Project Structure (PO Back Office)

The project appears to be a React-based web application, with a modular structure. The code is organized into various directories, such as:

src: The main source code directory.

src/app: Contains application-wide components and utilities.

src/components: A directory for reusable UI components.

src/constants: Holds constant values used throughout the application.

src/hooks: Custom React hooks.

src/utils: Utility functions.

Technologies and Libraries

The project uses the following technologies and libraries:

React: The JavaScript library for building user interfaces.

Next.js: A popular React framework for building server-rendered, statically generated, and performance-optimized applications.

TypeScript: A superset of JavaScript that adds optional static typing and other features.

React-i18next: A library for internationalization (i18n) and localization (L10n) in React applications.

Redux: A state management library (although its usage is not explicitly shown in the provided code snippets).

Tanstack React Table: A library for building customizable and performant tables in React applications.

Material-UI (MUI): A popular UI component library for React (used in some components, like CommoditiesView.tsx).

Classnames: A utility library for conditionally joining CSS class names.

Component Architecture

The project uses a component-based architecture, with a focus on reusability and modularity. Components are organized into smaller, independent pieces, making it easier to maintain and update the application.

Some notable components include:

Button: A reusable button component with various props for customization.

Table: A customizable table component built using Tanstack React Table.

CommoditiesView: A component that displays a table with commodities data.

State Management

Although not explicitly shown in the provided code snippets, the project likely uses Redux or another state management library to manage global state. The useSelector hook from react-redux is imported in some components, indicating that Redux is used in the project.

Internationalization and Localization

The project uses React-i18next for internationalization and localization. This library helps manage translations and formatting for different languages and regions.

Other Notable Features

The project uses a feature flag system (imported from @config/featureFlags) to enable or disable certain features. (**launchdarkly)**

There are various utility functions and constants defined throughout the project, which are used to perform tasks like currency formatting and API requests.

Overall, the project appears to be a complex, data-driven web application with a focus on reusability, modularity, and maintainability.

**###########################################################################**

**User id:**

[1000031621@hexaware.com](mailto:1000031621@hexaware.com)

Ten@2019s

**To include font awesome icons in React js:**

**npm :**

*npm i --save @fortawesome/fontawesome-svg-core*

*npm install --save @fortawesome/free-solid-svg-icons*

*npm install --save @fortawesome/react-fontawesome*

**Yarn:**

*yarn add @fortawesome/fontawesome-svg-core*

*yarn add @fortawesome/free-solid-svg-icons*

*yarn add @fortawesome/react-fontawesome*

**Adding Additional Styles**

**Npm:**

*npm i --save @fortawesome/pro-solid-svg-icons*

*npm i --save @fortawesome/pro-regular-svg-icons*

*npm i --save @fortawesome/pro-light-svg-icons*

*npm i --save @fortawesome/pro-duotone-svg-icons*

**Yarn:**

*yarn add @fortawesome/pro-solid-svg-icons*

*yarn add @fortawesome/pro-regular-svg-icons*

*yarn add @fortawesome/pro-light-svg-icons*

*yarn add @fortawesome/pro-duotone-svg-icons*

///////////////////////////////////////////////////

How to flatten a nested array javascript?

We can achieve this using different ways but we will focus on a few. Let’s see how they work.

**1: concat() and apply()**

In this example,this is an *empty array*as seen in apply([], arr) and arr is the *argument list* we are passing (we’ve defined it above). So, we are saying take *this argument* and merge it with what you already got - another*empty array,*in this case. MSDN really explains [concat()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/concat" \t "_blank) and [apply()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function/apply)

let arr = [[1, 2],[3, 4],[5, 6, 7, 8, 9],[10, 11, 12]];

let flattened = [].concat.apply([], arr);// [ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 ]

## 2: spread operator in ES6

Here, we are using the [spread](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Spread_syntax" \t "_blank)operator to spread out the array. Then concatenate it with an empty array.

let flattened = [].concat(...arr);// [ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 ]

## 3: array.reduce()

The [reduce()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/reduce)method as the name suggests reduces an array to a single value. I have explained in great detail what it is and how to use it. check out the article [here](https://medium.com/swlh/how-to-reduce-array-to-a-single-type-11cfe2bcae66?source=friends_link&sk=20dc60517ab8edf27ec96e9f60524e94)

let flattened = arr.reduce((acc, curVal) => {return acc.concat(curVal)}, []);// [ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 ]

# **How to remove duplicate elements from JavaScript Array ?**

**Use the**[**filter() method**](https://www.geeksforgeeks.org/javascript-array-filter-method/)**:**The filter() method creates a new array of elements that pass the condition we provide. It will include only those elements for which true is returned. We can remove duplicate values from the array by simply adjusting our condition.

* Javascript

|  |
| --- |
| <script>      var arr = ["apple", "mango", "apple",              "orange", "mango", "mango"];        function removeDuplicates(arr) {          return arr.filter((item,              index) => arr.indexOf(item) === index);      } |

|  |
| --- |
| console.log(removeDuplicates(arr));  </script> |

**Output:**

["apple", "mango", "orange"]

**Using a**[**Set()**](https://www.geeksforgeeks.org/sets-in-javascript/)**Method:** This method sets a new object type with ES6 (ES2015) that allows you to create collections of unique values.

* Javascript

|  |
| --- |
| <script>      var arr = ["apple", "mango", "apple",              "orange", "mango", "mango"];        function removeDuplicates(arr) {          return [...new Set(arr)];      }        console.log(removeDuplicates(arr));  </script> |

**Output:**

["apple", "mango", "orange"]

**Using the [forEach() Method](https://www.geeksforgeeks.org/javascript-array-foreach-method/):**By using forEach() method, we can iterate over the elements in the array, and we will push into the new array if it doesn’t exist in the array.

* Javascript

|  |
| --- |
| <script>      var arr = ["apple", "mango",          "apple", "orange", "mango", "mango"];        function removeDuplicates(arr) {          var unique = [];          arr.forEach(element => {              if (!unique.includes(element)) {                  unique.push(element);              }          });          return unique;      }        console.log(removeDuplicates(arr));  </script> |

**Output:**

["apple", "mango", "orange"]

**Using the**[**reduce() Method**](https://www.geeksforgeeks.org/javascript-array-reduce-method/)**:**The reduce() method is used to reduce the elements of the array and combine them into a final array based on some reducer function that you pass.

* Javascript

|  |
| --- |
| <script>      var arr = ["apple", "mango",          "apple", "orange", "mango", "mango"];        function removeDuplicates(arr) {          var unique = arr.reduce(function (acc, curr) {              if (!acc.includes(curr))                  acc.push(curr);              return acc;          }, []);          return unique;      }        console.log(removeDuplicates(arr));  </script> |

**Output:**

["apple", "mango", "orange"]

**Using the**[**indexOf() Method**](https://www.geeksforgeeks.org/java-string-indexof/)**:**The indexOf() method is used to find the first index of occurrence of an array element. we can iterate over the elements in the array, and we will push into the new array if it doesn’t exist in the resultant array.

* Javascript

|  |
| --- |
| <script>      var arr = ["apple", "mango",          "apple", "orange", "mango", "mango"];        function removeDuplicates(arr) {          var unique = [];          for(i=0; i < arr.length; i++){              if(unique.indexOf(arr[i]) === -1) {                  unique.push(arr[i]);              }          }          return unique;      }        console.log(removeDuplicates(arr));  </script>  **Redux:**   * Creating 2 reducers: (CountReducer.js, LoggedInReducer.js)   CounterReducer.js  const CountReducer = (state = { count: 0 }, action) => {  switch (action.type) {  case "INCREMENT":  return { count: state.count + 1 };  case "DECREMENT":  return { count: state.count - 1 };  default:  return { count: state.count };  }  };  export default CountReducer;  LoggedInReducer.js:  const LoggedInReducer = (state = false, action) => {  switch (action.type) {  case "SIGN\_IN":  return !state;  default:  return state;  }  };  export default LoggedInReducer;   * Combining the reducers using the combineReducers() method in the index.js file   import LoggedInReducer from "./LoggedIn";  import CountReducer from "./Counter";  import { combineReducers } from "redux";  import { createStore } from "redux";  const allReducers = combineReducers({  counter: CountReducer,  isLogged: LoggedInReducer  });  const store = createStore(allReducers);  export default store;   * Import the store and embedded the component   import { StrictMode } from "react";  import { createRoot } from "react-dom/client";  import { BrowserRouter } from "react-router-dom";  import App from "./App";  *//import allReducers from "./Reducer";*  import store from "./Reducer";  import { Provider } from "react-redux";  const rootElement = document.getElementById("root");  const root = createRoot(rootElement);  root.render(  <StrictMode>  <BrowserRouter>  <Provider store={store}>  <App />  </Provider>  </BrowserRouter>  </StrictMode>  );   * Create the actions:   export const Increment = () => {  return { type: "INCREMENT" };  };  export const Decrement = () => {  return { type: "DECREMENT" };  };   * Invoked the state change in a component   import { useSelector, useDispatch } from "react-redux";  import { Increment, Decrement } from "./Actions/CounterAction";  const Page1 = () => {  let count = useSelector((state) => state.counter.count);  const dispatch = useDispatch();  console.log("count", count);  *//to flattern the array*  let arr = [  [1, 2],  [3, 4],  [5, 6, 7, 8, 9],  [10, 11, 12]  ];  *//let flattened = [].concat.apply([], arr);*  let flattened = [].concat.apply([], arr);  console.log("flattened:", flattened);  flattened = [].concat(...arr);  console.log("flat:", flattened);  let modifiedArray = arr.reduce((total, item) => {  return total.concat(item);  }, []);  console.log("arr:", modifiedArray);  *//to remove the duplicates*  let dupArr = [1, 1, 3, 4, 6, 3, 1, 2];  let uniArr = dupArr.filter((item, index) => dupArr.indexOf(item) === index);  console.log("uni:", uniArr);  const fnAddHandler = () => {  dispatch(Increment());  };  const fnSubHandler = () => {  dispatch(Decrement());  };  return (  <>  <h1>Welcome to Page1</h1>  <h3>Counter:{count}</h3>  <button onClick={fnAddHandler}>Add</button>  <button onClick={fnSubHandler}>Sub</button>  </>  );  };  export default Page1;  **Higher Order Function:** It is used if we need to use common functionality in more than one places  const mul = (a, b) => {  return a \* b;  };  const add = (a, b) => {  return a + b;  };  *//Higher Order Function*  const Calculator = (type) => {  switch (type) {  case "MUL":  return mul;  case "ADD":  return add;  default:  }  };  export default Calculator;  **Invoking Higher Order Functions**  import "./styles.css";  import { Routes, Route, NavLink } from "react-router-dom";  import Page1 from "./Page1";  import Calculator from "./HigherOrderFunction";  import Todo from "./Todo";  export default function App() {  const custFunc = Calculator("MUL");  return (  <div className="App">  <h1>Hello CodeSandbox</h1>  <h2>Start editing to see some magic happen!</h2>  <h2>HigherOrderFunction Sample</h2>  {custFunc(3, 7)}  <Todo />  <Routes>  <Route path="/page1" element={<Page1 />}></Route>  </Routes>  <NavLink to="/page1">Click Page1</NavLink>  </div>  );  }  **Higher Order Components:**  The component takes a component as a prop and returns a new component. This is used if we need to use the common logic in many components.  import { useState } from "react";  import React from "react";  const HigherOrderComponent = (OrginialComponent) => {  function NewComponent(props) {  const [counter, setCounter] = useState(0);  const increment = () => {  setCounter((prev) => prev + 1);  };  return (  <OrginialComponent counter={counter} increment={increment} {...props} />  );  }  return NewComponent;  };  export default HigherOrderComponent;  The components that use the common logic (Higher order components)  TextBoxComp.js  import React from "react";  const TextBoxComp = (props) => {  return (  <>  <input type="text" onChange={props.increment} />  <h1>Entered letters {props.counter}</h1>.  </>  );  };  export default TextBoxComp;  HoverComp.js  import React from "react";  const HoverComp = (props) => {  console.log("prpo:", props);  return (  <h2 onMouseOver={props.increment}>  Hover Component Counter {props.counter}  </h2>  );  };  export default HoverComp;  import "./styles.css";  import { Routes, Route, NavLink } from "react-router-dom";  import Page1 from "./Page1";  import Calculator from "./HigherOrderFunction";  import Todo from "./Todo";  import HigherOrderComp from "./HigherOrderComp";  import HoverComp from "./HoverComp";  import TextBoxComp from "./TextBoxComp";  *//Invoking the HigherComponent*  const UpdatedComponent = HigherOrderComp(HoverComp);  const UpdatedTxtComponent = HigherOrderComp(TextBoxComp);  console.log("Comp:", UpdatedComponent);  export default function App() {  const custFunc = Calculator("MUL");  return (  <div className="App">  <h1>Hello CodeSandbox</h1>  <h2>Start editing to see some magic happen!</h2>  <h2>HigherOrderFunction Sample</h2>  {custFunc(3, 7)}  {*/\* Invoked the updatedComponent using the higher order component \*/*}  <UpdatedComponent />  <UpdatedTxtComponent />  <Todo />  <Routes>  <Route path="/page1" element={<Page1 />}></Route>  </Routes>  <NavLink to="/page1">Click Page1</NavLink>  </div>  );  }  **Asynchronous task:**  **Promise is a constructor to handle nested callback**  const fetchData = (callback) => {  setTimeout(() => {  callback("Done");  }, 1500);  };  const print = () => {  setTimeout(() => {  console.log("Task is done.....");  fetchData((msg) => {  console.log(msg);  });  }, 2000);  };  print();  console.log("one");  console.log("two"); **UUID Generator libraray** To use this library we need to install it.  **npm install uuid**  when I was working with react + typescript the import statement that works was: import \* as uuid from 'uuid';  **To invoke in the code:**  import uuid from ‘uuid/v4’;  To use in the code:  empData = [{  id: uuid(),  name:”Max”,  }]  RiDeleteBin5Line **react-id-generator:** Generate unique id's in React components (e.g. for accessibility).  **Features:**   * Generates unique but predictable id's ✔︎ * Works with server-side rendering ✔︎ * TypeScript support ✔︎  **To Install:** npm i react-id-generator Basic example: import React from "react";  import nextId from "react-id-generator";  class RadioButton extends React.Component {  htmlId = nextId();  render() {  const { children, ...rest } = this.props;  return (  <div>  <label htmlFor={this.htmlId}>{children}</label>  <input id={this.htmlId} type="radio" {...rest} />  </div>  );  }  }  // Or with hooks:  import React from "react";  import { useId } from "react-id-generator";  const RadioButton = ({ children, ...rest }) => {  const [htmlId] = useId();  return (  <div>  <label htmlFor={htmlId}>{children}</label>  <input id={htmlId} type="radio" {...rest} />  </div>  );  };  Each instance of RadioButton will have unique htmlId like: id-1, id-2, id-3, id-4 and so on. nextId This is simple function that returns unique id that's incrementing on each call. It can take an argument which will be used as prefix:  import nextId from "react-id-generator";  const id1 = nextId(); // id: id-1  const id2 = nextId("test-id-"); // id: test-id-2  const id3 = nextId(); // id: id-3  NOTE: Don't initialize htmlId in React lifecycle methods like render(). htmlId should stay the same during component lifetime. useId This is a hook that will generate id (or id's) which will stay the same across re-renders - it's a function component equivalent of nextId. However, with some additional features.  By default it will return an array with single element:  const idList = useId(); // idList: ["id1"]  but you can specify how many id's it should return:  const idList = useId(3); // idList: ["id1", "id2", "id3"]  you can also set a prefix for them:  const idList = useId(3, "test"); // idList: ["test1", "test2", "test3"]  **New id's will be generated only when one of the arguments change.** resetId This function will reset the id counter. Main purpose of this function is to avoid warnings thrown by React durring server-side rendering (and also avoid counter exceeding Number.MAX\_SAFE\_INTEGER):  Warning: Prop id did not match. Server: "test-5" Client: "test-1"  While in browser generator will always start from "1", durring SSR we need to manually reset it before generating markup for client:  import { resetId } from "react-id-generator";  server.get("\*", (req, res) => {  resetId();  const reactApp = (  <ServerLocation url={req.url}>  <StyleSheetManager sheet={sheet.instance}>  <Provider store={store}>  <App />  </Provider>  </StyleSheetManager>  </ServerLocation>  );  const html = renderToString(reactApp);  res.render("index", { html });  }  This should keep ids in sync both in server and browser generated markup. setPrefix You can set prefix globally for every future id that will be generated:  import { setPrefix } from "react-id-generator";  setPrefix("test-id-");  const id1 = nextId(); // id: test-id-1  const id2 = nextId(); // id: test-id-2  const id3 = nextId("local"); // id: local-3 - note that local prefix has precedence  **Integrating google maps with React**   1. Need to install the package @react-google-maps/api  Go back to the previous page with React Router [#](https://bobbyhadz.com/blog/react-router-go-back-to-previous-page#go-back-to-the-previous-page-with-react-router) **To go back to the previous page with React router:**   1. Use the **useNavigate()** hook, e.g. **const navigate = useNavigate();**. 2. Call the **navigate()** function passing it **-1** - **navigate(-1)**. 3. Calling **navigate** with **-1** is the same as hitting the back button.   **App.js**  import {Link, Routes, Route, useNavigate} from 'react-router-dom';  export default function App() {  const navigate = useNavigate();  return (  <div>  <button onClick={() => navigate(-1)}>Go back</button>  <button onClick={() => navigate(1)}>Go forward</button>  <div>  <nav>  <ul>  <li>  <Link to="/">Home</Link>  </li>  <li>  <Link to="/about">About</Link>  </li>  </ul>  </nav>  {*/\* 👇️ Wrap your Route components in a Routes component \*/*}  <Routes>  <Route path="/about" element={<About />} />  <Route path="/" element={<Home />} />  </Routes>  </div>  </div>  );  }  function Home() {  return <h2>Home</h2>;  }  function About() {  return <h2>About</h2>;  } |
| **React v14 and ReacRouter v1.0.0 (Sep 10, 2015)** You can do this:  var React = require("react");  var Router = require("react-router");  var SomePage = React.createClass({  ...  contextTypes: {  router: React.PropTypes.func  },  ...  handleClose: function () {  if (Router.History.length > 1) {  // this will take you back if there is history  Router.History.back();  } else {  // this will take you to the parent route if there is no history,  // but unfortunately also add it as a new route  var currentRoutes = this.context.router.getCurrentRoutes();  var routeName = currentRoutes[currentRoutes.length - 2].name;  this.context.router.transitionTo(routeName);  }  },  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  **Using React Hooks**  Import:  import { useHistory } from "react-router-dom";  In stateless component:  let history = useHistory();  Call the Event:  history.goBack()  Examples do use in event Button:  <button onClick={history.goBack}>Back</button>  or  <button onClick={() => history.goBack()}>Back</button>  What is a Redux persist?  The primary purpose of Redux Persist is to persist the Redux store between sessions, ensuring that the application's state is saved and can be reloaded even after the app is restarted or the page is refreshed  ***Redux Persist****— It takes your [redux](https://redux.js.org/" \t "_blank)state and saves it to persisted storage (i.e., localStorage, sessionStorage and etc).*  ***localStorage****— localStorage is a property that allows JavaScript sites and apps to save key-value pairs in a web browser with no expiration date. This means the data stored in the browser will persist even after the browser window is closed.*  Now, let’s highlight some great features of Redux Persist —  1. One of the best features of the redux persists is that it provides us with the PersistGate — it helps us to delay the rendering until the state has been retrieved and saved to redux. Here we can show loading components until our state gets the persisted value.  2. Another great feature it provides Blacklist & Whitelist. By this, we can choose which state we want to persist in and which we don’t want.  3. It provides us with Transforms — It allows us to customize our object that gets persisted and rehydrated. By using this we can encrypt the object, filter it, set an expiration value for an object, compress, etc.  4. Provides us with different storage engines such as — localStorage, sessionStorage, AsyncStorage, etc.  5. The State Reconciler property provides us with different options to how to restore or merge the incoming state values at different deep and shallow levels.  6. There are also different custom functions such as persistReducer and persistStore which helps us to store and call the states. Now, why do we need to use it? So, now if we are developing an application on the production level we will be requiring at least 3 or 4 features from the above. During refresh or close of the tab, we need to restore the state once the application is running. Sometimes we need encryption of data that is stored in localStorage. Also, we need to show the loading symbols until our application gets the required data. Also, one of the scenarios is that suppose we are having approx 20 states in which we only want 5 of the states to be persisted and also all the states are interdependent to each other. In this case, the ideal solution will be using redux-persist. It will be saving us from a lot of packages installation. But I can still manually persist state in localStorage? Yes, we can but if we don’t want the above-mentioned features in our application. If our application is very small or if we require only one or two-state to be persisted and it is not dependent on other states.  If you are using react, wrap your root component with **[PersistGate](https://github.com/rt2zz/redux-persist/blob/HEAD/docs/PersistGate.md)**. This delays the rendering of your app's UI until your persisted state has been retrieved and saved to redux. **NOTE** the PersistGate loading prop can be null, or any react instance, e.g. loading={<Loading />}  **import** { PersistGate } **from** 'redux-persist/integration/react'    *// ... normal setup, create store and persistor, import components etc.*    const App **=** () => {  **return** (  **<**Provider store**=**{store}**>**  **<**PersistGate loading**=**{null} persistor**=**{persistor}**>**  **<**RootComponent **/>**  **</**PersistGate**>**  **</**Provider**>**    );  };  Redux Thunk is a middleware for Redux that enables asynchronous actions, allowing you to write action creators that return functions instead of action objects. These functions, called thunks, receive the dispatch and getState methods as arguments, enabling them to dispatch actions asynchronously, perform side effects, and access the store's state.  Key Features of Redux Thunk:   * **Asynchronous Actions:**   Thunks are crucial for handling asynchronous tasks like fetching data from an API or performing network requests, as they allow you to dispatch actions based on the outcome of these tasks.   * **Side Effects:**   Thunks can execute side effects, like making HTTP requests or interacting with external systems, without directly modifying the state within the reducer.   * **Accessing Store State:**   The getState method allows thunks to access the current state of the Redux store, enabling them to make decisions based on the current state before dispatching actions.   * Using dispatch:   The dispatch method is used within a thunk to dispatch new actions to the store once an asynchronous task is complete or other logic is executed.  How Thunks Work:   1. **1. Action Creator Returns a Function:**   Instead of returning an action object, an action creator with Redux Thunk returns a function that takes dispatch and getState as arguments.   1. **2. Asynchronous Logic:**   Within the function, you can perform asynchronous operations and then dispatch actions based on the result.   1. **3. Dispatching Actions:**   The dispatch method is used to send actions to the Redux store, allowing the reducers to update the state.  import { createStore, applyMiddleware } from 'redux';  import thunk from 'redux-thunk';  import { fetchPosts } from './actions'; // Assuming you have a fetchPosts action creator  // Create a store with thunk middleware  const store = createStore(  (state = { posts: [] }) => state, // A dummy reducer  applyMiddleware(thunk)  );  // Dispatch a thunk to fetch posts  store.dispatch(fetchPosts());  // The fetchPosts thunk would then:  // 1. Make a request to an API endpoint  // 2. Dispatch an action to indicate that the request is pending  // 3. Once the data is received, dispatch an action to update the posts in the store  // 4. If an error occurs, dispatch an action to indicate that there was an error.  Benefits of Using Redux Thunk:   * **Simpler than Redux Saga:** Redux Thunk is often considered easier to learn and use than Redux Saga. * **Less Boilerplate:** Thunks require less boilerplate code than Redux Saga, making them a good choice for projects with simple asynchronous needs. * **Good for Basic Side Effects:** Thunks are well-suited for handling basic side effects like fetching data or performing simple asynchronous operations.   In summary, Redux Thunk is a valuable tool for managing asynchronous actions in Redux, allowing you to write clean and maintainable code when handling side effects and complex logic in your applications.  How to give more classes for an element  const buttonClasses = classnames(appearanceClass, className, {      [styles.disabled]: disabled,      [styles.slim]: slim,      [styles.small]: small,      [styles.inverted]: inverted,      [styles.fullWidth]: fullWidth,    });    **ClassNames:**  In React, classnames is a utility library that simplifies the management of CSS classes, especially when dealing with conditional or dynamic class names. It provides a function that takes multiple arguments (strings or objects) and returns a single string containing the combined class names, handling falsy values and duplicates intelligently.  Example: |
| classnames helps to write cleaner and more maintainable code by avoiding verbose conditional logic or string manipulation for managing CSS classes. It is widely used in React projects for its simplicity and effectiveness. useLayoutEffect :  useLayoutEffect is a React Hook that fires synchronously after all DOM mutations, but before the browser has a chance to paint those changes to the screen. It's primarily used for reading layout from the DOM and synchronously re-rendering. This ensures the user never sees an inconsistent state.  useLayoutEffect shares its function signature with useEffect, accepting a setup function containing the effect's logic and an optional dependency array. |
| The key difference between useLayoutEffect and useEffect lies in their timing. useEffect runs asynchronously after the browser paints, making it suitable for most side effects like data fetching or subscriptions. useLayoutEffect, on the other hand, runs synchronously before the paint, which is crucial for tasks requiring immediate DOM measurements or manipulations to prevent visual glitches.  When deciding between useEffect and useLayoutEffect, consider these points:   * **Flickering Prevention:**   If you need to make DOM changes that must be visible before the next screen paint, use useLayoutEffect to avoid flickering.   * **Accurate Measurements:**   For scenarios requiring precise DOM measurements (e.g., element dimensions or positions), useLayoutEffect ensures measurements are taken after updates but before rendering.   * **PePerformance:**   Since useLayoutEffect blocks the browser paint, it can impact performance if overused. For most side effects, useEffect is the preferred choice due to its asynchronous nature. **Latest version of React**The latest stable version of React is 19.1.0, released in March 2025. This version includes improvements to debugging tools like Owner Stack, enhanced Suspense support, and various enhancements to React DOM and Server Components, including better error handling and server-side rendering. React 18 introduced significant performance improvements and new features, including automatic batching, concurrent rendering, and improvements to server-side rendering (SSR) with Suspense. These changes aim to make React applications more responsive, faster, and easier to maintain.  Here's a more detailed look at the key enhancements:  1. Automatic Batching:   * React 18 automatically groups multiple state updates into a single re-render, reducing unnecessary renders and improving performance. * This applies to state updates within event handlers, promises, setTimeout, or any other context, not just React events. * This feature simplifies development by allowing developers to update multiple states without worrying about manual batching.   2. Concurrent Rendering:   * React 18 introduced concurrency, allowing React to prepare multiple versions of the UI at the same time. * This makes applications more responsive by enabling React to switch between tasks without blocking the main thread. * Concurrent rendering enables smoother UI experiences, especially for complex applications, and allows for features like streaming HTML during server-side rendering.   3. Improvements to Server-Side Rendering (SSR) with Suspense:   * React 18 enhanced Suspense, enabling developers to stream HTML to clients as components load, improving perceived performance and user experience. * This is particularly beneficial for applications that rely on data fetching during the rendering process. |
|  |