GSSoC '24 Mentor | 7K+ Family @Linkedin | Mentor At @topmate.io | Building @CryptoMinds | MERN Stack Web Developer | Web 3.0 | Blockchain Developer | Career Counselor | Graphic Designer

Darshan Vasani

Helping Business

vasanidk30@gmail.com   https://linktr.ee/dpvasani56

React Part 1 Chai Aur Code

Installation

To create a new React application using Create React App, you can run the following command:

```
npx create-react-app appname
```

However, for better performance and faster build times, it is recommended to use Vite or Parcel instead.

React Setup Options

- **For Web Applications:** `Use React + React-DOM.`
- **For Mobile Applications:** `Use React + React-Native.`

This `package.json` file is for a React application named "01basicreate," with version "0.1.0." It includes several key sections:

1. Metadata:

- `"name": "01basicreate"`: The name of the project.
- `"version": "0.1.0"`: The current version of the project.
- `"private": true`: Indicates that the project is private and should not be published to the npm registry.

2. Dependencies:

- `"@testing-library/jest-dom": "^5.17.0"`: A library for DOM assertions in tests.
- `"@testing-library/react": "^13.4.0"`: Provides utilities for testing React components.
- `"@testing-library/user-event": "^13.5.0"`: Simulates user interactions in tests.
- `"react": "^18.3.1"`: The core React library.
- `"react-dom": "^18.3.1"`: React's DOM rendering package.
- `"react-scripts": "5.0.1"`: Provides a set of scripts for building and developing React apps.
- `"web-vitals": "^2.1.4"`: Measures web performance metrics like page load speed.

3. Scripts:

- `"start": "react-scripts start"`: Runs the development server.
- `"build": "react-scripts build"`: Builds the app for production.
- `"test": "react-scripts test"`: Runs the tests.
- `"eject": "react-scripts eject"`: Ejects the app from the default configuration, giving full control over the configuration files.

4. ESLint Configuration:

- `"extends": ["react-app", "react-app/jest"]`: Extends ESLint configuration from `react-app` and includes rules for Jest testing.

5. Browserslist:

- Specifies which browsers the app should support in production and development environments.
- `"production"`: Targets browsers with more than 0.2% market share, excluding obsolete ones.
- `"development"`: Targets the last version of Chrome, Firefox, and Safari for local development.

This file manages the project's dependencies, scripts, linting rules, and browser compatibility settings, serving as the core configuration for the React application.

Running and Building Your Application

- **Start the Application on Localhost**:

  ```
  npm run start
  ```

  This command starts the application on your local server.
- **Build for Production**:

  ```
  npm run build
  ```

  This command creates static assets for deploying the production version of your application.

Using Vite

Vite is a fast and efficient bundler for modern web development.

1. **Create a New Vite Project**:

   ```
   npm create vite@latest
   ```

2. **Install Dependencies**:
   After creating the Vite app, navigate to the project directory and run:

   ```
   npm install
   ```

3. **Run the Development Server**:

   ```
   npm run dev
   ```

   This starts the application on localhost for development.
4. **Build for Production**:

   ```
   npm run build
   ```

   This creates the production version of your application.

`manifest.json` File

- The `manifest.json` file is used to configure how your application behaves on small-sized devices, like mobile phones. It provides essential information for progressive web applications (PWAs) and improves the user experience on these devices.
- manifest.json provides metadata used when your web app is installed on a
  user's mobile device or desktop.

Folder Structure

When you create a React application using `npx create-react-app` or `npm create vite@latest` (with the React + JavaScript template), the folder structure generated by each tool is different due to the different build tools and project setups. Here's a comparison of the default folder structures for both:

1. `npx create-react-app` Folder Structure

When you run:

```
npx create-react-app my-app
```

It generates the following folder structure:

```
my-app/
├── node_modules/
├── public/
│   ├── favicon.ico
│   ├── index.html
│   ├── logo192.png
│   ├── logo512.png
│   ├── manifest.json
│   └── robots.txt
├── src/
│   ├── App.css
│   ├── App.js
│   ├── App.test.js
│   ├── index.css
│   ├── index.js
│   ├── logo.svg
│   ├── reportWebVitals.js
│   └── setupTests.js
├── .gitignore
├── package.json
├── README.md
└── yarn.lock (or package-lock.json)
```

- **`public/`**: Contains static assets like `index.html`, icons, and other files that won't be processed by Webpack. The `index.html` file is the main HTML template.
- **`src/`**: Contains the source code for your React application.
  - **`App.js`**: The main component of your application.
  - **`index.js`**: The entry point of your application where the React app is rendered into the DOM.
  - **`App.css`**, **`index.css`**: CSS files for styling.
  - **`reportWebVitals.js`**: For measuring performance.
  - **`setupTests.js`**: Used for setting up testing configurations.

2. `npm create vite@latest` with React + JavaScript Folder Structure

When you run:

```
npm create vite@latest my-app
```

And choose `React` and `JavaScript`, it generates the following folder structure:

```
my-app/
├── node_modules/
├── public/
│   └── vite.svg
├── src/
│   ├── assets/
│   │   └── react.svg
│   ├── App.css
│   ├── App.jsx
```

```
|   ├── index.css
|   ├── main.jsx
├── .gitignore
├── index.html
├── package.json
├── README.md
├── vite.config.js
└── yarn.lock (or package-lock.json)
```

- `public/`: Similar to Create React App, this folder contains static assets. The `index.html` file is not here, as Vite uses it differently.
- `src/`: Contains the source code for your React application.
  - `App.jsx`: The main component of your application (Vite typically uses `.jsx` for React components).
  - `main.jsx`: The entry point of your application where the React app is rendered into the DOM.
  - `assets/`: A folder for storing static assets like images.
- `index.html`: The main HTML template. Unlike Create React App, Vite keeps this file in the root directory.
- `vite.config.js`: The configuration file for Vite, where you can customize the build process and development server.

Summary of Differences:

- **Build Tool**: `create-react-app` uses Webpack, while `vite` uses Vite as the build tool.
- **Configuration**: Vite projects include a `vite.config.js` for configuration, while Create React App hides most configuration by default.
- **Static Files**: `create-react-app` puts static files in `public/` and HTML templates there as well. Vite keeps the `index.html` in the root and uses `public/` mainly for static assets that should be served as-is.
- **Component Files**: In Vite, React components often use the `.jsx` extension by default, whereas Create React App uses `.js` unless you specify `.jsx`.

Both tools create an initial project structure designed for quick development, with Vite offering faster build times and more flexibility for modern front-end development.

Main Entry Point Is Index.js → No Required To That Name Must Be Index.js In Can Be Anything.js

`<div id="root"></div>`

This

`const root = ReactDOM.createRoot(document.getElementById('root'));`

Main React Injection

In a React application created using `npx create-react-app`, React is rendered into the DOM through a specific root element, typically a `<div>` with an `id` of `root`. Here's a breakdown of how this process works:

1. **HTML Structure**: The root of the React application is defined in the HTML file (usually `public/index.html`). There, you have a `div` with an `id` of `root`:

   ```
   <!-- public/index.html -->
   <div id="root"></div>
   ```

2. **JavaScript Code**: The entry point of the React application (usually `src/index.js`) contains the code to render the React component tree into the DOM. This is done using the `ReactDOM.createRoot` method, which targets the `div` with the `id` of `root`.

   ```
   // src/index.js
   import React from 'react';
   import ReactDOM from 'react-dom/client'; // Note the use of 'react-dom/client' for React 1
   ```

```
8+
import './index.css';
import App from './App';

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(
  <React.StrictMode>
    <App />
  </React.StrictMode>
);
```

- `ReactDOM.createRoot`: This method is part of the React 18 API. It creates a root to manage rendering a React tree into a DOM element. It's an alternative to the older `ReactDOM.render` method, which is now deprecated in React 18.
- `React.StrictMode`: This is a wrapper component that helps highlight potential problems in the application. It doesn't render any visible UI itself but enables checks and warnings for its children components in development mode.
- `<App />`: This is the root component of your React application. Inside `App`, you can define the structure and behavior of the rest of your application.

In summary, the line `const root = ReactDOM.createRoot(document.getElementById('root'));` selects the `div` with `id="root"` and sets up a React root to manage rendering the application, starting with the `<App />` component, under React's strict mode guidelines.

Main Entry Point in React (`index.js`)

- **Main Entry Point**: The main entry point is the first JavaScript file that is executed when your application runs. In React projects created with `create-react-app`, this file is commonly named `index.js` by convention. However, it's not mandatory that this file must be named `index.js`; you can name it anything you like (e.g., `main.js`, `app.js`).
- **Customizing the Entry Point**: If you rename `index.js` to something else (e.g., `main.js`), you'll need to update the configuration or references in your build tool (like Webpack or Vite) to ensure that it knows which file to use as the entry point.

HTML Structure with `<div id="root"></div>`

- `<div id="root"></div>`: This is a placeholder `<div>` element in your `index.html` file. When you create a React application, this `div` is where your entire React application will be rendered. The `id="root"` is crucial because it's used to identify this element in your JavaScript code.

Rendering React Components to the DOM

- `const root = ReactDOM.createRoot(document.getElementById('root'));`: This line of code is used to initialize the root of your React application and attach it to the `div` with the `id="root"` in your HTML.

Here's a detailed breakdown:

1. `document.getElementById('root')`:
   - This function retrieves the DOM element with the `id` of `root`. In this case, it's the `<div>` in your `index.html` file.
   - This element is where your entire React component tree will be mounted.
2. `ReactDOM.createRoot`:
   - This method is used to create a root for rendering React components using the new React 18 API.
   - This method returns a root object that controls the rendering of your React components into the DOM.
3. **Rendering to the Root**:
   - After creating the root, you typically call the `render` method on it to render your React components.
   - Example:

     ```
     const root = ReactDOM.createRoot(document.getElementById('root'));
     root.render(<App />);
     ```

- Here, `<App />` is your main React component (typically the top-level component that contains all other components). This line renders your entire React application into the `root` DOM element.

Why Can the Entry Point Be Named Anything?

- **Build Tools**: When you use build tools like Webpack, Vite, or even simple bundlers, the name of the entry file doesn't necessarily have to be `index.js`. These tools allow you to specify which file is the entry point for the application.
- **Flexibility**: Naming flexibility is beneficial for organizing larger applications or following specific naming conventions that might be more descriptive than `index.js`.

In summary:

- The entry point file (`index.js` by convention) can be named anything, as long as your build tool is configured to use it.
- The `div` with `id="root"` in `index.html` is where your React application is rendered.
- The `ReactDOM.createRoot` method creates a React root, allowing you to render your React component tree into that `div`.

Understanding the Code Snippet

```
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(
  <React.StrictMode>
    <App />
  </React.StrictMode>
);
```

1. `const root = ReactDOM.createRoot(document.getElementById('root'));`

- **`document.getElementById('root')`**:
  - This part of the code accesses the DOM and retrieves the HTML element with the `id` of `root`. In the context of a typical React application, this corresponds to a `<div id="root"></div>` element in your `index.html` file.
  - This element serves as the container where your entire React application will be rendered.
- **`ReactDOM.createRoot`**:
  - `ReactDOM.createRoot` is a method introduced in React 18 that creates a root object for your application. This root object is a place where your React components will be rendered.
  - The `createRoot` method is part of the new concurrent rendering feature in React, which improves the rendering process by allowing React to interrupt rendering to focus on higher-priority updates, resulting in a more responsive UI.
- **Storing the Root in a Variable**:
  - The `const root = ...` part stores the root object returned by `ReactDOM.createRoot` in a variable named `root`. This variable will be used to control the rendering of your React components.

2. `root.render(...)`

- **Rendering the App**:
  - The `root.render(...)` method is called on the root object created earlier. This is where you tell React to render your application into the DOM.
- **`<React.StrictMode>`**:
  - `<React.StrictMode>` is a wrapper component provided by React that helps you write better code. It doesn't render anything visible in the UI, but it activates additional checks and warnings for its children during development. It helps identify potential issues like deprecated APIs, side effects in render methods, and more.
  - While `React.StrictMode` is useful in development, it is automatically stripped out in production builds, so it has no impact on performance in the final app.
- **`<App />`**:

    - `<App />` is a React component, typically the top-level component of your application. It's a common convention to name this main component `App`.
    - The `App` component is where you'll define the structure and behavior of your application. Inside `App`, you might have routes, other components, or UI elements.

3. What Does This Code Do?

- This code initializes and renders your entire React application. Specifically:
    - It identifies the root DOM element where the React app will be injected.
    - It creates a root rendering context using `ReactDOM.createRoot`.
    - It renders the `App` component (wrapped in `React.StrictMode` for development warnings) into the root DOM element.

4. Why Is This the Entry Point?

- **Entry Point Definition**:
    - The entry point of a React application is the file or script that initializes the application and begins the rendering process.
    - In this code snippet, the entry point is responsible for kicking off the rendering of your React component tree by calling `root.render(...)`.
- **Importance**:
    - This entry point is critical because it connects your React components to the actual HTML document (`index.html`), making it possible to display your app in the browser.

Summary:

- **Root Element**: The `document.getElementById('root')` selects the DOM element where your app will be rendered.
- **Root Creation**: `ReactDOM.createRoot` creates a root object that enables concurrent rendering, a new feature in React 18.
- **Rendering**: `root.render` renders your React components, starting with `App`, into the root DOM element.
- **Strict Mode**: Wrapping `App` in `<React.StrictMode>` helps you catch potential issues during development.

This code serves as the foundation for any React application, ensuring that your React component tree is properly connected to the DOM and rendered efficiently.

Function Usage in React

- Create a function and return HTML within it.
- Export the function, then import it and use it within `.render()` to render it.

React Script

- React handles all the scripting tasks for you.

Loading in Vite

- In a Vite project, JavaScript is directly loaded with the following script tag:
- React Is Injected Throught This Script

```
<script type="module" src="/src/main.jsx"></script>
```

Naming Conventions and File Extensions

- Function names should start with an uppercase letter, and the export process is similar.
- In Vite, ensure you use the `.jsx` file extension, not `.js`.

File Extensions in React

- Using `.js` files in Create-React-App is acceptable.

Important Considerations

- In Vite, the uppercase naming convention and rendering a single root element are important, especially when using the `.jsx` file extension.
- **Upper Case Game**: In Vite (and generally in React), component names must start with an uppercase letter. This is a common convention to differentiate React components from regular HTML elements.
- **One Element Render Game**: In Vite, as in React, when you render components, there must be a single root element returned from a component. This means that a component must return one top-level JSX element, even if it contains multiple child elements. You can use a React fragment (`<>...</>`) or a single container element like a `<div>` to wrap multiple elements.
- **.JSX File Extension**: When using Vite, you should use the `.jsx` file extension for files that contain JSX syntax. This helps Vite correctly identify and process the file as one containing JSX
- In Create-React-App, also follow the uppercase naming convention and ensure a single root element is rendered.

```
<>
<App />
<h2>React With | Darshan Vasani</h2>
</>
```

```
<App />
<h2>React With | Darshan Vasani</h2>
// Some time This Will Give Error As This Support Only One Element Render So We Use Above
// Syntex
```

Best Practice

→ Function Export → Are Called As Components

→ File And Components Name's First Letter Is Capital

→ In React → File.js

→ In Vite → File.jsx

→ Render One Element So Multiple Elements Can Be Wrap By

→ `<>Multiple Element</> Or <div>Multiple Element </div>`

→ In React Js Code Is There → File.js

→ In React Some HTML Code Is Return By Them → File.jsx

Best Practices

- **Function Export**: Functions that are exported and used as components should follow the naming conventions and structure for consistency and clarity.
- **File and Component Naming**:
  - The first letter of file names and component names should be capitalized.
- **File Extensions**:
  - In React projects, use `.js` for files.
  - In Vite projects, use `.jsx` for files containing JSX syntax.
- **Rendering Elements**:
  - Ensure that each component renders a single root element. Multiple elements can be wrapped in:
    - React Fragments: `<>Multiple Elements</>`
    - Or a container element: `<div>Multiple Elements</div>`

- **Code and File Usage**:
  - In React, code is typically found in files with the `.js` extension.
  - In React, files with the `.jsx` extension may contain HTML code returned by components.

Custom React

The code you provided is a simplified custom rendering function that mimics some of the functionality of React by manually creating and rendering DOM elements based on a JavaScript object that represents a React element. Here's a breakdown of how the code works:

1. The `reactElement` Object

This object is designed to represent a React-like element. It contains information about the type of HTML element to create (`type`), its attributes (`props`), and its content (`children`).

```
const reactElement = {
    type: 'a', // Specifies the type of HTML element to create (in this case, an anchor eleme
nt)
    props: {
        href: '<http://google.com>', // The URL the anchor element should point to
        target: '_blank', // Opens the link in a new tab
        children: 'Click me' // The text that will be displayed inside the anchor element
    }
};
```

2. Selecting the Main Container

This line selects the DOM element where the custom-rendered element will be appended:

```
const mainContainer = document.querySelector('#root');
```

Here, `#root` is the ID of the HTML element where the rendered content will be placed.

3. The `customRender` Function

This function takes the `reactElement` object and a `mainContainer` DOM element as arguments and handles the creation and insertion of the corresponding DOM element into the document.

```
function customRender(reactElement, mainContainer) {
    // Step 1: Create the DOM element based on the type specified in reactElement.type
    const domElement = document.createElement(reactElement.type);

    // Step 2: Set the inner content of the element
    // Check if reactElement.props.children exists, if so, set it as innerHTML
    if (reactElement.props.children) {
        domElement.innerHTML = reactElement.props.children;
    } else if (reactElement.children) {
        // If props.children is not available, use reactElement.children
        domElement.innerHTML = reactElement.children;
    }

    // Step 3: Set attributes like href and target if they exist in props
    if (reactElement.props.href) {
        domElement.setAttribute('href', reactElement.props.href);
    }

    if (reactElement.props.target) {
        domElement.setAttribute('target', reactElement.props.target);
    }
    // Step 4: Append the created DOM element to the main container
    mainContainer.appendChild(domElement);
}
```

4. How It Works Step-by-Step

- **Step 1**: The function starts by creating a new DOM element using `document.createElement(reactElement.type)`. The `type` property in `reactElement` specifies which HTML element to create (`a` in this case, meaning an anchor or link element).
- **Step 2**: The function then checks if the `reactElement` has a `props.children` property. If it exists, it sets the `innerHTML` of the newly created element (`domElement`) to the value of `props.children`. If `props.children` does not exist, it checks if `reactElement.children` exists and uses that instead.
- **Step 3**: Next, the function checks if there are any additional attributes (like `href` and `target`) in the `props` of the `reactElement`. If these attributes exist, they are added to the `domElement` using `setAttribute`.
- **Step 4**: Finally, the created `domElement` is appended to the `mainContainer`, effectively rendering the element on the page.

5. Execution

When `customRender(reactElement, mainContainer);` is called, it:

- Creates an anchor (`<a>`) element.
- Sets its `href` to `"<http://google.com>"`.
- Sets its `target` to `"_blank"`, meaning the link will open in a new tab.
- Sets the inner text to `"Click me"`.
- Appends this newly created element to the `#root` element in the DOM.

Outcome

The final rendered HTML inside the `#root` container will look like this:

```
<a href="<http://google.com>" target="_blank">Click me</a>
```

When clicked, the link will open Google in a new tab. This is a very basic custom rendering function and does not cover the full range of capabilities that React provides, but it serves as an introduction to how React elements can be represented and rendered using plain JavaScript.

Understanding React and Its Mechanisms

- **Tree Structure and Variable Updates**:
  - After the tree structure is established, variables are added as needed.
- **React.createElement**:
  - This function works through Babel to transform JSX into JavaScript code that React can understand.
- **Variable Updates in React**:
  - React efficiently handles updates to variables. When a variable changes, React ensures that all parts of the UI where the variable is used are updated. This is a core feature of React, designed to streamline UI management.
- **UI Changes and Hooks**:
  - Changes in the UI are reflected through hooks, which help manage state and side effects in functional components.
- **UI Updating**:
  - React controls UI updates, ensuring that the interface reflects changes in the underlying state or variables seamlessly.

Hooks

1. useState

The `useState` hook lets you add state to a functional component. It returns an array containing the current state value and a function to update it.

```
const [count, setCount] = useState(0);
```

- **count**: The current state value.
- **setCount**: A function to update the state.

2. useEffect

The `useEffect` hook lets you perform side effects in your components, such as data fetching, subscriptions, or manually changing the DOM. It runs after the first render and after every update.

```
useEffect(() => {
  document.title = `You clicked ${count} times`;
}, [count]);
```

- The second argument is an array of dependencies. The effect runs only when one of the dependencies changes.

3. useContext

The `useContext` hook allows you to access the context value directly in your functional components.

```
const value = useContext(MyContext);
```

- **MyContext**: The context object created by `React.createContext`.

4. useReducer

The `useReducer` hook is an alternative to `useState` for managing more complex state logic. It works similarly to the reducer pattern in Redux.

```
const [state, dispatch] = useReducer(reducer, initialState);
```

- **reducer**: A function that determines how the state changes based on actions.
- **initialState**: The initial state value.
- **dispatch**: A function to send actions to the reducer.

5. useCallback

The `useCallback` hook returns a memoized version of a callback function that only changes if one of the dependencies changes. This is useful for optimizing performance in child components that rely on reference equality.

```
const memoizedCallback = useCallback(() => {
  doSomething(a, b);
}, [a, b]);
```

6. useMemo

The `useMemo` hook returns a memoized value that only recomputes when one of the dependencies changes. This is useful for optimizing performance in expensive computations.

```
const memoizedValue = useMemo(() => computeExpensiveValue(a, b), [a, b]);
```

7. useRef

The `useRef` hook returns a mutable object that persists between renders. It's often used to reference DOM elements or store mutable values that do not trigger a re-render when updated.

```
const inputRef = useRef();
inputRef.current.focus();
```

- **current**: The mutable value stored in the ref.

8. useImperativeHandle

The `useImperativeHandle` hook allows you to customize the instance value that is exposed when using `ref` in a parent component. It's typically used in conjunction with `forwardRef`.

```
useImperativeHandle(ref, () => ({
  focus: () => {
    inputRef.current.focus();
  }
}));
```

9. useLayoutEffect

The `useLayoutEffect` hook is similar to `useEffect` but fires synchronously after all DOM mutations. It's often used when you need to make changes to the DOM and ensure that they are visible before the browser repaints.

```
useLayoutEffect(() => {
  // DOM updates
}, [dependencies]);
```

10. useDebugValue

The `useDebugValue` hook is used to display a label in React DevTools for custom hooks.

```
useDebugValue(isOnline ? 'Online' : 'Offline');
```

11. useTransition

The `useTransition` hook allows you to mark some state updates as non-urgent and give the user an opportunity to interact with the UI while the transition is in progress.

```
const [isPending, startTransition] = useTransition();

startTransition(() => {
  setState(newState);
});
```

- **isPending**: A boolean indicating if the transition is still ongoing.
- **startTransition**: A function to start the transition.

12. useDeferredValue

The `useDeferredValue` hook lets you defer a value until after a high-priority update is handled.

```
const deferredValue = useDeferredValue(value);
```

- **deferredValue**: A version of the value that may be deferred to lower-priority updates.

13. useId

The `useId` hook generates unique IDs that can be used for accessibility attributes like `id` and `for`.

```
const id = useId();
```

14. useSyncExternalStore

The `useSyncExternalStore` hook is used to subscribe to an external store that has a synchronous API. It's mainly intended for libraries rather than application code.

```
const state = useSyncExternalStore(subscribe, getSnapshot);
```

- **subscribe**: A function to subscribe to the external store.
- **getSnapshot**: A function to get the current state of the store.

15. useInsertionEffect

The `useInsertionEffect` hook is similar to `useLayoutEffect` but is used for injecting styles into the DOM. It's particularly useful for CSS-in-JS libraries.

```
useInsertionEffect(() => {
  // Inject styles
}, []);
```

Virtual DOM

In Main.jsx → createRoot → Behind The Scene Create DOM Like Structure

Main DOM And Its DOM Ko Compare Karta He And Unhi Chijo Ko Update Karta He Jo Actually UI Me Update Hui

Browser → Remove Whole DOM And Repaint Whole DOM Again → Page Reload

What is the Virtual DOM?

The Virtual DOM is an in-memory representation of the actual DOM (Document Object Model). It is a lightweight copy of the real DOM that React uses to track changes in the UI.

How Does It Work?

1. **Initial Rendering:**
    - When a React component renders for the first time, React creates a Virtual DOM representation of the UI elements. This VDOM is just a JavaScript object that mirrors the structure of the real DOM but is not directly tied to the browser.
2. **Updating the UI:**
    - When a component's state or props change, React re-renders the component and creates a new Virtual DOM tree that reflects the updated UI.
3. **Diffing Algorithm:**
    - React then compares the new Virtual DOM tree with the previous one to identify what has changed. This process is called "reconciliation." React uses an efficient algorithm to find the minimal number of changes between the old and new Virtual DOM trees.
4. **Patching the Real DOM:**
    - Once React determines the differences (known as "diffs"), it updates only those parts of the real DOM that have changed. This process is called "patching" the DOM. Instead of re-rendering the entire UI, React only updates the specific elements that need to change.

Why Use the Virtual DOM?

- **Performance Optimization:**
  The Virtual DOM improves performance by reducing the number of direct manipulations of the real DOM, which are typically expensive operations. By minimizing the updates to the real DOM, React ensures that the UI remains responsive, even with frequent state changes.
- **Declarative UI:**
  With the Virtual DOM, developers can write code in a declarative style. You describe how the UI should look based on the current state, and React takes care of updating the actual DOM to match that state.

- **Cross-Browser Compatibility:**
  The Virtual DOM abstracts away the differences in how browsers handle DOM updates, leading to more consistent and predictable behavior across different environments.

Example

Consider a simple example where a button click increments a counter:

```
function Counter() {
  const [count, setCount] = useState(0);

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>Click me</button>
    </div>
  );
}
```

- **Initial Render:** React creates a Virtual DOM tree for the `<div>`, `<p>`, and `<button>` elements.
- **Update:** When the button is clicked, the `count` state updates, triggering a re-render of the component. React creates a new Virtual DOM tree where the `<p>` element now reflects the updated `count`.
- **Diffing:** React compares the new VDOM tree with the old one and finds that only the text inside the `<p>` element has changed.
- **Patching:** React updates only the text inside the `<p>` element in the real DOM, leaving the rest of the DOM untouched.

Conclusion

The Virtual DOM is a powerful tool that allows React to efficiently manage UI updates by minimizing direct interactions with the real DOM. It enables smooth performance, even in complex applications, by reducing unnecessary re-renders and ensuring that only the parts of the DOM that actually change are updated.

React Use Fiber

https://github.com/acdlite/react-fiber-architecture

The goal of React Fiber is to increase its suitability for areas like animation, layout, and gestures. Its headline feature is incremental rendering: the ability to split rendering work into chunks and spread it out over multiple frames.

Other key features include the ability to pause, abort, or reuse work as new updates come in; the ability to assign priority to different types of updates; and new concurrency primitives.

I strongly suggest that you are familiar with the following resources before continuing:

- React Components, Elements, and Instances - "Component" is often an overloaded term. A firm grasp of these terms is crucial.
- Reconciliation - A high-level description of React's reconciliation algorithm.
- React Basic Theoretical Concepts - A description of the conceptual model of React without implementation burden. Some of this may not make sense on first reading. That's okay, it will make more sense with time.
- React Design Principles - Pay special attention to the section on scheduling. It does a great job of explaining the *why* of React Fiber.

What is reconciliation?

*reconciliation*The algorithm React uses to diff one tree with another to determine which parts need to be changed.*update*A change in the data used to render a React app. Usually the result of `setState`. Eventually results in a re-render.

Reconciliation is the algorithm behind what is popularly understood as the "virtual DOM." A high-level description goes something like this: when you render a React application, a tree of nodes that describes the app is generated and saved in memory. This tree is then flushed to the rendering environment — for example, in the case of a browser application, it's translated to a set of DOM operations. When the app is updated (usually via `setState`), a new tree is generated. The new tree is diffed with the previous tree to compute which operations are needed to update the rendered app.

Although Fiber is a ground-up rewrite of the reconciler, the high-level algorithm described in the React docs will be largely the same. The key points are:

- Different component types are assumed to generate substantially different trees. React will not attempt to diff them, but rather replace the old tree completely.
- Diffing of lists is performed using keys. Keys should be "stable, predictable, and unique."

The key points are:

- **In a UI, it's not necessary for every update to be applied immediately; in fact, doing so can be wasteful, causing frames to drop and degrading the user experience.**
- **Different types of updates have different priorities — an animation update needs to complete more quickly than, say, an update from a data store.**
- **A push-based approach requires the app (you, the programmer) to decide how to schedule work. A pull-based approach allows the framework (React) to be smart and make those decisions for you.**

React doesn't currently take advantage of scheduling in a significant way; an update results in the entire subtree being re-rendered immediately. Overhauling React's core algorithm to take advantage of scheduling is the driving idea behind Fiber.

What is a fiber?

We've established that a primary goal of Fiber is to enable React to take advantage of scheduling. Specifically, we need to be able to

- pause work and come back to it later.
- assign priority to different types of work.
- reuse previously completed work.
- abort work if it's no longer needed.

In order to do any of this, we first need a way to break work down into units. In one sense, that's what a fiber is. A fiber represents a **unit of work**.

To go further, let's go back to the conception of React components as functions of data, commonly expressed as

```
v = f(d)
```

It follows that rendering a React app is akin to calling a function whose body contains calls to other functions, and so on. This analogy is useful when thinking about fibers.

Tailwind Installation Guide

For a detailed installation guide for Tailwind CSS, visit: Tailwind CSS Installation.

React Component Structure

- **Segregation Based on Functionality**: In React, organize components based on their functionality rather than the technology or technology stack they are based on. This approach helps in maintaining a clear and manageable codebase.

VS Code Snippets

- `rfce`: Use this shortcut in Visual Studio Code to quickly generate a functional component with an export statement.

```
function card(props) {
    // console.log("props",props)
    console.log(props.username)
```

```
function card({username}) {
    // console.log("props",props)
    console.log(username)
```

```
function card(username,btnText="Default Value") {
    // console.log("props",props)
    console.log(username)
```

```
<button className="mt-2 inline-flex cursor-pointer items-center text-sm font-semibold text-white
          {props.btnText || "Default Value"};
        </button>
```

```
  let counter = 5;
  const AddValue = ()=>{
    counter += 1;
    setCounter(counter)
    setCounter(counter)
    setCounter(counter)
    console.log(counter);
  }
  // This Output Is 6 Because Of UseState -> Its Send Changes In Batches
```

```
 let counter = 5;
 const AddValue = ()=>{
    counter += 1;
    setCounter((prevCounter) => prevCounter + 1)
    setCounter((prevCounter) => prevCounter + 1)
    setCounter((prevCounter) => prevCounter + 1)
    console.log(counter);
  }
  // Fetch Previous Value And Than Update -> 8
```

Background Color Changer

To handle changing colors dynamically in the UI, you need to store the color values in a variable. Since the UI needs to reflect these changes, you should use state management.

How It Works

- **Store Color in State**: Use a state variable to keep track of the current color. This ensures that any change to the color will be automatically reflected in the UI.

- **Update UI**: When the color value in the state changes, React will re-render the component to reflect the new color in the user interface.

useEffect, useRef and useCallback

In React, hooks are special functions that allow you to "hook into" React features and manage state and side effects in functional components. Here's a detailed explanation of `useState`, `useCallback`, `useEffect`, and `useRef`, which are some of the most commonly used hooks:

1. `useState`

- **Purpose**: `useState` is a hook that allows you to add state to a functional component. It returns a pair: the current state value and a function that updates it.
- **Syntax**:

```
const [state, setState] = useState(initialState);
```

- **Parameters**:
  - `initialState`: The initial value of the state. It can be a primitive value, an object, or even a function.
- **Example**:

```
import React, { useState } from 'react';

function Counter() {
  const [count, setCount] = useState(0); // Initialize state with 0

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>Click me</button>
    </div>
  );
}
```

  In this example, `count` is the state variable, and `setCount` is the function to update `count`.

2. `useCallback`

- **Purpose**: `useCallback` is a hook that returns a memoized version of a callback function that only changes if one of the dependencies has changed. It's useful to prevent unnecessary re-creation of functions when the component re-renders, which can help optimize performance, especially in child components that rely on those functions.
- **Syntax**:

```
const memoizedCallback = useCallback(() => {
  // function body
}, [dependency1, dependency2, ...]);
```

- **Parameters**:
  - A callback function to be memoized.
  - An array of dependencies that determine when the callback should be updated.
- **Example**:

```
import React, { useState, useCallback } from 'react';

function ParentComponent() {
  const [count, setCount] = useState(0);
  const handleClick = useCallback(() => {
    setCount(count + 1);
  }, [count]); // The callback updates when `count` changes

  return <ChildComponent onClick={handleClick} />;
}
```

Here, `handleClick` is memoized and will only change when `count` changes.

3. `useEffect`

- **Purpose**: `useEffect` is a hook that allows you to perform side effects in your components. Examples of side effects include data fetching, setting up a subscription, or manually changing the DOM. `useEffect` runs after every render by default, but you can control it using dependencies.
- **Syntax**:

```
useEffect(() => {
  // effect logic
  return () => {
    // cleanup logic (optional)
  };
}, [dependency1, dependency2, ...]);
```

- **Parameters**:
  - A function that contains the side-effect logic. This function can optionally return a cleanup function.
  - An array of dependencies. If any dependency changes, the effect runs again.
- **Example**:

```
import React, { useState, useEffect } from 'react';

function Example() {
  const [count, setCount] = useState(0);

  useEffect(() => {
    document.title = `You clicked ${count} times`;
    // Cleanup (optional) would go here if necessary
  }, [count]); // Effect depends on `count`

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>Click me</button>
    </div>
  );
}
```

In this example, the document title updates every time `count` changes.

4. `useRef`

- **Purpose**: `useRef` is a hook that allows you to create a mutable object that persists for the full lifetime of the component. You can use it to store a reference to a DOM element or any other mutable value that doesn't require re-rendering the component when it changes.
- **Syntax**:

```
const myRef = useRef(initialValue);
```

- **Parameters**:
  - `initialValue`: The initial value of the ref object, usually `null` for DOM references.
- **Example**:

```
import React, { useRef, useEffect } from 'react';

function TextInputWithFocusButton() {
  const inputEl = useRef(null);

  const onButtonClick = () => {
    // Access the DOM node using current property
    inputEl.current.focus();
  };

  return (
    <div>
      <input ref={inputEl} type="text" />
      <button onClick={onButtonClick}>Focus the input</button>
    </div>
  );
}
```

  In this example, `useRef` is used to directly reference a DOM element and programmatically focus the input when the button is clicked.

Summary

- `useState`: Manages state in functional components.
- `useCallback`: Memoizes functions to prevent unnecessary re-creations.
- `useEffect`: Manages side effects (like data fetching, subscriptions).
- `useRef`: Maintains a mutable reference that persists across renders.

These hooks provide a way to use state and side effects in functional components, which was previously only possible in class components.

Memoization is an optimization technique used primarily to speed up programs by storing the results of expensive function calls and reusing the cached result when the same inputs occur again. By doing so, it avoids the need to re-compute results for previously encountered inputs, thus saving computation time.

Key Points About Memoization:

1. **Caching Results**: Memoization involves caching the results of function calls. When a memoized function is called with a particular set of arguments, it stores the result in a cache. If the function is called again with the same arguments, the result is retrieved from the cache instead of being recomputed.
2. **Pure Functions**: Memoization works best with pure functions—those that, given the same input, will always produce the same output without causing any side effects. This ensures that the cache remains valid and can be reliably used.
3. **Performance Improvement**: By avoiding redundant calculations, memoization can significantly improve the performance of functions, especially those with high time complexity, like recursive functions (e.g., calculating Fibonacci numbers).

Example of Memoization:

Let's take an example of a simple recursive function to calculate Fibonacci numbers:

```
function fibonacci(n) {
  if (n <= 1) return n;
  return fibonacci(n - 1) + fibonacci(n - 2);
}
```

In the above function, `fibonacci(n)` calls itself twice for each call, leading to an exponential number of calls. This results in a lot of redundant calculations. With memoization, we can optimize this:

```
function memoizedFibonacci() {
  const cache = {}; // Create a cache to store results

  return function fib(n) {
    if (n in cache) {
      return cache[n]; // Return cached result if available
    }
    if (n <= 1) {
      return n;
    }

    // Compute result and store it in the cache
    cache[n] = fib(n - 1) + fib(n - 2);
    return cache[n];
  };
}

const fibonacci = memoizedFibonacci();
console.log(fibonacci(10)); // Outputs: 55
```

In this memoized version:

- A cache (an object) is used to store the results of function calls.
- Before computing the Fibonacci of `n`, the function checks if the result is already cached.
- If the result is cached, it returns the cached result, avoiding redundant computation.
- If not, it computes the result, caches it, and then returns it.

Use in React:

In React, memoization is often used with hooks like `useMemo` and `useCallback` to optimize component re-renders:

- `useMemo`: Memoizes the result of a computation, only recomputing it when its dependencies change.
- `useCallback`: Memoizes a callback function, ensuring that the same function reference is returned unless dependencies change. This is particularly useful when passing functions to child components, preventing unnecessary re-renders.

Benefits of Memoization:

- **Efficiency**: Reduces the time complexity of functions that are called with the same arguments multiple times.
- **Performance**: Improves the performance of applications, especially those with computationally expensive operations.
- **Resource Management**: Saves computational resources by avoiding unnecessary processing.

Limitations of Memoization:

- **Memory Usage**: Caching results consume memory. In cases where the number of possible inputs is very large, it might lead to excessive memory consumption.
- **Cache Management**: Properly managing the cache (e.g., clearing old entries) is essential to avoid memory bloat.

Conclusion

Memoization is a powerful technique that can drastically improve the performance of your applications by caching and reusing results of expensive function calls. In the context of React and other modern JavaScript frameworks, it plays a crucial role in optimizing rendering performance and ensuring applications remain responsive and efficient.

In the provided React component, several React hooks are utilized to manage state, side effects, and references within the application. Let's go over each of these hooks—`useState`, `useCallback`, `useEffect`, and `useRef`—and explain how they are applied in the context of the code.

1. `useState`

`useState` is a React hook used to manage state in functional components. It returns a state variable and a function to update that state. Each call to `useState` creates a piece of state that is independent of the others.

Usage in the Code:

- **`length`**: This state variable controls the length of the password to be generated. It is initially set to `8` using `const [length, setLength] = useState(8)`. When the user changes the range input, `setLength` updates the `length` value.
- **`numberAllowed`**: This boolean state determines if numbers are allowed in the generated password. It is toggled by a checkbox, with `const [numberAllowed, setNumberAllowed] = useState(false)` initializing it to `false`.
- **`charAllowed`**: Similar to `numberAllowed`, this state determines if special characters are allowed in the password. It is initialized to `false` using `const [charAllowed, setCharAllowed] = useState(false)`.
- **`password`**: This state holds the generated password string. Initially set to an empty string with `const [password, setPassword] = useState("")`. The `passwordGenerator` function updates this state.

2. `useCallback`

`useCallback` is a React hook that returns a memoized version of the callback function. It only recreates the function if one of its dependencies has changed. This can improve performance by preventing unnecessary re-creation of functions on each render.

Usage in the Code:

- **`passwordGenerator`**: This function generates a password based on the current state values (`length`, `numberAllowed`, and `charAllowed`). By using `useCallback`, the component ensures that `passwordGenerator` is only recreated if one of these dependencies changes. This prevents unnecessary recalculations and renders.

  ```
  const passwordGenerator = useCallback(() => {
    // Password generation logic
  }, [length, numberAllowed, charAllowed, setPassword]);
  ```

- **`copyPasswordToClipboard`**: This function copies the generated password to the clipboard using the `passwordRef`. The memoization ensures this function is recreated only when `password` changes, optimizing the button click response.

  ```
  const copyPasswordToClipboard = useCallback(() => {
    passwordRef.current?.select();
    passwordRef.current?.setSelectionRange(0, 999);
    window.navigator.clipboard.writeText(password);
  }, [password]);
  ```

3. `useEffect`

`useEffect` is a React hook that allows you to perform side effects in function components. These effects can include fetching data, updating the DOM, and subscribing to event listeners. It takes a function and a dependency array; the effect runs when one of the dependencies changes.

Usage in the Code:

- The `useEffect` hook here is used to automatically generate a new password whenever the relevant dependencies change. If `length`, `numberAllowed`, `charAllowed`, or `passwordGenerator` changes, `useEffect` calls `passwordGenerator` to update the `password`.

  ```
  useEffect(() => {
    passwordGenerator();
  }, [length, numberAllowed, charAllowed, passwordGenerator]);
  ```

4. `useRef`

`useRef` is a React hook that returns a mutable ref object whose `.current` property is initialized to the passed argument (or `null`). Refs are primarily used to access or modify DOM elements directly, bypassing React's state management.

Usage in the Code:

- `passwordRef`: This ref is attached to the password input field to allow direct DOM manipulation, such as selecting the text when the "copy" button is clicked.

  ```
  const passwordRef = useRef(null);
  ```

  In the `copyPasswordToClipboard` function, `passwordRef.current?.select();` selects the input field's text, and `passwordRef.current?.setSelectionRange(0, 999);` sets the range for the text selection, making it ready for copying to the clipboard.

Summary

- `useState`: Manages the state for the password length, allowed characters, and the generated password.
- `useCallback`: Optimizes the password generation and clipboard copying functions by memoizing them, ensuring they are recreated only when their dependencies change.
- `useEffect`: Triggers the password generation whenever the relevant state changes.
- `useRef`: Provides a reference to the input field for direct DOM manipulation, facilitating easy text selection and copying.

These hooks work together to create a responsive and efficient password generator component. They help manage state, optimize performance, and interact with the DOM, showcasing the power of React hooks in building interactive user interfaces.

Custom Hooks

- **General Use of Hooks**: In most cases, hooks return JavaScript functionality or data.
  - **JavaScript Only**: If a custom hook returns only JavaScript (e.g., functions, objects, or variables), it's advisable to use a `.js` file extension for the hook.
  - **JSX Return**: If a custom hook returns JSX elements or involves rendering components, it's better to use a `.jsx` file extension to ensure proper handling by tools like Babel and Vite.

Yes, custom hooks in React are essentially functions. They allow you to extract and reuse stateful logic across multiple components. Here's a breakdown of how they work and why they are useful:

What is a Custom Hook?

A custom hook is a JavaScript function whose name starts with "use" and that may call other hooks inside it. The primary purpose of custom hooks is to encapsulate and share stateful logic between components without having to use higher-order components (HOCs) or render props.

Why Use Custom Hooks?

- **Reusability**: You can encapsulate complex logic in a custom hook and reuse it in multiple components.
- **Separation of Concerns**: Custom hooks help keep components clean and focused on rendering, while the logic is handled by the hook.
- **Maintainability**: By moving logic into a custom hook, you make your components simpler and easier to maintain.

How to Create a Custom Hook

Here's a basic example of a custom hook that manages form input state:

```
import { useState } from 'react';

// Custom Hook
function useForm(initialState) {
```

```
  const [formState, setFormState] = useState(initialState);

  const handleChange = (e) => {
    setFormState({
      ...formState,
      [e.target.name]: e.target.value,
    });
  };

  const resetForm = () => {
    setFormState(initialState);
  };

  return {
    formState,
    handleChange,
    resetForm,
  };
}

// Component using the Custom Hook
function MyForm() {
  const { formState, handleChange, resetForm } = useForm({ name: '', email: '' });

  return (
    <form>
      <input
        name="name"
        value={formState.name}
        onChange={handleChange}
        placeholder="Name"
      />
      <input
        name="email"
        value={formState.email}
        onChange={handleChange}
        placeholder="Email"
      />
      <button type="button" onClick={resetForm}>
        Reset
      </button>
    </form>
  );
}
```

Key Points

1. **Custom Hook Function**: A custom hook is just a function that can use other hooks. It returns an object or array that contains the values and functions that you want to expose to components.
2. **Naming Convention**: Custom hooks should start with the word "use" (e.g., `useForm`, `useFetch`). This naming convention helps identify hooks and ensures that React can enforce rules about hooks usage.
3. **Usage**: You call a custom hook in the same way as you would use built-in hooks. Inside a component, you call the custom hook and destructure the returned values and functions.

By using custom hooks, you can abstract away complex logic, making your components more readable and reusable.

Custom Hook: `useCurrencyInfo`

Purpose

The custom hook `useCurrencyInfo` is designed to fetch and provide currency information based on the `currency` argument passed to it. It abstracts the logic for fetching and managing currency data, so you don't have to repeat this logic in every component where you need currency data.

Code Explanation

Here's a detailed explanation of each part of the code:

```
import { useEffect, useState } from "react";
```

- `useEffect`: A hook that allows you to perform side effects in function components, such as data fetching or subscribing to external data sources.
- `useState`: A hook that allows you to add state to function components.

```
function useCurrencyInfo(currency) {
    const [data, setData] = useState({});
```

- `useCurrencyInfo`: This is the custom hook function. It takes a `currency` parameter, which specifies which currency information to fetch.
- `useState({})`: Initializes the state variable `data` with an empty object. `setData` is the function used to update this state.

```
    useEffect(() => {
        fetch(`https://cdn.jsdelivr.net/npm/@fawazahmed0/currency-api@latest/v1/currencies/
${currency}.json`)
            .then((res) => res.json())
            .then((res) => setData(res[currency]))
            .catch((error) => console.error("Error fetching currency data:", error));
    }, [currency]);
```

- `useEffect`: This hook runs the effect whenever the `currency` value changes.
  - `fetch`: Performs an HTTP request to retrieve data from the API.
  - `.then((res) => res.json())`: Parses the response JSON.
  - `.then((res) => setData(res[currency]))`: Updates the `data` state with the specific currency information from the response.
  - `.catch((error) => console.error("Error fetching currency data:", error))`: Logs any errors that occur during the fetch process.
- **Dependency Array**: `[currency]` specifies that the effect should re-run whenever the `currency` value changes.

```
    return data;
}
```

- `return data`: The custom hook returns the `data` state, which contains the currency information.

```
export default useCurrencyInfo;
```

- `export default`: Exports the `useCurrencyInfo` hook so it can be imported and used in other components.

How to Use This Hook

Here's an example of how you might use the `useCurrencyInfo` hook in a component:

```
import React from "react";
import useCurrencyInfo from "./useCurrencyInfo";

function CurrencyComponent({ currency }) {
    const currencyData = useCurrencyInfo(currency);

    return (
        <div>
            <h1>{currencyData.name}</h1>
            <p>Symbol: {currencyData.symbol}</p>
            <p>Exchange Rate: {currencyData.rate}</p>
        </div>
    );
}

export default CurrencyComponent;
```

Summary

- **Encapsulation**: The custom hook encapsulates the logic for fetching currency data, making it reusable in any component that needs this data.
- **Simplicity**: Components that use the `useCurrencyInfo` hook can focus on rendering and interacting with the data, while the hook handles the data fetching and state management.
- **Reusability**: You can use the `useCurrencyInfo` hook in multiple components and pass different currency values to fetch and display different currency information.

This approach helps keep your components clean and focused, improving code readability and maintainability.

Breakdown of the `InputBox` Component

Imports

```
import React, { useId } from 'react';
```

- **React**: The main library for building user interfaces.
- **useId**: A hook introduced in React 18, which generates unique IDs for accessibility. It is particularly useful when you need to pair input elements with labels using the `for` attribute.

Component Definition

```
function InputBox({
    label,
    amount,
    onAmountChange,
    onCurrencyChange,
    currencyOptions = [],
    selectCurrency = "usd",
    amountDisable = false,
    currencyDisable = false,
    className = "",
}) {
    const amountInputId = useId();
```

- **Function Component:** `InputBox` is a functional component that takes several props to customize its behavior.
- **Props**:
  - `label`: A string for the label of the amount input.
  - `amount`: The current value of the amount input.
  - `onAmountChange`: A callback function to handle changes in the amount input.
  - `onCurrencyChange`: A callback function to handle changes in the selected currency.

- `currencyOptions`: An array of currency options to populate the dropdown.
  - `selectCurrency`: The currently selected currency.
  - `amountDisable`: A boolean to disable the amount input field.
  - `currencyDisable`: A boolean to disable the currency dropdown.
  - `className`: A string for additional CSS classes to style the component.
- **`useId` Hook**: `const amountInputId = useId();` generates a unique ID for the input element. This ID is used for associating the input with its label for accessibility.

JSX Structure

```
    return (
        <div className={`bg-white p-3 rounded-lg text-sm flex ${className}`}>
            <div className="w-1/2">
                <label htmlFor={amountInputId} className="text-black/40 mb-2 inline-block">
                    {label}
                </label>
                <input
                    id={amountInputId}
                    className="outline-none w-full bg-transparent py-1.5"
                    type="number"
                    placeholder="Amount"
                    disabled={amountDisable}
                    value={amount}
                    onChange={(e) => onAmountChange && onAmountChange(Number(e.target.valu
e))}
                />
            </div>
            <div className="w-1/2 flex flex-wrap justify-end text-right">
                <p className="text-black/40 mb-2 w-full">Currency Type</p>
                <select
                    className="rounded-lg px-1 py-1 bg-gray-100 cursor-pointer outline-none"
                    value={selectCurrency}
                    onChange={(e) => onCurrencyChange && onCurrencyChange(e.target.value)}
                    disabled={currencyDisable}
                >
                    {currencyOptions.map((currency) => (
                        <option key={currency} value={currency}>
                            {currency}
                        </option>
                    ))}
                </select>
            </div>
        </div>
    );
}
```

- **Wrapper `div`**: A container with default styling for padding, background, and text size.
- **Amount Input Section**:
  - **Label**: Associated with the input using `htmlFor` and the ID generated by `useId`.
  - **Input Field**: Handles number input with `type="number"`, bound to `amount`, and updates via `onAmountChange`.
- **Currency Dropdown Section**:
  - **`<select>` Element**: A dropdown for selecting currency, populated with `currencyOptions`, and bound to `selectCurrency`.
  - **Options**: Generated dynamically from the `currencyOptions` array, ensuring each option is unique with a `key` attribute.

Export

```
export default InputBox;
```

- The `InputBox` component is exported as the default export of the module, making it available for import in other files.

How to Use the `InputBox` Component

Here's an example of how you can use the `InputBox` component in a parent component:

```
import React, { useState } from 'react';
import InputBox from './InputBox';

function App() {
    const [amount, setAmount] = useState(0);
    const [currency, setCurrency] = useState('usd');

    const handleAmountChange = (newAmount) => {
        setAmount(newAmount);
    };

    const handleCurrencyChange = (newCurrency) => {
        setCurrency(newCurrency);
    };

    return (
        <div className="App">
            <InputBox
                label="Enter Amount"
                amount={amount}
                onAmountChange={handleAmountChange}
                onCurrencyChange={handleCurrencyChange}
                currencyOptions={['usd', 'eur', 'inr']}
                selectCurrency={currency}
            />
        </div>
    );
}

export default App;
```

Summary

- `useId`: Generates a unique ID to associate the label with the input, improving accessibility.
- **Customization**: The component's behavior and appearance can be customized using props.
- **Reusability**: This component can be reused throughout the application wherever an input field for amount and a currency dropdown are needed.
- **State Management**: The component receives state and callback functions via props, making it a controlled component that updates the parent component's state.

This design pattern helps keep the component flexible and reusable, while also ensuring that it integrates smoothly with its parent components' state and behavior.

Crash Course: React Router with Vite

Installation

To set up React Router in a Vite project, run the following command:

```
npm install react-router-dom
```

Usage

- **Import Components**:

  ```
  import { Link, NavLink } from 'react-router-dom';
  ```

- **Replacing `<a>` Tags**:
  - Using a traditional `<a>` tag results in a full page **refresh**. For client-side navigation without page reloads, use the `Link` component instead:

    ```
    <Link to="/some-path">Go to Some Path</Link>
    ```

- **`NavLink` for Active States**:
  - The `NavLink` component provides additional functionality, such as applying styles to the active link. It is useful for navigation menus where you want to highlight the current page:

    ```
    <NavLink to="/some-path" activeClassName="active">Go to Some Path</NavLink>
    ```

- **Components for Navigation**:
  - Both `Link` and `NavLink` are provided by `react-router-dom` and are used for navigation in a React application. They facilitate client-side routing and improve user experience by avoiding full page reloads.

1. `Link` Component

Purpose:

- The `Link` component is used to create navigation links to different routes within your application. It is a straightforward replacement for the standard HTML `<a>` tag that works with the React Router.

Usage:

- `Link` is used when you simply need to navigate between routes without any special styling or active state handling. It allows navigation without reloading the page, leveraging the single-page application nature of React.

Syntax:

```
import { Link } from 'react-router-dom';

function App() {
    return (
        <div>
            <Link to="/home">Home</Link>
            <Link to="/about">About</Link>
        </div>
    );
}
```

2. `NavLink` Component

Purpose:

- The `NavLink` component extends the functionality of `Link` by providing additional styling capabilities, especially for indicating the active route. It is particularly useful for navigation bars where you want to highlight the currently active page or route.

Usage:

- `NavLink` is used when you need to apply specific styles to the active link (the link that corresponds to the current route). It automatically applies an `active` class to the link that matches the current URL.

Syntax:

```
import { NavLink } from 'react-router-dom';

function App() {
    return (
        <div>
            <NavLink to="/home" activeClassName="active-link">
                Home
            </NavLink>
            <NavLink to="/about" activeClassName="active-link">
                About
            </NavLink>
        </div>
    );
}
```

Key Differences

1. **Active Class Management**:
    - `Link` does not manage active states by default. It is just a basic navigation link.
    - `NavLink` automatically manages active states. It applies an `active` class to the link if the link's target route is active.
2. **Styling**:
    - With `Link`, you manually manage any active styling or state. You may need to check the current route and apply classes accordingly.
    - With `NavLink`, you can use `activeClassName` or `activeStyle` props to automatically apply styles when the link is active. This makes it easier to create a navigation bar with clear indication of the current route.
3. **Props**:
    - `Link` takes only the basic `to` prop to define the destination path.
    - `NavLink` can take additional props like `activeClassName`, `activeStyle`, `isActive`, etc., to customize how the link should behave and look when active.

Example Usage with Active Styling

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

function App() {
    return (
        <Router>
            <nav>
                <NavLink to="/home" activeClassName="active-link" className="nav-link">
                    Home
                </NavLink>
                <NavLink to="/about" activeClassName="active-link" className="nav-link">
                    About
                </NavLink>
                <Link to="/contact" className="nav-link">
                    Contact
                </Link>
            </nav>

            <Routes>
                <Route path="/home" element={<Home />} />
                <Route path="/about" element={<About />} />
                <Route path="/contact" element={<Contact />} />
            </Routes>
        </Router>
    );
}

function Home() {
    return <div>Home Page</div>;
}

function About() {
    return <div>About Page</div>;
}

function Contact() {
    return <div>Contact Page</div>;
}

export default App;
```

Summary

- Use `Link` for simple navigation links without the need for active state indication.
- Use `NavLink` for navigation items that need active styling or state indication, such as in navigation bars or menus.
- `NavLink` provides a built-in mechanism for styling active links, making it a preferred choice for creating intuitive, user-friendly navigation experiences.

`Link` Component

- **Usage in the Code**: The `Link` component is used for navigating to specific routes without involving any special active state styling. In this example, it is used for navigation links that don't need to indicate whether they are active.
- **Examples in the Code**:
    - The `Link` component is used to navigate to the home route with a logo image. Clicking on this link will take the user to the homepage without any additional visual cues about active state.

    ```
    <Link to="/" className="flex items-center">
        <img
            src="<https://alexharkness.com/wp-content/uploads/2020/06/logo-2.png>"
            className="mr-3 h-12"
            alt="Logo"
        />
    </Link>
    ```

    - Another use of `Link` is for "Log in" and "Get started" buttons. These links likely navigate to certain sections or pages, and they don't need to indicate an active state (i.e., whether the user is currently on the login page or not).

    ```
    <Link
        to="#"
        className="text-gray-800 hover:bg-gray-50 focus:ring-4 focus:ring-gray-300 font-med
    ium rounded-lg text-sm px-4 lg:px-5 py-2 lg:py-2.5 mr-2 focus:outline-none"
    >
        Log in
    </Link>
    <Link
        to="#"
        className="text-white bg-orange-700 hover:bg-orange-800 focus:ring-4 focus:ring-ora
    nge-300 font-medium rounded-lg text-sm px-4 lg:px-5 py-2 lg:py-2.5 mr-2 focus:outline-n
    one"
    ```

```
>
    Get started
</Link>
```

`NavLink` Component

- **Usage in the Code**: The `NavLink` component is used for navigation items where active state indication is necessary. It automatically applies styling to indicate which route is currently active, making it perfect for navigation menus where it's useful to show the user which page they are on.
- **Examples in the Code**:
    - `NavLink` is used in the unordered list (`<ul>`) to provide navigation links to different pages like Home, About, Contact, and Github. Each of these links changes its style based on whether its target route is the current active route.

    ```
    <NavLink
        to="/"
        className={({isActive}) =>
            `block py-2 pr-4 pl-3 duration-200 ${isActive ? "text-orange-700" : "text-gray-
    700"} border-b border-gray-100 hover:bg-gray-50 lg:hover:bg-transparent lg:border-0 hov
    er:text-orange-700 lg:p-0`
        }
    >
        Home
    </NavLink>
    ```

        - **Active Styling**: The `NavLink` uses a function to dynamically assign the class name based on whether the link is active. The `isActive` prop is used to determine if the link matches the current URL. If it does, it applies the `text-orange-700` class to highlight the active link; otherwise, it applies `text-gray-700`.

Summary

- `Link`:
    - Used for general navigation.
    - No built-in mechanism for active state styling.
    - Suitable for links that don't need active indication, such as buttons or logo navigation.
- `NavLink`:
    - Specialized version of `Link` for navigation menus.
    - Provides automatic active state indication using the `isActive` prop.
    - Ideal for highlighting the current page in navigation bars, helping users know their current location within the app.

Making Router Two Ways In `Main.jsx`

In React Router v6, you can define routes using two main methods: directly as an array of objects or using JSX with `createRoutesFromElements`. Both methods achieve the same goal of defining the navigation structure of the application, but each offers a different approach. Let's go through both methods using the provided code examples:

Method 1: Using an Array of Objects

This method involves defining routes using an array of route objects, which is straightforward and declarative. Here's how it would look:

```
const router = createBrowserRouter([
  {
    path: '/',
    element: <Layout />,
    children: [
```

```
        {
          path: "",
          element: <Home />
        },
        {
          path: "about",
          element: <About />
        },
        {
          path: "contact",
          element: <Contact />
        }
      ]
    }
]);
```

Explanation:

- `createBrowserRouter`: This function creates a router instance using the provided route configuration.
- **Routes Definition**:
  - Each route is an object with properties like `path` and `element`.
  - The root path (`/`) has the `Layout` component as its main element, indicating that `Layout` is the parent component for all nested routes.
  - `children` array specifies nested routes under the root path.
    - Empty path (`""`) denotes the default route, which maps to the `Home` component.
    - Other paths like `"about"` and `"contact"` map to their respective components (`About` and `Contact`).

Method 2: Using JSX with `createRoutesFromElements`

This method uses JSX to define routes, which can be more intuitive as it mirrors the structure of components in React. Here's the example:

```
const router = createBrowserRouter(
  createRoutesFromElements(
    <Route path='/' element={<Layout />}>
      <Route path='' element={<Home />} />
      <Route path='about' element={<About />} />
      <Route path='contact' element={<Contact />} />
      <Route path='user/:userid' element={<User />} />
      <Route
        loader={githubInfoLoader}
        path='github'
        element={<Github />}
      />
    </Route>
  )
);
```

Explanation:

- `createRoutesFromElements`: A function that takes JSX elements to define routes, making it easier to visualize the routing hierarchy and relationships.
- **JSX Route Definition**:
  - `<Route>` components are nested inside a parent `<Route>`, indicating the layout or parent-child relationships.
  - The root `<Route path='/' element={<Layout />}>` defines the base layout component.

- Child routes are defined by nesting `<Route>` components inside the parent route, similar to nesting components in JSX.
  - `<Route path='' element={<Home />} />` defines the default route for the root path, rendering the `Home` component.
  - `<Route path='about' element={<About />} />` and similar routes define the paths for the `About`, `Contact`, and other components.
  - Dynamic routes, such as `<Route path='user/:userid' element={<User />} />`, can include parameters (`:userid`) that get passed as props to the component.
  - The route `<Route loader={githubInfoLoader} path='github' element={<Github />} />` demonstrates how to use a loader function to fetch data before rendering the component.

When to Use Each Method

- **Array of Objects**: This method is useful when you want to define routes in a more structured and easily serializable way. It's also more concise for simple route definitions.
- **JSX with `createRoutesFromElements`**: This method is more intuitive for React developers because it mirrors the JSX structure. It's easier to read and visualize, especially with nested routes, and allows for directly embedding loader functions and other properties.

How the Routes are Used in the Application

Both methods define routes that are eventually passed to the `RouterProvider`, which is rendered at the root of the application:

```
ReactDOM.createRoot(document.getElementById('root')).render(
  <React.StrictMode>
    <RouterProvider router={router} />
  </React.StrictMode>
);
```

- **`RouterProvider`**: A component that provides the routing context to the rest of the app, enabling navigation and rendering based on the defined routes.
- The router instance, defined using either method, dictates how URL paths map to components.

Conclusion

Both methods have their place in React Router v6, and the choice largely depends on personal preference and the complexity of the routing needs. For more complex applications with many nested routes, the JSX-based approach (`createRoutesFromElements`) is often preferred due to its readability and closer alignment with React's component-based structure. For simpler applications or when a more programmatic approach is desired, the array of objects method can be more straightforward.

Outlet `<Outlet />`

The `Outlet` component in React Router is a placeholder for nested routes. It acts as a dynamic placeholder for where the child routes are rendered in the parent component's layout. Using `Outlet` allows you to define a common layout (like a template) for multiple routes, ensuring that the structure around the child components (e.g., headers, footers, sidebars) remains consistent while the main content changes based on the route.

Explanation of the Provided Code

Here's the provided `Layout` component code, broken down to explain how the `Outlet` functionality works:

```
import React from 'react';
import Header from './components/Header/Header';
import Footer from './components/Footer/Footer';
import { Outlet } from 'react-router-dom';

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

export default Layout;
```

Code Breakdown

1. **Import Statements**:
   - `React`: This is imported to use React components.
   - `Header` and `Footer`: These are imported components that represent the header and footer sections of the layout, respectively.
   - `Outlet`: Imported from `react-router-dom`, it serves as the placeholder for the nested routes' content.
2. **Layout Component**:
   - This component acts as a wrapper or a common layout for several other components or pages. It typically includes elements that are consistent across different pages, like headers, footers, and sidebars.
3. **JSX Structure**:
   - `<Header />`: This renders the `Header` component. It appears at the top of every page that uses this layout.
   - `<Outlet />`: This is the crucial part where `Outlet` comes into play. It serves as a placeholder for nested routes. Whatever component is associated with the child route will be rendered here.
   - `<Footer />`: This renders the `Footer` component. It appears at the bottom of every page that uses this layout.

How `Outlet` Works in This Example

In this example, the `Layout` component serves as a wrapper that includes both a `Header` and `Footer`. The `Outlet` is positioned between these two components:

- `<Header />`: Appears at the top of the page.
- `<Outlet />`: Dynamic content area. It displays different components based on the current route.
- `<Footer />`: Appears at the bottom of the page.

When a user navigates to different routes, the `Outlet` component will be replaced by the corresponding route's component. This allows you to maintain a consistent layout (with the `Header` and `Footer`) while the main content changes according to the route.

Example Usage in Routing

Consider this routing setup:

```
const router = createBrowserRouter(
  createRoutesFromElements(
    <Route path="/" element={<Layout />}>
      <Route path="" element={<Home />} />
      <Route path="about" element={<About />} />
      <Route path="contact" element={<Contact />} />
    </Route>
  )
);
```

- `<Layout />` is the root component for these routes.
- For each route (`/`, `/about`, `/contact`), the `Layout` component will render.

- The `<Outlet />` in the `Layout` will dynamically display:
  - The `<Home />` component when the path is `/`.
  - The `<About />` component when the path is `/about`.
  - The `<Contact />` component when the path is `/contact`.

Summary

- `Outlet` is a dynamic placeholder for child routes, allowing the main layout to remain the same while the content changes based on the active route.
- It helps in maintaining a consistent layout structure (with headers, footers, etc.) while varying the central content area.
- Using `Outlet` simplifies the management of nested routes by providing a clear and organized way to insert route-specific content into a common layout.

UseEffect → Components Mount → Its Load Everything Inside There

React Router Dom Official Documentation ⇒ Documentation

All Routes Can Be Nested

```
const router = createBrowserRouter(
  createRoutesFromElements(
    <Route path='/' element={<Layout />}>
      <Route path='' element={<Home />} />
      <Route path='about' element={<About />}>
      <Route path='SelfNested' element={<SelfNested />} />
       <Routes/>
      <Route path='contact' element={<Contact />} />
      <Route path='user/:userid' element={<User />} />
      <Route
      loader={githubInfoLoader}
      path='github'
      element={<Github />}
       />
    </Route>
  )
)

ReactDOM.createRoot(document.getElementById('root')).render(
  <React.StrictMode>
    <RouterProvider router={router} />
  </React.StrictMode>,
)
```

Dynamic Segments

Dynamic segments in routes allow parts of the URL to act as placeholders that can match any value. These segments are defined using a colon (`:`) followed by a variable name, such as `:projectId` or `:taskId`. They enable you to create more flexible and reusable routes.

Example Code:

```
<Route path="projects/:projectId/tasks/:taskId" />
```

In this example:

- `:projectId` and `:taskId` are dynamic segments. They can match any value in those positions of the URL.
- If the current location is `/projects/abc/tasks/3`, `:projectId` will be `abc`, and `:taskId` will be `3`.

Using Dynamic Segments:

1. **In `loader` and `action` Functions**: Dynamic segments are accessible in the `loader` and `action` functions via the `params` object.

```
<Route
  // Access dynamic segments in the loader
  loader={({ params }) => {
    params.projectId; // "abc"
    params.taskId; // "3"
  }}
  // Access dynamic segments in the action
  action={({ params }) => {
    params.projectId; // "abc"
    params.taskId; // "3"
  }}
  element={<Task />}
/>;


// Another Way
createBrowserRouter(
  createRoutesFromElements(
    <Route path="/" element={<Root />}>
      <Route path="contact" element={<Contact />} />
      <Route
        path="dashboard"
        element={<Dashboard />}
        loader={({ request }) =>
          fetch("/api/dashboard.json", {
            signal: request.signal,
          })
        }
      />
      <Route element={<AuthLayout />}>
        <Route
          path="login"
          element={<Login />}
          loader={redirectIfUser}
        />
        <Route path="logout" action={logoutUser} />
      </Route>
    </Route>
  )
);

// Or use plain objects
createBrowserRouter([
  {
    path: "/",
    element: <Root />,
    children: [
      {
        path: "contact",
        element: <Contact />,
      },
      {
        path: "dashboard",
        element: <Dashboard />,
        loader: ({ request }) =>
          fetch("/api/dashboard.json", {
            signal: request.signal,
          }),
      },
      {
        element: <AuthLayout />,
        children: [
          {
            path: "login",
            element: <Login />,
            loader: redirectIfUser,
          },
          {
            path: "logout",
            action: logoutUser,
          },
        ],
      },
    ],
  },
]);
```

- Here, `params.projectId` and `params.taskId` provide access to the dynamic segment values (`abc` and `3` respectively) in both `loader` and `action` functions.

2. **In Components with `useParams`**: Inside a component, you can use the `useParams` hook to access the values of dynamic segments.

```
function Task() {
  // Get dynamic segments using useParams
  const params = useParams();
  console.log(params.projectId); // "abc"
  console.log(params.taskId); // "3"
}
```

- `useParams` returns an object containing all the dynamic segments defined in the route.

3. **Using `useMatch` for Custom Matching**: The `useMatch` hook can match the current location against a pattern, returning parameters for that match.

```
function Random() {
  const match = useMatch("/projects/:projectId/tasks/:taskId");
  console.log(match.params.projectId); // "abc"
  console.log(match.params.taskId); // "3"
}
```

- `useMatch` provides a way to check if the current route matches a specific path pattern and extract the dynamic segment values.

Ranked Route Matching

React Router uses ranked route matching to determine the best match for a given URL. This ranking is based on several factors:

- Number of segments
- Static vs. dynamic segments
- Presence of splats (wildcards, e.g.,)

Example of Ranked Route Matching:

Consider the following routes:

```
<Route path="/teams/:teamId" />
<Route path="/teams/new" />
```

And the URL: `http://example.com/teams/new`

- Both routes match the URL: `/teams/:teamId` could interpret `new` as a `:teamId`, and `/teams/new` explicitly matches `new`.
- **Ranked Matching**: React Router ranks these routes and selects the most specific match. Here, `/teams/new` is more specific (it has a static segment) compared to `/teams/:teamId`, which is dynamic. Thus, `/teams/new` is chosen.

Summary of Concepts:

- **Dynamic Segments**: Allow you to capture parts of the URL as variables using `:` notation.
- `useParams` **Hook**: Provides access to dynamic segment values within a component.
- `useMatch` **Hook**: Used for matching custom paths and extracting parameters.
- **Ranked Route Matching**: React Router automatically picks the most specific route match, reducing the need to order routes manually.

Practical Use in an Application

Dynamic segments and ranked route matching allow building scalable applications where routes can handle various input patterns flexibly. You can create routes for specific IDs, handle generic and specific cases, and keep your routes organized and predictable.

These features help developers create rich, dynamic web applications with precise control over how users navigate through different parts of their app, ensuring a seamless and intuitive user experience.

Moral of the Story

To optimize data fetching and improve user experience, consider the following approach:

- **Pre-fetching Data**: Use loaders and actions to make API calls before the user interacts with the page. For example, if a user hovers over a link or performs an action that indicates they might navigate to a page, trigger the API call during this event. This allows data fetching to start earlier, potentially before the user fully navigates to the new page.
- **Leveraging Event Timing**: By initiating API calls on events like cursor hover, you can start data fetching ahead of time, ensuring that the data is ready when the user actually navigates.
- **Caching**: Once the data is fetched, store it in cache memory. This way, you avoid redundant API calls and improve the efficiency of data retrieval, resulting in a smoother and faster user experience.

In React, you can achieve pre-fetching data and optimizing performance through several techniques. Here's a detailed approach to implement this:

1. Pre-fetching Data with React Router and Loaders

If you're using React Router with a setup that supports loaders (like Remix), you can use loaders to fetch data before the user navigates to a new page. Here's how you can implement it:

- **Define Loaders**: In your route configuration, define loaders to fetch data before rendering the page.

  ```
  // Example with React Router (v6.4+)
  import { createBrowserRouter, RouterProvider, Route } from 'react-router-dom';
  import HomePage from './HomePage';
  import fetchHomeData from './fetchHomeData';

  const router = createBrowserRouter([
  ```

```
  {
    path: "/home",
    element: <HomePage />,
    loader: fetchHomeData, // This will fetch data before rendering HomePage
  },
  // other routes
]);

function App() {
  return (
    <RouterProvider router={router} />
  );
}

export default App;
```

- **Loader Function**: The `fetchHomeData` function will be used to pre-fetch data.

```
// fetchHomeData.js
async function fetchHomeData() {
  const response = await fetch('/api/home-data');
  const data = await response.json();
  return data;
}

export default fetchHomeData;
```

2. Event-Based Data Fetching

For scenarios where you want to start fetching data before a user clicks a link, you can use event listeners and React hooks:

- **Pre-fetch on Hover**:

```
import React, { useState, useEffect } from 'react';

function LinkWithPrefetch({ to, fetchData }) {
  const [data, setData] = useState(null);

  const handleMouseEnter = async () => {
    if (!data) {
      const response = await fetchData();
      setData(response);
    }
  };

  return (
    <a href={to} onMouseEnter={handleMouseEnter}>
      Go to {to}
    </a>
  );
}

// Example usage
function App() {
  const fetchData = async () => {
    const response = await fetch('/api/data');
    const result = await response.json();
    return result;
  };

  return (
    <div>
      <LinkWithPrefetch to="/page" fetchData={fetchData} />
    </div>
  );
}

export default App;
```

3. Caching Data

You can use local storage, session storage, or state management libraries to cache fetched data:

- **Using State and Local Storage**:

```
import React, { useState, useEffect } from 'react';

function fetchData() {
  return fetch('/api/data').then(response => response.json());
}

function App() {
  const [data, setData] = useState(() => {
    const cachedData = localStorage.getItem('data');
    return cachedData ? JSON.parse(cachedData) : null;
  });

  useEffect(() => {
    if (!data) {
      fetchData().then(fetchedData => {
        setData(fetchedData);
        localStorage.setItem('data', JSON.stringify(fetchedData));
      });
    }
  }, [data]);

  return (
    <div>
      {/* Render your data */}
    </div>
  );
}

export default App;
```

4. React Query for Data Fetching and Caching

Using a library like React Query can simplify data fetching and caching:

- **Install React Query**:

```
npm install @tanstack/react-query
```

- **Usage**:

```
import React from 'react';
import { useQuery } from '@tanstack/react-query';

const fetchData = async () => {
  const response = await fetch('/api/data');
  return response.json();
};

function App() {
  const { data, status } = useQuery(['data'], fetchData);

  if (status === 'loading') return <div>Loading...</div>;
  if (status === 'error') return <div>Error fetching data</div>;

  return (
    <div>
      {/* Render your data */}
    </div>
  );
}

export default App;
```

These techniques will help you pre-fetch and cache data effectively in a React application, improving performance and user experience.

Yes, there are hooks and libraries that can help with pre-fetching and caching data in React. Here are a few approaches using React hooks and popular libraries:

5. Using `useEffect` for Data Fetching

You can use the built-in `useEffect` hook to fetch data when a component mounts or based on specific conditions:

```
import React, { useState, useEffect } from 'react';

function App() {
  const [data, setData] = useState(null);
  const [loading, setLoading] = useState(true);
  const [error, setError] = useState(null);

  useEffect(() => {
    const fetchData = async () => {
      try {
        const response = await fetch('/api/data');
        if (!response.ok) throw new Error('Network response was not ok');
        const result = await response.json();
        setData(result);
      } catch (error) {
        setError(error);
      } finally {
        setLoading(false);
      }
    };

    fetchData();
  }, []); // Empty dependency array means this runs once when the component mounts

  if (loading) return <div>Loading...</div>;
  if (error) return <div>Error: {error.message}</div>;

  return <div>{/* Render your data here */}</div>;
}

export default App;
```

6. React Query

React Query is a powerful library that simplifies data fetching, caching, and synchronization. It provides hooks that manage data fetching and caching for you.

- **Install React Query**:

```
npm install @tanstack/react-query
```

- **Setup**:

```
import React from 'react';
import { useQuery, QueryClient, QueryClientProvider } from '@tanstack/react-query';

const queryClient = new QueryClient();

const fetchData = async () => {
  const response = await fetch('/api/data');
  if (!response.ok) throw new Error('Network response was not ok');
  return response.json();
};

function DataComponent() {
  const { data, error, isLoading } = useQuery(['data'], fetchData);

  if (isLoading) return <div>Loading...</div>;
  if (error) return <div>Error: {error.message}</div>;

  return <div>{/* Render your data here */}</div>;
}

function App() {
  return (
    <QueryClientProvider client={queryClient}>
      <DataComponent />
    </QueryClientProvider>
  );
}

export default App;
```

7. React Query's `usePrefetch` Hook

React Query provides a `useQuery` hook that can be used to pre-fetch data ahead of time. While React Query itself doesn't have a specific `usePrefetch` hook, you can achieve pre-fetching by calling the `queryClient.prefetchQuery` method.

```
import React from 'react';
import { useQuery, QueryClient, QueryClientProvider, useQueryClient } from '@tanstack/react-q
uery';

const queryClient = new QueryClient();
const fetchData = async () => {
  const response = await fetch('/api/data');
  if (!response.ok) throw new Error('Network response was not ok');
  return response.json();
};

function PreFetchComponent() {
  const queryClient = useQueryClient();

  React.useEffect(() => {
    queryClient.prefetchQuery(['data'], fetchData);
  }, [queryClient]);

  return null; // This component is for pre-fetching and does not render anything
}

function DataComponent() {
  const { data, error, isLoading } = useQuery(['data'], fetchData);

  if (isLoading) return <div>Loading...</div>;
  if (error) return <div>Error: {error.message}</div>;

  return <div>{/* Render your data here */}</div>;
}

function App() {
  return (
    <QueryClientProvider client={queryClient}>
      <PreFetchComponent />
      <DataComponent />
    </QueryClientProvider>
  );
}

export default App;
```

8. Custom Hook for Pre-fetching

You can also create a custom hook to handle pre-fetching:

```
import { useState, useEffect } from 'react';

function usePreFetch(url) {
  const [data, setData] = useState(null);
  const [loading, setLoading] = useState(true);
  const [error, setError] = useState(null);

  useEffect(() => {
    const fetchData = async () => {
      try {
        const response = await fetch(url);
        if (!response.ok) throw new Error('Network response was not ok');
        const result = await response.json();
        setData(result);
      } catch (error) {
        setError(error);
      } finally {
        setLoading(false);
      }
    };

    fetchData();
  }, [url]);

  return { data, loading, error };
}

export default usePreFetch;
```

- **Usage**:

  ```
  import React from 'react';
  import usePreFetch from './usePreFetch';

  function App() {
    const { data, loading, error } = usePreFetch('/api/data');

    if (loading) return <div>Loading...</div>;
    if (error) return <div>Error: {error.message}</div>;

    return <div>{/* Render your data here */}</div>;
  }

  export default App;
  ```

These hooks and libraries streamline data fetching and caching in React, allowing you to optimize performance and improve the user experience.

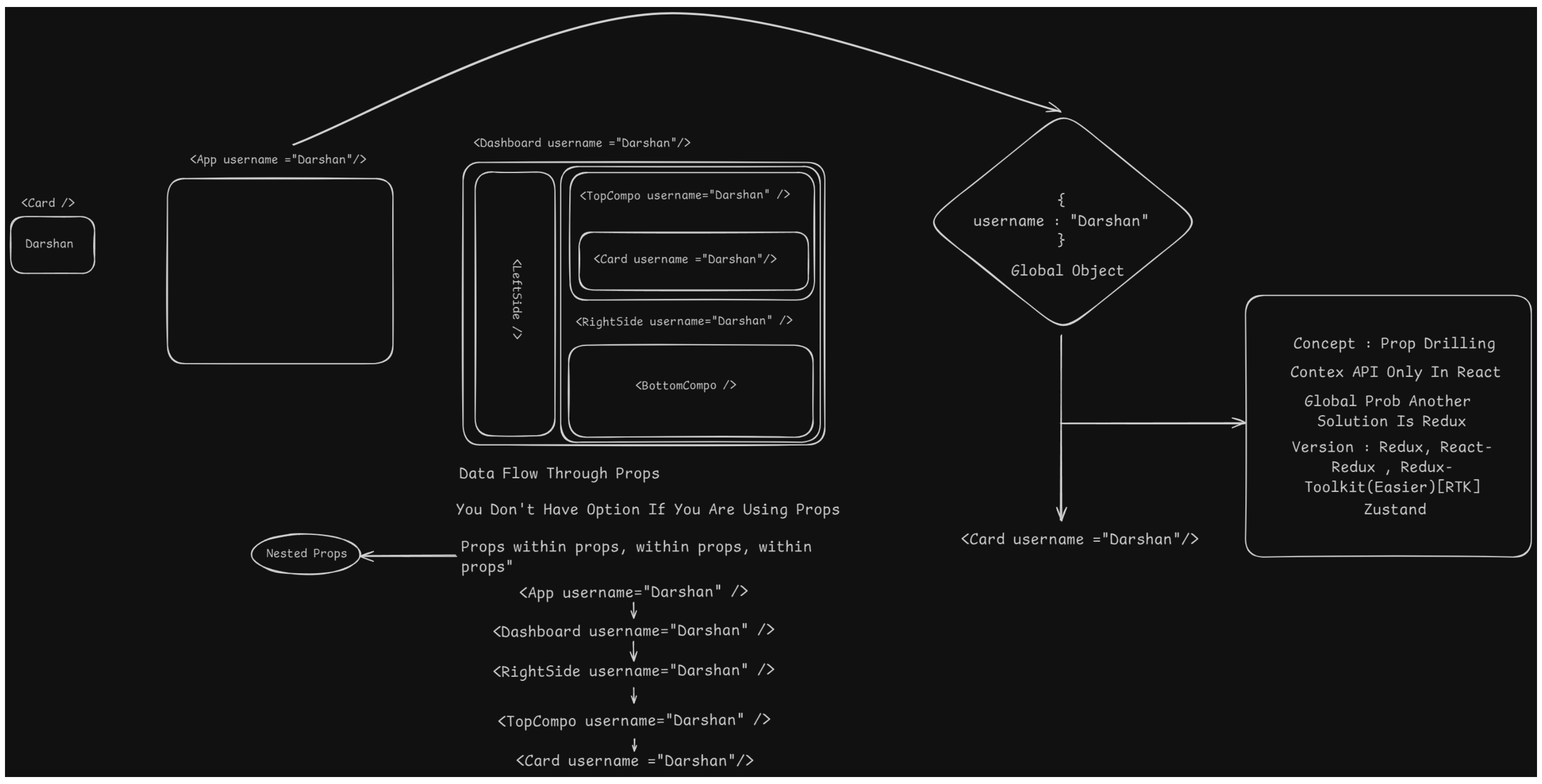
This approach ensures that data is available when needed, reducing load times and enhancing overall performance.

In React Router, **Direct API Call Through Loader and Action** optimizes data fetching by triggering API calls before the actual component mounts. This means:

- **Optimization**: When a user hovers over or interacts with a link, the loader or action starts fetching data in the background, even before the user clicks the link. This way, the data is ready by the time the component renders, reducing the wait time and improving the user experience.
- **Comparison to `useEffect`**: Unlike `useEffect`, which runs after the component mounts, loaders and actions fetch data earlier in the process. This early fetching leads to faster page load times and smoother transitions.

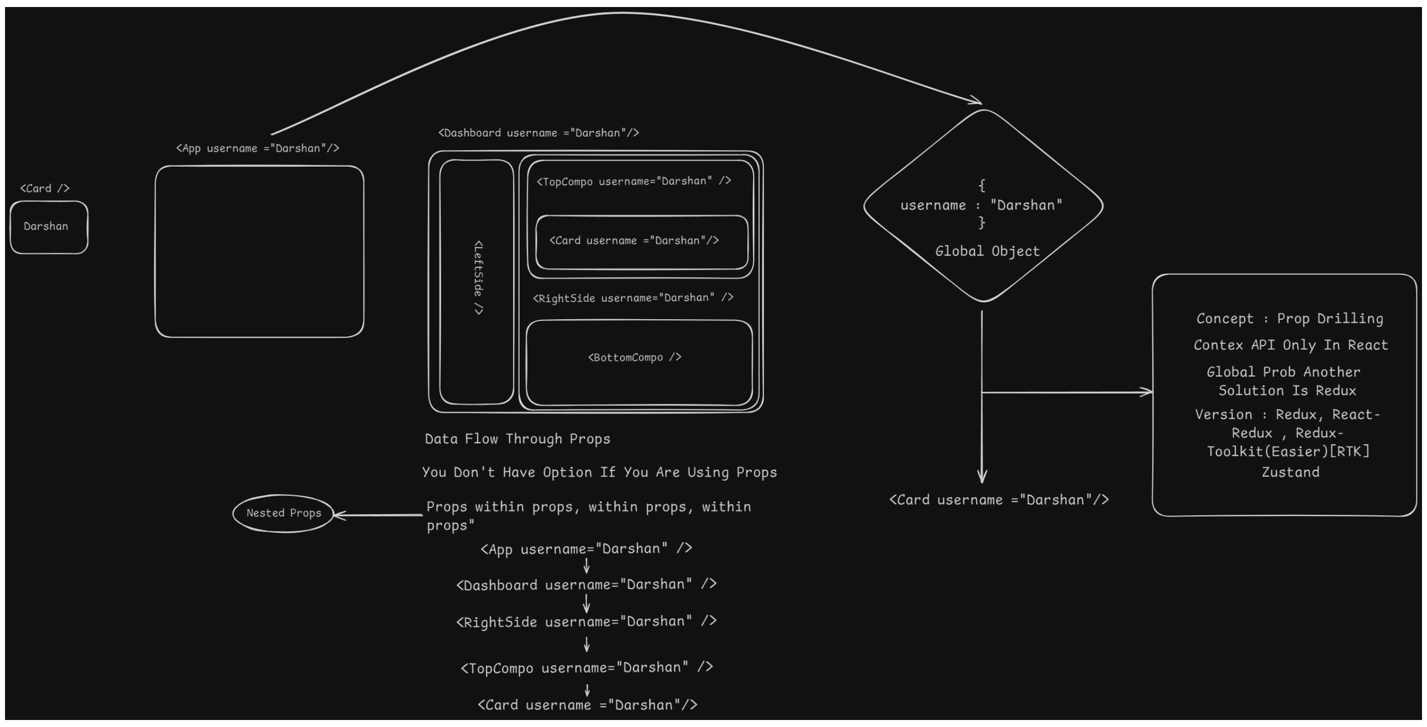
In short, loaders and actions optimize the process by fetching data ahead of time, making the application feel more responsive and efficient.

Context API ⇒ State Management



<Card />
Darshan
<App username ="Darshan"/>
<Dashboard username ="Darshan"/>
<LeftSide />
<TopCompo username="Darshan" />
<Card username ="Darshan"/>
<RightSide username="Darshan" />
<BottomCompo />
{ username : "Darshan" }
Global Object
Concept : Prop Drilling
Contex API Only In React
Global Prob Another Solution Is Redux
Version : Redux, React-Redux , Redux-Toolkit(Easier)[RTK] Zustand
<Card username ="Darshan"/>
Data Flow Through Props
You Don't Have Option If You Are Using Props
Nested Props
Props within props, within props, within props"
<App username="Darshan" />
<Dashboard username="Darshan" />
<RightSide username="Darshan" />
<TopCompo username="Darshan" />
<Card username ="Darshan"/>

Link : Excelidraw.com



<Card />
Darshan
<App username ="Darshan"/>
<Dashboard username ="Darshan"/>
<LeftSide />
<TopCompo username="Darshan" />
<Card username ="Darshan"/>
<RightSide username="Darshan" />
<BottomCompo />
{ username : "Darshan" }
Global Object
Concept : Prop Drilling
Contex API Only In React
Global Prob Another Solution Is Redux
Version : Redux, React-Redux , Redux-Toolkit(Easier)[RTK] Zustand
<Card username ="Darshan"/>
Data Flow Through Props
You Don't Have Option If You Are Using Props
Nested Props
Props within props, within props, within props"
<App username="Darshan" />
<Dashboard username="Darshan" />
<RightSide username="Darshan" />
<TopCompo username="Darshan" />
<Card username ="Darshan"/>

1. Context Creation:

```
import React from "react";

const UserContext = React.createContext();

export default UserContext;
```

- `React.createContext()` is used to create a new Context object called `UserContext`.
- This `UserContext` will hold and manage the shared state that can be accessed by multiple components within the app.

2. Provider Component:

```
<UserContext.Provider>
    <Login />
    <User />
    <Logout />
    <Card>
        <Data />
    </Card>
</UserContext.Provider>
```

- `Provider`: `UserContext.Provider` is a special component that comes with every context object created by `React.createContext()`. The `Provider` component is used to wrap the components that need access to the context's values.
- The `Provider` takes a `value` prop, which represents the data that will be shared among all the child components wrapped by this `Provider`.

In the provided example:

- Components like `<Login />`, `<User />`, `<Logout />`, `<Card>`, and `<Data />` are all wrapped within the `<UserContext.Provider>`.
- This means that all these components will have access to the values provided by `UserContext`.

3. Accessing Context in Components:

- Components wrapped within the `Provider` can access the context data directly using the `useContext` hook in functional components, or the `Context.Consumer` in class components.

For example:

```
import React, { useContext } from 'react';
import UserContext from './UserContext';

function User() {
    const user = useContext(UserContext); // Access context value here
    return <div>Welcome, {user.name}!</div>;
}
```

- Here, the `User` component accesses the shared state via `useContext(UserContext)`. It retrieves the value provided by the `UserContext.Provider` higher up in the component tree.

Summary:

- **Context** provides a way to pass data through the component tree without having to pass props manually at every level.
- The **Provider** component wraps all components that need access to the shared state and provides the data to them.
- This approach helps avoid "prop drilling" and keeps state management more centralized and maintainable.

By using context, you ensure that components can access the necessary state directly from the context, making your code cleaner and easier to manage, especially in larger applications.

```
// Andar Ke Components Directly UserContext Ke Through Sari States Ka Acess Le Sakte He
// This Is Provider
<UserContext>
<Login />
<User />
<Logout />
<Card>
    <Data />
</Card>
</UserContext>
```

Breakdown of the Code:

1. **UserContext.js**

   ```
   import React from 'react';

   // Create a Context object
   const UserContext = React.createContext();
   ```

```
// Export the context object so it can be used in other components
export default UserContext;
```

- This file defines a new context called `UserContext`.
- `React.createContext()` is called to create a context object. This object contains a `Provider` and a `Consumer` component. The `Provider` will be used to wrap components and provide them with the state, while the `Consumer` can be used to consume the state in any component that needs it (though using the `useContext` hook is more common now).
- `UserContext` is exported so it can be used in other parts of the application.

2. **UserContextProvider.jsx**

```
import React from "react";
import UserContext from "./UserContext";

const UserContextProvider = ({ children }) => {
    const [user, setUser] = React.useState(null);

    return (
        <UserContext.Provider value={{ user, setUser }}>
            {children}
        </UserContext.Provider>
    );
}

export default UserContextProvider;
```

- This file defines a `UserContextProvider` component, which is a React component that wraps its children with a `UserContext.Provider`.
- `useState(null)` is used to create a `user` state that starts as `null` and a `setUser` function to update this state. This state and the updater function are what we want to share with any component that needs to know about the user.
- `<UserContext.Provider value={{ user, setUser }}>` is where the magic happens. It uses the `Provider` component from `UserContext` and passes an object `{ user, setUser }` as its value prop. This means that any component inside `UserContextProvider` can access the `user` state and the `setUser` function.
- `{children}` is used to render any child components that are wrapped inside `UserContextProvider`.

How to Use `UserContextProvider` in Your Application:

To make this setup useful, you would wrap parts of your application that need access to the `user` state with `UserContextProvider`. Here's how you might do it in your main application file, such as `App.js`:

```
import React from "react";
import ReactDOM from "react-dom";
import UserContextProvider from "./UserContextProvider";
import SomeComponent from "./SomeComponent";

function App() {
    return (
        <UserContextProvider>
            <SomeComponent />
        </UserContextProvider>
    );
}

export default App;
```

In this example:

- `SomeComponent` (and any of its children) can access the `user` state and the `setUser` function by using the `useContext` hook from React.

Here's a quick example of how a component can consume the context:

```
import React, { useContext } from "react";
import UserContext from "./UserContext";

function SomeComponent() {
    const { user, setUser } = useContext(UserContext);

    return (
        <div>
            <h1>{user ? `Hello, ${user.name}` : "Hello, Guest"}</h1>
            <button onClick={() => setUser({ name: "John Doe" })}>
                Set User
            </button>
        </div>
    );
}

export default SomeComponent;
```

Summary:

- **`UserContext.js`**: Defines the context to manage user state.
- **`UserContextProvider.jsx`**: Provides the `user` state and `setUser` function to all child components using the `UserContext.Provider`.
- This setup allows any component within the `UserContextProvider` to easily access and update the user state without passing props down through multiple levels of components. This makes managing and sharing state across different parts of your application much more efficient and clean.

In React, a `Provider` component is a part of the Context API that allows you to supply or "provide" data to components that are nested inside it. This code example shows how to set up a context provider to manage and share user-related state across your application.

Explanation of the Code Provider:

```
import React from "react";
import UserContext from "./UserContext";

const UserContextProvider = ({ children }) => {
    const [user, setUser] = React.useState(null);

    return (
        <UserContext.Provider value={{ user, setUser }}>
            {children}
        </UserContext.Provider>
    );
}

export default UserContextProvider;
```

Key Components of This Code:

1. **Creating the Provider Component (`UserContextProvider`):**
    - This is a functional component named `UserContextProvider`.

- It takes a `children` prop, which represents any nested components inside this provider. These nested components will be able to access the context.

2. **Managing State:**
   - Inside the `UserContextProvider`, the state for the user is managed using the `useState` hook:

     ```
     const [user, setUser] = React.useState(null);
     ```

   - `user` is the state variable that holds the current user information (initially set to `null`).
   - `setUser` is the function used to update the `user` state.

3. **Using `UserContext.Provider`:**
   - The `UserContextProvider` component uses the `UserContext.Provider` to pass down the `user` state and the `setUser` function to all components that are wrapped by this provider.
   - The `value` prop of the `Provider` holds the data that needs to be shared:

     ```
     <UserContext.Provider value={{ user, setUser }}>
     ```

   - By passing `{ user, setUser }` as the value, both the state and the function to update the state are made accessible to any component that consumes this context.

4. **Rendering Children:**
   - `{children}` inside the provider renders whatever components are nested within `UserContextProvider`. This makes those components able to access and use the provided context.

How to Use `UserContextProvider`:

To make use of this provider, you wrap parts of your application that need access to the user state with the `UserContextProvider`. For example:

```
import React from "react";
import UserContextProvider from "./UserContextProvider";
import SomeComponent from "./SomeComponent";

function App() {
    return (
        <UserContextProvider>
            <SomeComponent />
        </UserContextProvider>
    );
}

export default App;
```

- Here, `SomeComponent` and any other components inside `UserContextProvider` can access the `user` state and the `setUser` function using `useContext(UserContext)`.

Summary:

- The `UserContextProvider` component wraps around components that need access to shared `user` state.
- It uses the `UserContext.Provider` to pass the current state and a state updater function down the component tree.
- This pattern is useful for managing global state (like authentication, themes, etc.) in a more scalable and maintainable way, avoiding the need for prop drilling.

Mini Context API

The given code illustrates how to implement a user authentication context using the React Context API. This allows the `Login` and `Profile` components to share the user state without passing props down through each level of the component tree. Let's break down the code step-by-step:

1. Creating the UserContext

File: `./Context/UserContext.js`

```
import React from 'react';

const UserContext = React.createContext();

export default UserContext;
```

- **What it does**: This code creates a new Context object named `UserContext` using `React.createContext()`. This object will hold the user state and provide a way to share this state across different components.
- **Why it's needed**: `UserContext` will be used to provide and consume the user-related data throughout the component tree. This allows for easier state management without prop drilling.

2. Creating the UserContext Provider

File: `./Context/UserContext.jsx`

```
import React from "react";
import UserContext from "./UserContext";

const UserContextProvider = ({ children }) => {
    const [user, setUser] = React.useState(null);

    return (
        <UserContext.Provider value={{ user, setUser }}>
            {children}
        </UserContext.Provider>
    );
};

export default UserContextProvider;
```

- **What it does**: This component, `UserContextProvider`, acts as a wrapper for any part of the application that needs access to the `UserContext`. It initializes a state (`user`) and a function to update it (`setUser`), which are both shared via the `UserContext.Provider`.
- **Why it's needed**: This provider component allows any child component to access and modify the `user` state. The `value` prop of the `UserContext.Provider` is an object that contains the `user` state and the `setUser` function, making them available to any component that consumes the context.

3. Using the Provider in the App Component

File: `App.jsx`

```
import './App.css';
import Login from './components/Login';
import Profile from './components/Profile';
import UserContextProvider from './context/UserContextProvider';

function App() {
    return (
        <UserContextProvider>
            <h1>React with Chai and share is important</h1>
            <Login />
            <Profile />
        </UserContextProvider>
    );
}

export default App;
```

- **What it does**: The `App` component uses the `UserContextProvider` to wrap its child components (`Login` and `Profile`). This means that both `Login` and `Profile` components (and any other nested components) can access the `UserContext`.
- **Why it's needed**: By wrapping the `Login` and `Profile` components with `UserContextProvider`, it enables these components to share the same `user` state, maintaining consistent user information throughout the app.

4. Implementing the Login Component

File: `Login.jsx`

```
import React, { useState, useContext } from 'react';
import UserContext from '../context/UserContext';

function Login() {
    const [username, setUsername] = useState('');
    const [password, setPassword] = useState('');

    const { setUser } = useContext(UserContext);

    const handleSubmit = (e) => {
        e.preventDefault();
        setUser({ username, password });
    };

    return (
        <div>
            <h2>Login</h2>
            <input
                type='text'
                value={username}
                onChange={(e) => setUsername(e.target.value)}
                placeholder='username'
            />
            {" "}
            <input
                type='password'
                value={password}
                onChange={(e) => setPassword(e.target.value)}
                placeholder='password'
            />
            <button onClick={handleSubmit}>Submit</button>
        </div>
    );
}

export default Login;
```

- **What it does**: This component provides a simple login form with two input fields for the username and password. Upon submitting the form, it sets the `user` state with the entered username and password using the `setUser` function from `UserContext`.
- **Why it's needed**: The `Login` component interacts with the `UserContext` by setting the user information. This interaction demonstrates how to update the context's state, which can then be used by other components.

5. Implementing the Profile Component

File: `Profile.jsx`

```
import React, { useContext } from 'react';
import UserContext from '../context/UserContext';

function Profile() {
    const { user } = useContext(UserContext);

    if (!user) return <div>please login</div>;

    return <div>Welcome {user.username}</div>;
}

export default Profile;
```

- **What it does**: This component displays a welcome message with the username of the logged-in user. If no user is logged in, it prompts the user to log in.
- **Why it's needed**: The `Profile` component shows how to consume context data using the `useContext` hook, allowing it to access the current `user` state provided by `UserContext`.

Summary of How Context API Works in This Example

1. **UserContext Creation**: A context (`UserContext`) is created to hold the user state.
2. **Provider Setup**: `UserContextProvider` component initializes state and provides it to any component wrapped inside it.
3. **Using Provider in App**: The `App` component wraps its children with `UserContextProvider`, allowing the `Login` and `Profile` components to share the user state.
4. **Consuming Context in Login**: The `Login` component uses the `useContext` hook to update the `user` state.
5. **Consuming Context in Profile**: The `Profile` component uses the `useContext` hook to read the `user` state and conditionally render content based on whether a user is logged in.

This flow allows different parts of the app to easily share and update user-related data without prop drilling.

Folder Structure

```
08miniContext/
├── node_modules/
├── public/
│   ├── index.html
│   └── ...
├── src/
│   ├── assets/
│   ├── components/
│   │   ├── Login.jsx
│   │   └── Profile.jsx
│   ├── context/
│   │   ├── UserContext.js
│   │   └── UserContextProvider.jsx
│   ├── App.css
│   ├── App.jsx
│   ├── index.css
│   ├── main.jsx
│   └── ...
├── .eslintrc.cjs
├── .gitignore
├── eslint.config.js
├── package-lock.json
├── package.json
├── README.md
└── vite.config.js
```

Breakdown of the Folder Structure

- **08miniContext/**: Root directory of the React project.
  - **node_modules/**: Contains all the dependencies and packages installed via npm.
  - **public/**: This directory holds static files, including the main `index.html` file that serves as the entry point for the React app.
    - **index.html**: Main HTML file that loads the React application.
  - **src/**: This is the main source directory where all the application code resides.
    - **assets/**: (Optional) This folder can store images, fonts, or other static assets used in the project.
    - **components/**: Contains the React components of the application.
      - **Login.jsx**: Component for user login, allowing the user to enter a username and password.
      - **Profile.jsx**: Component for displaying the user's profile or a login prompt if not authenticated.
    - **context/**: Contains files related to context creation and management.
      - **UserContext.js**: Defines and exports the `UserContext` object, which holds user-related data.
      - **UserContextProvider.jsx**: Defines and exports the `UserContextProvider` component, which wraps the application and provides the `UserContext`.
    - **App.css**: Styling for the main `App` component.
    - **App.jsx**: Main component of the application, sets up the application with the `UserContextProvider`.
    - **index.css**: Global CSS for the application.
    - **main.jsx**: Entry point for the React application; renders the `App` component into the DOM.
  - **.eslintrc.cjs**: ESLint configuration file to enforce coding standards and style guidelines.
  - **.gitignore**: Specifies files and directories that Git should ignore.
  - **eslint.config.js**: Additional configuration for ESLint.
  - **package-lock.json**: Auto-generated file that describes the exact dependency tree installed, used to ensure consistent installs across different environments.
  - **package.json**: Lists dependencies, scripts, and metadata for the project.
  - **README.md**: Provides information about the project, how to set it up, and how to use it.
  - **vite.config.js**: Configuration file for Vite, the build tool used in this project.

Summary

- **Context API** is implemented to manage the user state (`user` and `setUser`).
- **Components** are organized into a `components` folder for easy access and separation of concerns.
- **Context files** are grouped into a `context` folder, highlighting the modular approach for state management.
- **Global styling** and **entry points** are clearly defined, making the application easy to navigate and extend.

This organized folder structure helps keep the code modular, readable, and maintainable, facilitating future updates or scaling of the project.

We Can Give Variables And Method(Function) In UseContext

```
import React from "react";
import UserContext from "./UserContext";

const UserContextProvider = ({ children }) => {
```

```
// We Can Give Variables And Method(Function) In UseContext
    const [variable, Function] = React.useState(null);

    return (
        <UserContext.Provider value={{ user, setUser }}>
            {children}
        </UserContext.Provider>
    );
};

export default UserContextProvider;
```

Custom Hook Itself Is A Function

2nd Syntax Is Here For UseContext

Theme Switcher

`useTheme` has access to `ThemeContext`, which can toggle between dark and light themes. By default, the theme is set to light.

The code you've provided implements a theme switcher using the React Context API. This allows for managing the theme state (light or dark) throughout the entire application without passing props down manually at every level. Let's break down how this works through the relevant files.

1. Context Setup (`contexts/theme.js`)

```
import { createContext, useContext } from "react";

// Create the ThemeContext with default values
export const ThemeContext = createContext({
    themeMode: "light",
    darkTheme: () => {},
    lightTheme: () => {},
});

// Create a ThemeProvider using the ThemeContext
export const ThemeProvider = ThemeContext.Provider;

// Custom hook to use the ThemeContext
export default function useTheme() {
    return useContext(ThemeContext);
}
```

- `createContext`: This creates a context object with default values for `themeMode`, `darkTheme`, and `lightTheme` functions. These values can be provided by the nearest `ThemeProvider` in the component tree.
- `ThemeProvider`: This is a provider component that will supply the context to its children.
- `useTheme`: This custom hook allows easy access to the context values within any component.

2. Theme Toggle Button (`components/ThemeBtn.jsx`)

```
import React from 'react';
import useTheme from '../contexts/theme';

export default function ThemeBtn() {
    const { themeMode, lightTheme, darkTheme } = useTheme();
    const onChangeBtn = (e) => {
        const darkModeStatus = e.currentTarget.checked;
        if (darkModeStatus) {
            darkTheme();
        } else {
            lightTheme();
        }
    };

    return (
        <label className="relative inline-flex items-center cursor-pointer">
            <input
                type="checkbox"
                className="sr-only peer"
                onChange={onChangeBtn}
                checked={themeMode === "dark"}
            />
            <div className="w-11 h-6 bg-gray-200 peer-focus:outline-none peer-focus:ring-4 peer-focus:ring-blue-300 dark:peer-focus:ring-blue-800 rounded-full peer dark:bg-gray-700 peer-checked:after:translate-x-full peer-checked:after:border-white after:content-[''] after:absolute after:top-[2px] after:left-[2px] after:bg-white after:border-gray-300 after:border after:rounded-full after:h-5 after:w-5 after:transition-all dark:border-gray-600 peer-checked:bg-blue-600"></div>
            <span className="ml-3 text-sm font-medium text-gray-900">Toggle Theme</span>
        </label>
    );
}
```

- `useTheme`: The custom hook is used to access `themeMode`, `lightTheme`, and `darkTheme` functions from the context.
- **Toggle Logic**: Based on the checkbox state (`checked`), it triggers `lightTheme` or `darkTheme` to switch the theme.

3. Application Root (`App.jsx`)

```
import { useEffect, useState } from 'react';
import './App.css';
import { ThemeProvider } from './contexts/theme';
import ThemeBtn from './components/ThemeBtn';
import Card from './components/Card';

function App() {
    const [themeMode, setThemeMode] = useState("light");

    const lightTheme = () => {
        setThemeMode("light");
    };

    const darkTheme = () => {
        setThemeMode("dark");
    };

    // Effect to change the actual theme of the document
    useEffect(() => {
        document.querySelector('html').classList.remove("light", "dark");
        document.querySelector('html').classList.add(themeMode);
    }, [themeMode]);
    return (
        <ThemeProvider value={{ themeMode, lightTheme, darkTheme }}>
            <div className="flex flex-wrap min-h-screen items-center">
                <div className="w-full">
                    <div className="w-full max-w-sm mx-auto flex justify-end mb-4">
                        <ThemeBtn />
                    </div>
                    <div className="w-full max-w-sm mx-auto">
                        <Card />
                    </div>
                </div>
            </div>
        </ThemeProvider>
    );
}

export default App;
```

- **State Management**: `themeMode` state is managed locally within the `App` component and modified using the `lightTheme` and `darkTheme` functions.
- **Effect Hook**: The `useEffect` hook ensures that the HTML document's class is updated whenever `themeMode` changes. This is necessary for Tailwind's dark mode to work.
- **Provider**: `ThemeProvider` is used to wrap the component tree, providing `themeMode`, `lightTheme`, and `darkTheme` to all children components.

4. Tailwind CSS Configuration (`tailwind.config.js`)

```
/** @type {import('tailwindcss').Config} */
export default {
  content: [
    "./index.html",
    "./src/**/*.{js,ts,jsx,tsx}",
  ],
  darkMode: "class", // Enables class-based dark mode
  theme: {
    extend: {},
  },
  plugins: [],
}
```

- **`darkMode: "class"`**: This tells Tailwind to use class-based dark mode switching. When the `dark` class is added to the HTML element, dark mode styles are applied.

How the Theme Switcher Works

1. **Context Setup**: The `ThemeContext` provides a global state for theme management.
2. **Custom Hook**: The `useTheme` hook simplifies access to the theme context within components.
3. **Theme Toggle**: The `ThemeBtn` component uses the theme context to toggle between light and dark modes.
4. **Theme Effect**: The `useEffect` hook in `App` updates the HTML element's class based on the `themeMode` state, effectively toggling between light and dark themes by adding/removing the appropriate classes.
5. **Tailwind Integration**: Tailwind's configuration is set up to apply dark mode styles when the `dark` class is present on the HTML element.

Summary

This implementation is a clean and scalable approach to theme switching in React applications, leveraging the Context API for global state management and integrating with Tailwind CSS for styling. By using context, the theme state is efficiently shared across components, and Tailwind's class-based dark mode makes it easy to apply the appropriate styles.

Context API Directly Available In React Through useContext

Todo Context Local Storage

1. Context Setup with `TodoContext`

The `TodoContext` is created using the `createContext` function from React. It provides a way to pass data through the component tree without having to pass props down manually at every level.

```
// context/TodoContext.js
import { createContext, useContext } from "react";

export const TodoContext = createContext({
    todos: [
        {
            id: 1,
            todo: " Todo msg",
            completed: false,
        }
    ],
    addTodo: (todo) => {},
    updateTodo: (id, todo) => {},
    deleteTodo: (id) => {},
    toggleComplete: (id) => {}
});

export const useTodo = () => {
    return useContext(TodoContext);
};

export const TodoProvider = TodoContext.Provider;
```

- `TodoContext`: This context holds the default values for the to-do list (`todos`) and the functions to manipulate it (`addTodo`, `updateTodo`, `deleteTodo`, `toggleComplete`).
- `useTodo`: A custom hook that provides a convenient way to access the context values.
- `TodoProvider`: This is a provider component used to wrap around any part of the application that needs access to the to-do data.

2. Index File for Contexts

To make importing easier and more organized, the contexts are exported from an `index.js` file.

```
// context/index.js
export { TodoContext, TodoProvider, useTodo } from "./TodoContext";
```

3. Application Setup (`App.jsx`)

The main application component uses the `TodoProvider` to provide the context to all components within it. The `App` component manages the to-do state and handles local storage integration.

```
// App.jsx
import { useState, useEffect } from 'react';
import { TodoProvider } from './contexts';
import './App.css';
import TodoForm from './components/TodoForm';
import TodoItem from './components/TodoItem';
```

```
function App() {
  const [todos, setTodos] = useState([]);

  const addTodo = (todo) => {
    setTodos((prev) => [{ id: Date.now(), ...todo }, ...prev]);
  };

  const updateTodo = (id, todo) => {
    setTodos((prev) => prev.map((prevTodo) => (prevTodo.id === id ? todo : prevTodo)));
  };

  const deleteTodo = (id) => {
    setTodos((prev) => prev.filter((todo) => todo.id !== id));
  };

  const toggleComplete = (id) => {
    setTodos((prev) =>
      prev.map((prevTodo) =>
        prevTodo.id === id ? { ...prevTodo, completed: !prevTodo.completed } : prevTodo
      )
    );
  };

  useEffect(() => {
    // Load todos from local storage when the component mounts
    const todos = JSON.parse(localStorage.getItem("todos"));
    if (todos && todos.length > 0) {
      setTodos(todos);
    }
  }, []);

  useEffect(() => {
    // Save todos to local storage whenever the todos state changes
    localStorage.setItem("todos", JSON.stringify(todos));
  }, [todos]);

  return (
    <TodoProvider value={{ todos, addTodo, updateTodo, deleteTodo, toggleComplete }}>
      <div className="bg-[#172842] min-h-screen py-8">
        <div className="w-full max-w-2xl mx-auto shadow-md rounded-lg px-4 py-3 text-white">
          <h1 className="text-2xl font-bold text-center mb-8 mt-2">Manage Your Todos</h1>
          <div className="mb-4">
            <TodoForm />
          </div>
          <div className="flex flex-wrap gap-y-3">
            {todos.map((todo) => (
              <div key={todo.id} className="w-full">
                <TodoItem todo={todo} />
              </div>
            ))}
          </div>
        </div>
      </div>
    </TodoProvider>
  );
}
```

```
export default App;
```

- **State Management**: The state for the to-do list is managed using React's `useState` hook.
- **CRUD Operations**: Functions for adding, updating, deleting, and toggling the completion status of to-dos are defined and passed down through the context.
- **Local Storage Integration**: The `useEffect` hooks are used to load to-dos from local storage when the component mounts and to save to-dos to local storage whenever the to-do list changes. This ensures that to-do items persist even after a page reload.

4. Todo Form and Item Components

These components use the context to interact with the to-do list.

TodoForm.jsx

```
// components/TodoForm.jsx
import React, { useState } from 'react';
import { useTodo } from '../contexts/TodoContext';

function TodoForm() {
    const [todo, setTodo] = useState("");
    const { addTodo } = useTodo();

    const add = (e) => {
      e.preventDefault();
      if (!todo) return;
      addTodo({ todo, completed: false });
      setTodo("");
    };

  return (
      <form onSubmit={add} className="flex">
          <input
              type="text"
              placeholder="Write Todo..."
              className="w-full border border-black/10 rounded-l-lg px-3 outline-none duratio
n-150 bg-white/20 py-1.5"
              value={todo}
              onChange={(e) => setTodo(e.target.value)}
          />
          <button type="submit" className="rounded-r-lg px-3 py-1 bg-green-600 text-white shr
ink-0">
              Add
          </button>
      </form>
  );
}

export default TodoForm;
```

TodoItem.jsx

```
// components/TodoItem.jsx
import React, { useState } from 'react';
import { useTodo } from '../contexts/TodoContext';
```

```
function TodoItem({ todo }) {
  const [isTodoEditable, setIsTodoEditable] = useState(false);
  const [todoMsg, setTodoMsg] = useState(todo.todo);
  const { updateTodo, deleteTodo, toggleComplete } = useTodo();

  const editTodo = () => {
    updateTodo(todo.id, { ...todo, todo: todoMsg });
    setIsTodoEditable(false);
  };

  const toggleCompleted = () => {
    toggleComplete(todo.id);
  };

  return (
      <div className={`flex border border-black/10 rounded-lg px-3 py-1.5 gap-x-3 shadow-sm s
hadow-white/50 duration-300 text-black ${todo.completed ? "bg-[#c6e9a7]" : "bg-[#ccbed7]"}`}>
          <input
              type="checkbox"
              className="cursor-pointer"
              checked={todo.completed}
              onChange={toggleCompleted}
          />
          <input
              type="text"
              className={`border outline-none w-full bg-transparent rounded-lg ${isTodoEditab
le ? "border-black/10 px-2" : "border-transparent"} ${todo.completed ? "line-through" : ""}`}
              value={todoMsg}
              onChange={(e) => setTodoMsg(e.target.value)}
              readOnly={!isTodoEditable}
          />
          <button
              className="inline-flex w-8 h-8 rounded-lg text-sm border border-black/10 justif
y-center items-center bg-gray-50 hover:bg-gray-100 shrink-0 disabled:opacity-50"
              onClick={() => {
                  if (todo.completed) return;
                  if (isTodoEditable) {
                      editTodo();
                  } else setIsTodoEditable((prev) => !prev);
              }}
              disabled={todo.completed}
          >
              {isTodoEditable ? "📁" : "✏️"}
          </button>
          <button
              className="inline-flex w-8 h-8 rounded-lg text-sm border border-black/10 justif
y-center items-center bg-gray-50 hover:bg-gray-100 shrink-0"
              onClick={() => deleteTodo(todo.id)}
          >
              ❌
          </button>
      </div>
  );
}

export default TodoItem;
```

Summary

- **Context API**: It provides a way to manage the state of to-do items globally, making it accessible across different components without prop drilling.
- **Local Storage**: It ensures that the state persists across page reloads by saving the to-do list to local storage whenever it changes, and loading it back into state when the app initializes.
- **Component Interaction**: Components like `TodoForm` and `TodoItem` interact with the global state using the custom `useTodo` hook, which provides access to the state and functions defined in the `TodoContext`.

This structure provides a robust and scalable way to manage state and persist data in a React application using Context API and local storage.

1. Context Setup:

The `TodoContext` is created using React's `createContext()` function. It defines the default structure for the context, including an array of todos and functions for manipulating these todos.

```
// context/TodoContext.js
import { createContext, useContext } from "react";

export const TodoContext = createContext({
    todos: [
        {
            id: 1,
            todo: "Todo msg",
            completed: false,
        }
    ],
    addTodo: (todo) => {},
    updateTodo: (id, todo) => {},
    deleteTodo: (id) => {},
    toggleComplete: (id) => {}
});

export const useTodo = () => {
    return useContext(TodoContext);
}

export const TodoProvider = TodoContext.Provider;
```

2. Provider Usage in App Component:

The `App` component uses the `TodoProvider` to wrap the application's UI components. This makes the todos and their manipulation functions available throughout the component tree.

```
// App.jsx
import { useState, useEffect } from 'react';
import { TodoProvider } from './contexts';
import './App.css';
import TodoForm from './components/TodoForm';
import TodoItem from './components/TodoItem';

function App() {
  const [todos, setTodos] = useState([]);

  const addTodo = (todo) => {
    setTodos((prev) => [{ id: Date.now(), ...todo }, ...prev]);
  };
  const updateTodo = (id, todo) => {
    setTodos((prev) => prev.map((prevTodo) => (prevTodo.id === id ? todo : prevTodo)));
  };

  const deleteTodo = (id) => {
    setTodos((prev) => prev.filter((todo) => todo.id !== id));
  };

  const toggleComplete = (id) => {
    setTodos((prev) =>
      prev.map((prevTodo) =>
        prevTodo.id === id ? { ...prevTodo, completed: !prevTodo.completed } : prevTodo
      )
    );
  };

  useEffect(() => {
    const todos = JSON.parse(localStorage.getItem("todos"));
    if (todos && todos.length > 0) {
      setTodos(todos);
    }
  }, []);

  useEffect(() => {
    localStorage.setItem("todos", JSON.stringify(todos));
  }, [todos]);

  return (
    <TodoProvider value={{ todos, addTodo, updateTodo, deleteTodo, toggleComplete }}>
      <div className="bg-[#172842] min-h-screen py-8">
        <div className="w-full max-w-2xl mx-auto shadow-md rounded-lg px-4 py-3 text-white">
          <h1 className="text-2xl font-bold text-center mb-8 mt-2">Manage Your Todos</h1>
          <div className="mb-4">
            <TodoForm />
          </div>
          <div className="flex flex-wrap gap-y-3">
            {todos.map((todo) => (
              <div key={todo.id} className='w-full'>
                <TodoItem todo={todo} />
              </div>
            ))}
          </div>
        </div>
      </div>
    </TodoProvider>
  );
}

export default App;
```

3. Functionality Explanation:

a. Adding a Todo:

The `addTodo` function is used to add a new todo item. It updates the `todos` state by adding a new todo object at the beginning of the array.

```
const addTodo = (todo) => {
    setTodos((prev) => [{ id: Date.now(), ...todo }, ...prev]);
};
```

- **TodoForm Component**:
  The
  `TodoForm` component uses the `addTodo` function to add a new todo.

```
// components/TodoForm.jsx
import React, { useState } from 'react';
import { useTodo } from '../contexts/TodoContext';

function TodoForm() {
    const [todo, setTodo] = useState("");
    const { addTodo } = useTodo();

    const add = (e) => {
        e.preventDefault();
        if (!todo) return;
        addTodo({ todo, completed: false });
        setTodo("");
    }

    return (
        <form onSubmit={add} className="flex">
            <input
                type="text"
                placeholder="Write Todo..."
                className="w-full border border-black/10 rounded-l-lg px-3 outline-none durat
ion-150 bg-white/20 py-1.5"
                value={todo}
                onChange={(e) => setTodo(e.target.value)}
            />
            <button type="submit" className="rounded-r-lg px-3 py-1 bg-green-600 text-white s
hrink-0">
                Add
            </button>
        </form>
    );
}

export default TodoForm;
```

b. Updating a Todo:

The `updateTodo` function updates the text of a specific todo item. It finds the todo by its `id` and replaces it with the updated todo.

```
const updateTodo = (id, todo) => {
    setTodos((prev) => prev.map((prevTodo) => (prevTodo.id === id ? todo : prevTodo)));
};
```

- **TodoItem Component**:
  The
  `editTodo` function within `TodoItem` calls `updateTodo` to save the changes made to a todo item.

```
const editTodo = () => {
    updateTodo(todo.id, { ...todo, todo: todoMsg });
    setIsTodoEditable(false);
}
```

c. Deleting a Todo:

The `deleteTodo` function removes a todo item by filtering out the todo with the specified `id`.

```
const deleteTodo = (id) => {
    setTodos((prev) => prev.filter((todo) => todo.id !== id));
};
```

- **TodoItem Component**:
  The
  `deleteTodo` function is called when the user clicks the delete button.

```
<button onClick={() => deleteTodo(todo.id)}>❌</button>
```

d. Toggling Completion Status:

The `toggleComplete` function toggles the `completed` status of a todo item. It finds the todo by its `id` and flips its `completed` property.

```
const toggleComplete = (id) => {
    setTodos((prev) =>
        prev.map((prevTodo) =>
            prevTodo.id === id ? { ...prevTodo, completed: !prevTodo.completed } : prevTodo
        )
    );
};
```

- **TodoItem Component**:
  The
  `toggleCompleted` function is called when the user clicks the checkbox.

```
const toggleCompleted = () => {
    toggleComplete(todo.id);
}
```

4. Using Local Storage:

To persist the state between sessions, the todos are stored in local storage.

- **Retrieving from Local Storage**: On component mount, `useEffect` retrieves the `todos` from local storage.

```
useEffect(() => {
    const todos = JSON.parse(localStorage.getItem("todos"));
    if (todos && todos.length > 0) {
        setTodos(todos);
    }
}, []);
```

- **Saving to Local Storage**: Whenever the `todos` state changes, another `useEffect` stores the updated list in local storage.

```
useEffect(() => {
    localStorage.setItem("todos", JSON.stringify(todos));
}, [todos]);
```

Conclusion

The code provides a complete solution for managing todos using React Context API and local storage. It allows for creating, updating, deleting, and toggling the status of todos, ensuring that the data persists across page reloads. This approach is modular and scalable, making it easy to extend and maintain the application.

SetTodos

The `setTodos` function is a state updater function provided by the `useState` hook in React. It is used to update the state variable `todos` that holds the list of todo items in the application.

What Does `setTodos` Do?

`setTodos` is responsible for updating the `todos` state with a new array of todo items. Every time you call `setTodos` with a new value, React re-renders the component, updating the UI to reflect the current state of the `todos` array. This is crucial for managing dynamic data in React applications, as it allows the UI to stay in sync with the underlying data.

Where Is `setTodos` Defined?

`setTodos` is defined as part of the `useState` hook when initializing the `todos` state variable. The `useState` hook returns an array with two elements:

1. The current state (`todos` in this case).
2. The state updater function (`setTodos` in this case).

Here is the line of code where `setTodos` is defined:

```
const [todos, setTodos] = useState([]);
```

Explanation of `useState([])`

- `useState([])` initializes the `todos` state as an empty array.
- `todos` is the current state value that represents the list of todos.
- `setTodos` is the function used to update the `todos` state.

Example Usage of `setTodos`

In the `App` component, `setTodos` is used in several functions to update the state:

1. **Adding a Todo**: When a new todo is added, `setTodos` is used to add the new todo object to the start of the array.

   ```
   const addTodo = (todo) => {
       setTodos((prev) => [{ id: Date.now(), ...todo }, ...prev]);
   };
   ```

   - Here, `setTodos` updates the state by adding a new todo object with a unique `id` to the beginning of the existing array of todos.

2. **Updating a Todo**: When a todo is edited, `setTodos` updates the corresponding todo in the array.

   ```
   const updateTodo = (id, todo) => {
       setTodos((prev) => prev.map((prevTodo) => (prevTodo.id === id ? todo : prevTodo)));
   };
   ```

   - This uses `setTodos` to iterate over the existing array of todos. If a todo with the matching `id` is found, it is replaced with the updated todo object.

3. **Deleting a Todo**: To delete a todo, `setTodos` is used to filter out the todo with the specified `id`.

```
const deleteTodo = (id) => {
    setTodos((prev) => prev.filter((todo) => todo.id !== id));
};
```

- This filters the array of todos, removing the one with the specified `id`.

4. **Toggling Completion Status**: When the completion status of a todo is toggled, `setTodos` updates the `completed` property of the specified todo.

```
const toggleComplete = (id) => {
    setTodos((prev) =>
        prev.map((prevTodo) =>
            prevTodo.id === id ? { ...prevTodo, completed: !prevTodo.completed } : prevTod
o
        )
    );
};
```

- This updates the `completed` status of the todo with the matching `id`.

Summary

`setTodos` is a powerful function that is key to managing the state of the `todos` array in the application. It is defined by the `useState` hook and is used to update the state whenever changes to the list of todos are made, ensuring the UI stays consistent with the data. This use of state and state management is a fundamental concept in React, allowing for reactive and interactive UIs.

Object Passing ⇒ `{}`

In Loop ⇒ `{}` ⇒ You Have To Write Return Statement

In Loop ⇒ `()` ⇒ You Have Not To Write Return Statement ⇒ Auto Return

Redux-Toolkit

In Redux Toolkit, which simplifies the standard Redux workflow, the concepts of `store`, `reducer`, `useSelector`, and `useDispatch` play crucial roles in managing the state of a React application. Here's a detailed explanation of each concept, along with code snippets to illustrate how they are used.

1. Store

The **store** is the central place that holds the entire state of your application. In Redux, you create a store using the `configureStore` method provided by Redux Toolkit, which sets up the store with good defaults and automatically adds the Redux DevTools and middleware.

Example: Setting up the Store

```
import { configureStore } from '@reduxjs/toolkit';
import todosReducer from './todosSlice';

const store = configureStore({
  reducer: {
    todos: todosReducer, // Adding the todos reducer to the store
  },
});

export default store;
```

- Here, `configureStore` is called with an object that has a `reducer` property.

- The `todosReducer` is the slice reducer that manages the `todos` state. You can combine multiple reducers here for different slices of state.

2. Reducer

A **reducer** is a function that determines how the state should change in response to an action. It takes the current state and an action as arguments and returns the new state.

In Redux Toolkit, reducers are typically created using the `createSlice` function, which automatically generates action creators and action types.

Example: Creating a Reducer with `createSlice`

```
import { createSlice } from '@reduxjs/toolkit';

const todosSlice = createSlice({
  name: 'todos',
  initialState: [],
  reducers: {
    addTodo: (state, action) => {
      state.push(action.payload);
    },
    updateTodo: (state, action) => {
      const { id, todo } = action.payload;
      const existingTodo = state.find((t) => t.id === id);
      if (existingTodo) {
        existingTodo.todo = todo;
      }
    },
    deleteTodo: (state, action) => {
      const id = action.payload;
      return state.filter((todo) => todo.id !== id);
    },
    toggleComplete: (state, action) => {
      const id = action.payload;
      const todo = state.find((t) => t.id === id);
      if (todo) {
        todo.completed = !todo.completed;
      }
    },
  },
});

export const { addTodo, updateTodo, deleteTodo, toggleComplete } = todosSlice.actions;
export default todosSlice.reducer;
```

- `createSlice` helps to define a slice of the state and automatically generates action creators and action types based on the reducers you define.
- Here, we define four actions: `addTodo`, `updateTodo`, `deleteTodo`, and `toggleComplete`.

3. `useSelector`

The `useSelector` hook is used to extract data from the Redux store state. It's analogous to `mapStateToProps` in class components and allows functional components to read the state.

Example: Using `useSelector` to Select Todos

```
import React from 'react';
import { useSelector } from 'react-redux';

const TodoList = () => {
  const todos = useSelector((state) => state.todos);

  return (
    <ul>
      {todos.map((todo) => (
        <li key={todo.id}>
          {todo.todo} - {todo.completed ? 'Completed' : 'Not Completed'}
        </li>
      ))}
    </ul>
  );
};

export default TodoList;
```

- `useSelector` accepts a function that takes the store's state as an argument and returns the part of the state you want.
- In this example, it selects the `todos` slice of the state.

4. `useDispatch`

The `useDispatch` hook is used to dispatch actions to the Redux store. It's a way to trigger state changes.

Example: Using `useDispatch` to Dispatch Actions

```
import React, { useState } from 'react';
import { useDispatch } from 'react-redux';
import { addTodo } from './todosSlice';

const AddTodoForm = () => {
  const [text, setText] = useState('');
  const dispatch = useDispatch();

  const handleSubmit = (e) => {
    e.preventDefault();
    if (text.trim()) {
      dispatch(addTodo({ id: Date.now(), todo: text, completed: false }));
      setText('');
    }
  };

  return (
    <form onSubmit={handleSubmit}>
      <input
        type="text"
        value={text}
        onChange={(e) => setText(e.target.value)}
        placeholder="Add a todo"
      />
      <button type="submit">Add Todo</button>
    </form>
  );
};

export default AddTodoForm;
```

- `useDispatch` gives you a reference to the store's dispatch function, which you can use to send actions.
- In this example, `addTodo` is dispatched when the form is submitted, adding a new todo to the state.

Summary

- **Store**: The central repository that holds the application's state.
- **Reducer**: A function that defines how state changes in response to actions.
- `useSelector`: A hook to access state from the Redux store.
- `useDispatch`: A hook to dispatch actions to the Redux store.

These tools are integral to managing state with Redux Toolkit in a structured and efficient way, making it easier to build scalable and maintainable applications.

Reducer In Redux-Toolkit ⇒ Slice 🍕

```
const initialState = [
  {
    id: 1,
    todos: "Hello world",
    text: ""
  }
];

export const todoSlice = createSlice({
  name: 'todo',
  initialState,
  reducers: {
      // Properties And Function
    // Define your reducer functions here
    addTodo: (state,action) =>{},
    removeTodo : (state,action) =>{},
}
});
```

Key Points:

1. **Initial State**: The `initialState` is defined as an array of objects, where each object represents a todo item. It has properties like `id`, `todos`, and `text`.
2. **Creating the Slice**: The `todoSlice` is created using `createSlice` from Redux Toolkit.
3. **Name**: The name of the slice is set to `'todo'`.
4. **Initial State**: It uses the defined `initialState`.
5. **Reducers**: A placeholder comment is added where the reducer functions can be defined to handle various actions related to the todo items.

`addTodo: (state, action) => {}` is part of the `reducers` object in a Redux Toolkit slice. It defines a reducer function for handling the `addTodo` action. Let's break down what this means and how it works:

Explanation:

1. `state`: This parameter represents the current state of the slice. In Redux Toolkit, the state is usually an object or an array that holds the relevant data for a particular feature or module of your application. When an action is dispatched, the current state is passed to the reducer function, allowing it to update and return a new state.
2. `action`: This parameter represents the action object that was dispatched. An action typically has a `type` property, which indicates the type of action being performed, and a `payload` property, which carries any data necessary to perform the update. In the context of `addTodo`, the `payload` would likely contain the information for the new todo item (e.g., an ID and text).

Example Usage of `addTodo` Reducer

```
import { createSlice } from '@reduxjs/toolkit';

const initialState = [
  {
    id: 1,
    text: "Hello world",
    completed: false
  }
];

const todoSlice = createSlice({
  name: 'todo',
  initialState,
  reducers: {
    addTodo: (state, action) => {
      // Access the new todo item from the action's payload
      const newTodo = action.payload;
      // Add the new todo item to the current state
      state.push(newTodo);
    }
  }
});

export const { addTodo } = todoSlice.actions;
export default todoSlice.reducer;
```

How it Works:

- **Action Dispatch**: When you want to add a new todo item, you would dispatch the `addTodo` action with the new todo item's details as the payload. For example:

  ```
  dispatch(addTodo({ id: 2, text: "Learn Redux Toolkit", completed: false }));
  ```

- **Reducer Function**: The `addTodo` function is called automatically by Redux Toolkit when the `addTodo` action is dispatched. The function:
  - Receives the current `state`, which is the array of todos.
  - Receives the `action` object, which has the `payload` containing the new todo item.
  - Adds the new todo item to the state array using the `push` method.

Why Use `state` and `action`?

- `state`: Allows the reducer to know what the current list of todos is so that it can modify or append to it.
- `action`: Carries the data necessary to perform the update, ensuring that reducers are flexible and reusable for different types of updates.

Summary

The `addTodo: (state, action) => {}` is a concise way to define how your application's state should be updated when a specific action occurs (in this case, adding a new todo). The use of `state` and `action` parameters makes it possible to write predictable and pure functions that handle state transitions in a controlled manner.

`useDispatch` reducer का उपयोग करते हुए स्टोर में वैल्यू को बदलता है।

It changes the value in the store using the `useDispatch` hook and a reducer.

We Usually Give Reference Of Thing For Browser Events So Fire Call Back for That

Chrome Extension: Redux Dev Tools

Redux Steps

Redux is a standalone library that is separate from **React**.

React-Redux is different from Redux as well; it is an implementation of Redux specifically for React. React-Redux acts as a connector, allowing React and Redux to communicate with each other.

Steps to Implement Redux with React:

Step 1: Create a Store

- Typically, there is one store, but there can be multiple stores. The store serves as a single source of truth for the application's state.

Step 2: Define Reducers

- Create reducers, and ensure that the store is aware of these reducers. We refer to the different parts of the state as "slices."

Step 3: Create a Slice

- Define a slice with the following:
  1. A name
  2. An initial state
  3. A list of reducers, which handle the state and actions
- Export all the reducers.
- Export the main reducer.

Step 4: Use the Reducer in Components

- In your components, use the `dispatch` function to call reducers for actions like adding or deleting items.
- To access values from the state, use `useSelector((state) => state.propertyName)`.

Step 5: Set Up the Provider

- Import the `Provider` from `react-redux`.
- Wrap your application with the `Provider` component, passing in the store as a prop to make the Redux state available throughout the app.

Redux Toolkit Concepts

1. **Redux Toolkit**: This is the official, recommended way to write Redux logic. It provides utility functions that simplify the process of setting up and managing the Redux store. The toolkit focuses on reducing boilerplate code and improving the developer experience.
2. **Store**: In Redux, the store is a single JavaScript object that holds the entire state of the application. The store is created using `configureStore` from Redux Toolkit, which sets up the store with good defaults.
3. **Slice**: A slice is a collection of Redux reducer logic and actions for a single feature of your application. The `createSlice` function from Redux Toolkit automatically generates action creators and action types that correspond to the reducers and state.
4. **Reducers**: Reducers specify how the application's state changes in response to actions sent to the store. A reducer is a function that takes the current state and an action, then returns the next state.
5. **Actions**: Actions are plain JavaScript objects that have a `type` field and describe "what happened" in the app. In Redux Toolkit, action creators are generated automatically based on the reducer functions.

6. **`useSelector` and `useDispatch`**: `useSelector` is a hook that allows you to extract data from the Redux store state, and `useDispatch` is a hook that allows you to dispatch actions to the Redux store.

Code Breakdown

1. Store Setup (`app/store.js`)

```
// Import configureStore from Redux Toolkit
import { configureStore } from '@reduxjs/toolkit';

// Import the todoReducer from the todoSlice
import todoReducer from '../features/todo/todoSlice';

// Create the Redux store with the todoReducer
export const store = configureStore({
    reducer: todoReducer,
});
```

- **`configureStore`**: This function sets up a Redux store with good defaults. It takes an object as an argument, where you can specify the reducers to use.
- **`todoReducer`**: This is the reducer imported from `todoSlice.js`. It's responsible for managing the state of the `todo` feature. We register it under the `todo` key in the root state.

2. Todo Slice (`features/todo/todoSlice.js`)

```
// Import createSlice and nanoid from Redux Toolkit
import { createSlice, nanoid } from '@reduxjs/toolkit';

// Define the initial state for the todos
const initialState = {
    todos: [{ id: 1, text: "Hello world" }],
};

// Create a slice for the todo feature
export const todoSlice = createSlice({
    name: 'todo', // Name of the slice
    initialState, // Initial state of the slice
    reducers: {
        // Reducer to add a new todo
        addTodo: (state, action) => {
            const todo = {
                id: nanoid(), // Generate a unique id using nanoid
                text: action.payload, // Use the text passed in the action payload
            };
            state.todos.push(todo); // Add the new todo to the state
        },
        // Reducer to remove a todo by id
        removeTodo: (state, action) => {
            state.todos = state.todos.filter((todo) => todo.id !== action.payload);
            // Update the state by filtering out the todo with the matching id
        },
    },
});

// Export actions to use them in components
export const { addTodo, removeTodo } = todoSlice.actions;
```

```
// Export the reducer to be used in the store
export default todoSlice.reducer;
```

- `createSlice`: This function generates a slice of the state, along with the reducers and actions. The slice is named `todo`, and it starts with an `initialState` containing one todo item.
- `initialState`: This object defines the starting state for the `todo` slice, which is an array with a single todo item.
- **Reducers**:
  - `addTodo`: Takes the current state and an action as arguments. It adds a new todo item to the `todos` array. The new todo item is created with a unique `id` using `nanoid` and the text passed in `action.payload`.
  - `removeTodo`: Takes the current state and an action. It filters the `todos` array to remove the todo item with the `id` provided in `action.payload`.
- **Action Creators**: `addTodo` and `removeTodo` action creators are automatically generated and exported from the slice.

3. AddTodo Component (`components/AddTodo.jsx`)

```
import React, { useState } from 'react';
import { useDispatch } from 'react-redux';
import { addTodo } from '../features/todo/todoSlice';

function AddTodo() {
    // Local state to manage the input field
    const [input, setInput] = useState('');
    // Get the dispatch function to dispatch actions
    const dispatch = useDispatch();

    // Handler function to handle form submission
    const addTodoHandler = (e) => {
        e.preventDefault(); // Prevent default form submission
        dispatch(addTodo(input)); // Dispatch the addTodo action with the input value
        setInput(''); // Reset the input field
    };

    return (
        <form onSubmit={addTodoHandler} className="space-x-3 mt-12">
            <input
                type="text"
                className="bg-gray-800 rounded border border-gray-700 focus:border-indigo-500
focus:ring-2 focus:ring-indigo-900 text-base outline-none text-gray-100 py-1 px-3 leading-8 t
ransition-colors duration-200 ease-in-out"
                placeholder="Enter a Todo..."
                value={input} // Controlled input component
                onChange={(e) => setInput(e.target.value)} // Update input state on change
            />
            <button
                type="submit"
                className="text-white bg-indigo-500 border-0 py-2 px-6 focus:outline-none hov
er:bg-indigo-600 rounded text-lg"
            >
                Add Todo
            </button>
        </form>
    );
}

export default AddTodo;
```

- **Local State**: Uses a `useState` hook to manage the local state of the input field.
- **Dispatching Actions**: The `useDispatch` hook is used to get the dispatch function from the Redux store. When the form is submitted, the `addTodo` action is dispatched with the current input value.

4. Todos Component (`components/Todos.jsx`)

```
import React from 'react';
import { useSelector, useDispatch } from 'react-redux';
import { removeTodo } from '../features/todo/todoSlice';

function Todos() {
    // Get todos from the Redux store using useSelector
    const todos = useSelector((state) => state.todos);
    // Get the dispatch function to dispatch actions
    const dispatch = useDispatch();

    return (
        <>
            <div>Todos</div>
            <ul className="list-none">
                {todos.map((todo) => (
                    <li
                        className="mt-4 flex justify-between items-center bg-zinc-800 px-4 py
-2 rounded"
                        key={todo.id}
                    >
                        <div className='text-white'>{todo.text}</div>
                        <button
                            onClick={() => dispatch(removeTodo(todo.id))}
                            // Dispatch the removeTodo action with the todo id
                            className="text-white bg-red-500 border-0 py-1 px-4 focus:outline
-none hover:bg-red-600 rounded text-md"
                        >
                            <svg
                                xmlns="http://www.w3.org/2000/svg"
                                fill="none"
                                viewBox="0 0 24 24"
                                strokeWidth={1.5}
                                stroke="currentColor"
                                className="w-6 h-6"
                            >
                                <path
                                    strokeLinecap="round"
                                    strokeLinejoin="round"
                                    d="M14.74 9l-.346 9m-4.788 0L9.26 9m9.968-3.21c.342.052.6
82.107 1.022.166m-1.022-.165L18.16 19.673a2.25 2.25 0 01-2.244 2.077H8.084a2.25 2.25 0 01-2.2
44-2.077L4.772 5.79m14.456 0a48.108 48.108 0 00-3.478-.397m-12 .562c.34-.059.68-.114 1.022-.1
65m0 0a48.11 48.11 0 013.478-.397m7.5 0v-.916c0-1.18-.91-2.164-2.09-2.201a51.964 51.964 0 00-
3.32 0c-1.18.037-2.09 1.022-2.09 2.201v.916m7.5 0a48.667 48.667 0 00-7.5 0"
                                />
                            </svg>
                        </button>
                    </li>
                ))}
            </ul>
        </>
    );
}

export default Todos;
```

- `useSelector`: This hook selects data from the Redux store. It accesses the `todos` array from the `todo` slice of the state.
- `useDispatch`: Again, this hook is used to dispatch the `removeTodo` action, passing in the ID of the todo to be removed.

5. Main Application (`App.jsx` and `main.jsx`)

`App.jsx`

```
import React from 'react';
import './App.css';
import AddTodo from './components/AddTodo'; // Import AddTodo component
import Todos from './components/Todos'; // Import Todos component

function App() {
    return (
        <>
            <h1>React-Redux Toolkit</h1>
            <AddTodo /> {/* Render AddTodo component */}
            <Todos /> {/* Render Todos component */}
        </>
    );
}

export default App;
```

`main.jsx`

```
import React from 'react';
import ReactDOM from 'react-dom/client';
import App from './App.jsx'; // Import main App component
import './index.css';
import { Provider } from 'react-redux'; // Import Provider from react-redux
import { store } from './app/store'; // Import store

// Render the App component wrapped in the Redux Provider
ReactDOM.createRoot(document.getElementById('root')).render(
    <Provider store={store}> {/* Provide the Redux store to the application */}
        <App /> {/* Render the main App component */}
    </Provider>,
);
```

- `App.jsx`: Renders the main components, `AddTodo` and `Todos`, which interact with the Redux store.
- `main.jsx`: The `Provider` component from React-Redux wraps the `App` component, passing the Redux store down to all components.

Summary

1. **Store**: Configured using `configureStore`, contains all the reducers, state, and middleware.
2. **Slices**: Manage state and actions for a specific part of the app. Here, `todoSlice` handles todo-related state and actions.
3. **Reducers and Actions**: Automatically generated by `createSlice`. `addTodo` and `removeTodo` are used to update the state.
4. **Components**: Use `useSelector` to read from the store and `useDispatch` to dispatch actions.
5. **Provider**: Wraps the root component to make the store available throughout the application.

By using Redux Toolkit, this application is structured in a way that reduces boilerplate code, making it easier to manage state and actions efficiently.