered.
2406  */
2407
2408  // ##### Returns:
2409  /*
2410  - **`state`**: The current state managed by the reducer.
2411  - **`dispatch`**: A function to send actions to the reducer, triggering a state update.
2412  */
2413
2414
2415  // ### Example: Calculator with `useReducer`
2416  /*
2417  This example demonstrates how `useReducer` can manage a simple calculator that performs basic arithmetic operations.
2418  It uses a reducer function to handle operations like addition, subtraction, multiplication, and division.
2419  */
2420
2421  // ##### File Structure
2422  ```
2423  src
2424  |— App.js
2425  |— reducers
2426  |   |— resultReducer.js
2427  |— App.css
2428  ```
2429
2430  // ### Step 1: Define the Reducer Function
2431  /*
2432  In `src/reducers/resultReducer.js`, define the `resultReducer` function. This function manages the state logic based
2433  on the action type. Each action type corresponds to a basic arithmetic operation.
2434  */
2435  ```javascript
```

```
2436 // src/reducers/resultReducer.js
2437
2438 export function resultReducer(state, action) {
2439   const { firstValue, secondValue } = action.payload;
2440   switch (action.type) {
2441     case "addition":
2442       return firstValue + secondValue;
2443     case "subtraction":
2444       return firstValue - secondValue;
2445     case "multiplication":
2446       return firstValue * secondValue;
2447     case "division":
2448       return secondValue !== 0 ? firstValue / secondValue : "Cannot divide by zero";
2449     default:
2450       return state;
2451   }
2452 }
2453 ```
2454
2455 // In this reducer:
2456 /*
2457 - The `state` represents the current result.
2458 - Each `case` in the `switch` statement handles a specific action type by performing the corresponding arithmetic operation on
  `firstValue` and `secondValue`.
2459 */
2460
2461 // ### Step 2: Set Up `useReducer` in the Component
2462 /*
2463 In `src/App.js`, use `useReducer` to initialize the state and manage the dispatch function. Define helper functions to handle input
  and dispatch actions.
2464 */
2465
2466 // src/App.js
2467
2468 import React, { useReducer, useRef } from "react";
2469 import { resultReducer } from "../reducers/resultReducer";
2470 import { Stack, Button, FilledInput } from "@mui/material";
2471 import { blue } from "@mui/material/colors";
2472
2473 function App() {
2474   const firstInputRef = useRef();
2475   const secondInputRef = useRef();
2476
2477   // Initialize useReducer with the reducer function and an initial result of 0
2478   const [reducerResult, resultDispatch] = useReducer(resultReducer, 0);
2479
2480   // Function to retrieve the input values, converting them to floats or defaulting to 0
2481   function getInputsValues() {
2482     const firstValue = parseFloat(firstInputRef.current.value) || 0;
2483     const secondValue = parseFloat(secondInputRef.current.value) || 0;
```

```
2484     return { firstValue, secondValue };
2485   }
2486
2487   // Dispatch functions for each arithmetic operation
2488   function handleAddition(e) {
2489     e.preventDefault();
2490     resultDispatch({ type: "addition", payload: getInputsValues() });
2491   }
2492
2493   function handleSubtraction(e) {
2494     e.preventDefault();
2495     resultDispatch({ type: "subtraction", payload: getInputsValues() });
2496   }
2497
2498   function handleMultiplication(e) {
2499     e.preventDefault();
2500     resultDispatch({ type: "multiplication", payload: getInputsValues() });
2501   }
2502
2503   function handleDivision(e) {
2504     e.preventDefault();
2505     const { firstValue, secondValue } = getInputsValues();
2506     if (secondValue === 0) {
2507       alert("Cannot divide by zero");
2508       return;
2509     }
2510     resultDispatch({ type: "division", payload: { firstValue, secondValue } });
2511   }
2512
2513   return (
2514     <div className="App">
2515       <h1>Calculator Result</h1>
2516       <h2>{reducerResult}</h2>
2517
2518       <form>
2519         <Stack alignItems="center" spacing="10px">
2520           <FilledInput
2521             type="number"
2522             inputRef={firstInputRef}
2523             sx={{ width: "70%", borderLeft: `4px solid ${blue[500]} ` }}
2524             placeholder="Enter the first number"
2525           />
2526           <FilledInput
2527             type="number"
2528             inputRef={secondInputRef}
2529             sx={{ width: "70%", borderLeft: `4px solid ${blue[500]} ` }}
2530             placeholder="Enter the second number"
2531           />
2532         </Stack>
2533       </div>
```

```

2534     <Stack alignItems="center" sx={{ marginTop: "30px" }} spacing="15px">
2535       <Button variant="outlined" onClick={handleAddition} sx={{ width: "200px" }}>Add</Button>
2536       <Button variant="outlined" onClick={handleSubtraction} sx={{ width: "200px" }}>Subtract</Button>
2537       <Button variant="outlined" onClick={handleMultiplication} sx={{ width: "200px" }}>Multiply</Button>
2538       <Button variant="outlined" onClick={handleDivision} sx={{ width: "200px" }}>Divide</Button>
2539     </Stack>
2540   </form>
2541 </div>
2542 );
2543 }
2544
2545 export default App;
2546
2547
2548 // ### Explanation of Key Parts
2549
2550 // 1. useReducer Initialization:
2551 /*
2552   Here, useReducer initializes the reducerResult state to 0 and provides resultDispatch to handle actions.
2553 */
2554 const [reducerResult, resultDispatch] = useReducer(resultReducer, 0);
2555
2556 // 2. Dispatching Actions:
2557 /*
2558   Each button in the form triggers an arithmetic operation by dispatching an action with a
2559   specific type ("addition", "subtraction", etc.) and payload containing the input
2560   values.
2561 */
2562 // For example:
2563
2564 function handleAddition(e) {
2565   e.preventDefault();
2566   resultDispatch({ type: "addition", payload: getInputsValues() });
2567 }
2568
2569
2570 // 3. Helper Function getInputsValues:
2571 /*
2572   This function retrieves the numeric values from the input fields,
2573   defaulting to 0 if the input is empty or invalid.
2574 */
2575 // ### Summary of Benefits of Using useReducer Here
2576 /*
2577 - Centralized Logic: All arithmetic operations are handled in one place (resultReducer),
2578   making the logic easier to read and maintain.
2579 - Predictability: State changes happen only through resultReducer, ensuring predictable
2580   transitions based on action.type.
2581 - Scalability: Adding more operations (like exponentiation) requires just adding another
2582   case to resultReducer and creating a corresponding dispatch function.
2583 */

```

```
2584 // #### Pros and Cons of `useReducer`
2585
2586 // Pros |
2587 /*
2588     1- Centralized and modular logic for complex state updates
2589     2- Easily traceable actions, making debugging easier
2590     3- Facilitates future scalability in managing state transitions
2591 */
2592 // Cons |
2593 /*
2594     1- Can be overkill for simple state updates
2595     2- Might feel more complex than `useState`
2596     3- Not ideal for small, isolated state values
2597 */
2598
2599 // #### Best Practices
2600 /*
2601     - Use `useReducer` when managing complex state logic or when updates
2602       involve multiple actions or dependencies.
2603     - Keep the reducer function pure. Avoid side effects (like API calls or asynchronous logic)
2604       inside the reducer.
2605
2606     - Use constants for action types to avoid typos and make updates easier.
2607 */
2608 // #### When to Use `useReducer` vs `useState`
2609 /*
2610     - Use **`useState`** for simple state needs, like toggling a boolean or managing a small counter.
2611     - Use **`useReducer`** when:
2612     - The state has multiple sub-values or complex transitions.
2613     - State updates depend on previous state values.
2614     - The state transitions require explicit handling for clarity and scalability.
2615 */
2616
2617 // #### Final Notes
2618 /*
2619     This example demonstrates how `useReducer` can simplify handling complex state updates
2620     in a React functional component. It's particularly useful for modularizing logic in
2621     more advanced components or applications. By encapsulating state transitions in the
2622     reducer, the application remains organized, predictable, and easy to expand in functionality.
2623 */
2624
2625 // redux :
2626 /*
2627     Redux: A Comprehensive Guide
2628     Redux is a predictable state management library for JavaScript applications,
2629     commonly used with React but can be used with any JavaScript framework or library.
2630     It helps manage the state of an application in a consistent way, making debugging
2631     and testing easier.
2632 */
2633
```

```
2634 // ### **What is Redux Toolkit (RTK)?**
2635 /*
2636 Redux Toolkit is the official, opinionated way to write Redux applications. It simplifies
2637 common Redux tasks like:
2638 - Configuring the store.
2639 - Creating reducers and actions.
2640 - Handling complex state updates.
2641 - Managing side effects with tools like `createAsyncThunk`.
2642
2643 ---
2644
2645 ### **Core Features of Redux Toolkit**
2646
2647 1. **`configureStore`**:
2648   - Simplifies store setup with built-in support for middleware and DevTools.
2649
2650 2. **`createSlice`**:
2651   - Combines actions and reducers into a single "slice" of the state.
2652
2653 3. **`createAsyncThunk`**:
2654   - Simplifies handling asynchronous logic (e.g., API calls).
2655
2656 4. **DevTools Integration**:
2657   - Automatically integrates Redux DevTools without extra configuration.
2658
2659 ---
2660 */
2661 // ### **Modern Redux Workflow**
2662
2663 // ##### 1. **Install Redux Toolkit and React-Redux**:
2664   npm install @reduxjs/toolkit react-redux
2665
2666
2667
2668 // ##### 2. **Create a Slice**:
2669 // The `createSlice` method allows you to define the reducer and actions in one place.
2670
2671
2672 // features/counterSlice.js
2673 import { createSlice } from '@reduxjs/toolkit';
2674
2675 const counterSlice = createSlice({
2676   name: 'counter',
2677   initialState: { value: 0 },
2678   reducers: {
2679     increment: (state) => {
2680       state.value += 1; // RTK allows mutating state via Immer
2681     },
2682     decrement: (state) => {
2683       state.value -= 1;
```

```
2684     },
2685     incrementByAmount: (state, action) => {
2686       state.value += action.payload;
2687     },
2688   },
2689 });
2690
2691 export const { increment, decrement, incrementByAmount } = counterSlice.actions;
2692 export default counterSlice.reducer;
2693
2694
2695
2696
2697 // ##### 3. **Configure the Store**:
2698 // Use `configureStore` to set up your store, middleware, and reducers.
2699
2700 // app/store.js
2701 import { configureStore } from '@reduxjs/toolkit';
2702 import counterReducer from '../features/counterSlice';
2703
2704 export const store = configureStore({
2705   reducer: {
2706     counter: counterReducer, // Add slices here
2707   },
2708 });
2709
2710 export default store;
2711
2712
2713
2714 // ##### 4. **Connect Redux to React**:
2715 // Wrap your app with the `Provider` component to make the Redux store available throughout the component tree.
2716
2717
2718 // index.js
2719 import React from 'react';
2720 import ReactDOM from 'react-dom';
2721 import { Provider } from 'react-redux';
2722 import { store } from './app/store';
2723 import App from './App';
2724
2725 ReactDOM.render(
2726   <Provider store={store}>
2727     <App />
2728   </Provider>,
2729   document.getElementById('root')
2730 );
2731
2732
2733 // ##### 5. **Use Redux State and Dispatch in Components**:
```

```
2734 // Use `useSelector` to access the state and `useDispatch` to dispatch actions.
2735
2736
2737 // Counter.js
2738 import React from 'react';
2739 import { useSelector, useDispatch } from 'react-redux';
2740 import { increment, decrement, incrementByAmount } from './features/counterSlice';
2741
2742 const Counter = () => {
2743   const count = useSelector((state) => state.counter.value);
2744   const dispatch = useDispatch();
2745
2746   return (
2747     <div>
2748       <h1>{count}</h1>
2749       <button onClick={() => dispatch(increment())}>Increment</button>
2750       <button onClick={() => dispatch(decrement())}>Decrement</button>
2751       <button onClick={() => dispatch(incrementByAmount(5))}>Increment by 5</button>
2752     </div>
2753   );
2754 };
2755
2756 export default Counter;
2757
2758
2759
2760 // ### **Handling Async Logic with Redux Toolkit**
2761 /*
2762 For asynchronous operations (e.g., fetching data from an API),
2763 you can use `createAsyncThunk`.
2764 */
2765 // ##### 1. **Create an Async Thunk**:
2766
2767 // features/userSlice.js
2768 import { createSlice, createAsyncThunk } from '@reduxjs/toolkit';
2769
2770 // Async thunk to fetch users
2771 export const fetchUsers = createAsyncThunk('users/fetchUsers', async () => {
2772   const response = await fetch('https://jsonplaceholder.typicode.com/users');
2773   return response.json();
2774 });
2775
2776 const userSlice = createSlice({
2777   name: 'users',
2778   initialState: { data: [], status: 'idle', error: null },
2779   reducers: {},
2780   extraReducers: (builder) => {
2781     builder
2782       .addCase(fetchUsers.pending, (state) => {
2783         state.status = 'loading';
```



```
2784     })
2785     .addCase(fetchUsers.fulfilled, (state, action) => {
2786       state.status = 'succeeded';
2787       state.data = action.payload;
2788     })
2789     .addCase(fetchUsers.rejected, (state, action) => {
2790       state.status = 'failed';
2791       state.error = action.error.message;
2792     });
2793   },
2794 });
2795
2796 export default userSlice.reducer;
2797
2798
2799 // ##### 2. **Use the Thunk in a Component**:
2800
2801 import React, { useEffect } from 'react';
2802 import { useSelector, useDispatch } from 'react-redux';
2803 import { fetchUsers } from './features/userSlice';
2804
2805 const UserList = () => {
2806   const dispatch = useDispatch();
2807   const users = useSelector((state) => state.users.data);
2808   const status = useSelector((state) => state.users.status);
2809
2810   useEffect(() => {
2811     if (status === 'idle') {
2812       dispatch(fetchUsers());
2813     }
2814   }, [status, dispatch]);
2815
2816   return (
2817     <div>
2818       {status === 'loading' && <p>Loading...</p>}
2819       {status === 'succeeded' && (
2820         <ul>
2821           {users.map((user) => (
2822             <li key={user.id}>{user.name}</li>
2823           ))}
2824         </ul>
2825       )}
2826       {status === 'failed' && <p>Error loading users</p>}
2827     </div>
2828   );
2829 };
2830
2831 export default UserList;
2832
2833 // ### **Advantages of Redux Toolkit**
```

```
2834  /*
2835  1. Simplified Boilerplate:
2836  - Reduces the need to write separate action types, action creators, and switch statements.
2837
2838  2. Built-In Middleware:
2839  - Automatically includes `redux-thunk` for handling async logic.
2840
2841  3. Immer for Immutability:
2842  - Allows writing mutable-looking code that is internally immutable.
2843
2844  4. Developer-Friendly Defaults:
2845  - Pre-configured Redux DevTools support.
2846
2847  5. Scalable Structure:
2848  - Organizes your code into "slices," making it easy to manage large-scale applications.
2849
2850
2851  */
2852  // ### When to Use Redux Toolkit
2853  /*
2854  - Your app requires complex state management.
2855  - State is shared across multiple components.
2856  - You need to handle asynchronous operations (e.g., API calls).
2857  */
2858  /*
2859  For smaller applications, consider alternatives
2860  like React Context API if Redux feels too heavy.
2861  */
```