REACT

Disclaimer:

These notes have been created by *Khushal Savalakha* (khushal.sav@gmail.com) and are based on practical applications and personal understanding of the concepts. Although I have thoroughly checked and verified the information and code provided, there is a possibility that some aspects may contain errors or inaccuracies. Please use this content as a reference and verify details independently as needed. I am sharing these notes as part of a GitHub repository to help others learn, and I hope they are beneficial.

▼ USE OF REACT

React, as a powerful and versatile library, provides several functionalities that can serve as alternatives to various traditional approaches and libraries. Here are the key areas where React can replace or significantly reduce the need for other libraries or custom implementations:

1. State Management

- **Alternative to**: Redux, MobX, Recoil, Zustand
- **React's Solution**: React's built-in `useState`, `useReducer`, and Context API for managing local and global state.

2. DOM Manipulation

- **Alternative to**: jQuery, vanilla JavaScript

Khushal Savalakha

- **React's Solution**: React's declarative approach abstracts direct DOM manipulation, making it unnecessary to use jQuery for most tasks.

3. Event Handling

- **Alternative to**: jQuery, vanilla JavaScript
- **React's Solution**: React provides a synthetic event system that works consistently across all browsers, simplifying event handling.

4. AJAX Requests

- **Alternative to**: jQuery.ajax, Axios, Fetch API
- **React's Solution**: While React doesn't handle AJAX requests directly, it integrates seamlessly with the Fetch API or Axios, allowing you to manage data fetching in lifecycle methods or hooks (`useEffect`).

5. Form Handling and Validation

- **Alternative to**: Formik, Redux Form, jQuery Validation Plugin
- **React's Solution**: React's controlled components and state management make it straightforward to handle form inputs and validations without additional libraries.

6. Routing

- **Alternative to**: Backbone.js, Angular Router, traditional server-side routing
- **React's Solution**: React Router (a complementary library) provides a powerful routing solution that integrates seamlessly with React components.

7. Component-Based UI Development

- **Alternative to**: Traditional MVC frameworks, templating engines like Handlebars or Mustache
- **React's Solution**: React's component-based architecture promotes reusable and modular UI development, replacing the need for many traditional approaches.

8. Animations and Transitions

- **Alternative to**: CSS Animations, jQuery animations, GreenSock (GSAP)
- **React's Solution**: React Transition Group and React Spring provide powerful tools for handling animations and transitions within React components.

Khushal Savalakha

9. Real-Time Updates

- **Alternative to**: Socket.io, Firebase Realtime Database
- **React's Solution**: While React doesn't provide real-time capabilities out of the box, it integrates well with WebSockets, Server-Sent Events (SSE), and libraries like Socket.io to manage real-time updates effectively.

10. Client-Side Rendering (CSR)

- **Alternative to**: Server-Side Rendering (SSR) with traditional frameworks
- **React's Solution**: React supports CSR by default, allowing for dynamic and interactive web applications. For SSR, frameworks like Next.js build on React to provide optimized SSR solutions.

11. Static Site Generation (SSG)

- **Alternative to**: Jekyll, Hugo, Gatsby
- **React's Solution**: Frameworks like Gatsby and Next.js (which are built on top of React) provide static site generation capabilities, combining the benefits of React with static site performance.

12. Contextual State Management

- **Alternative to**: Global state management libraries
- **React's Solution**: The Context API allows you to share state across the component tree without having to pass props down manually at every level.

13. Conditional Rendering

- **Alternative to**: Templating engines, complex conditional logic in vanilla JavaScript
- **React's Solution**: React's JSX and conditional rendering capabilities simplify displaying components based on application state.

14. Component Lifecycle Management

- **Alternative to**: Manual setup and teardown of components
- **React's Solution**: Lifecycle methods (class components) and hooks like `useEffect` (functional components) manage side effects, data fetching, subscriptions, and cleanup efficiently.

Summary

Khushal Savalakha

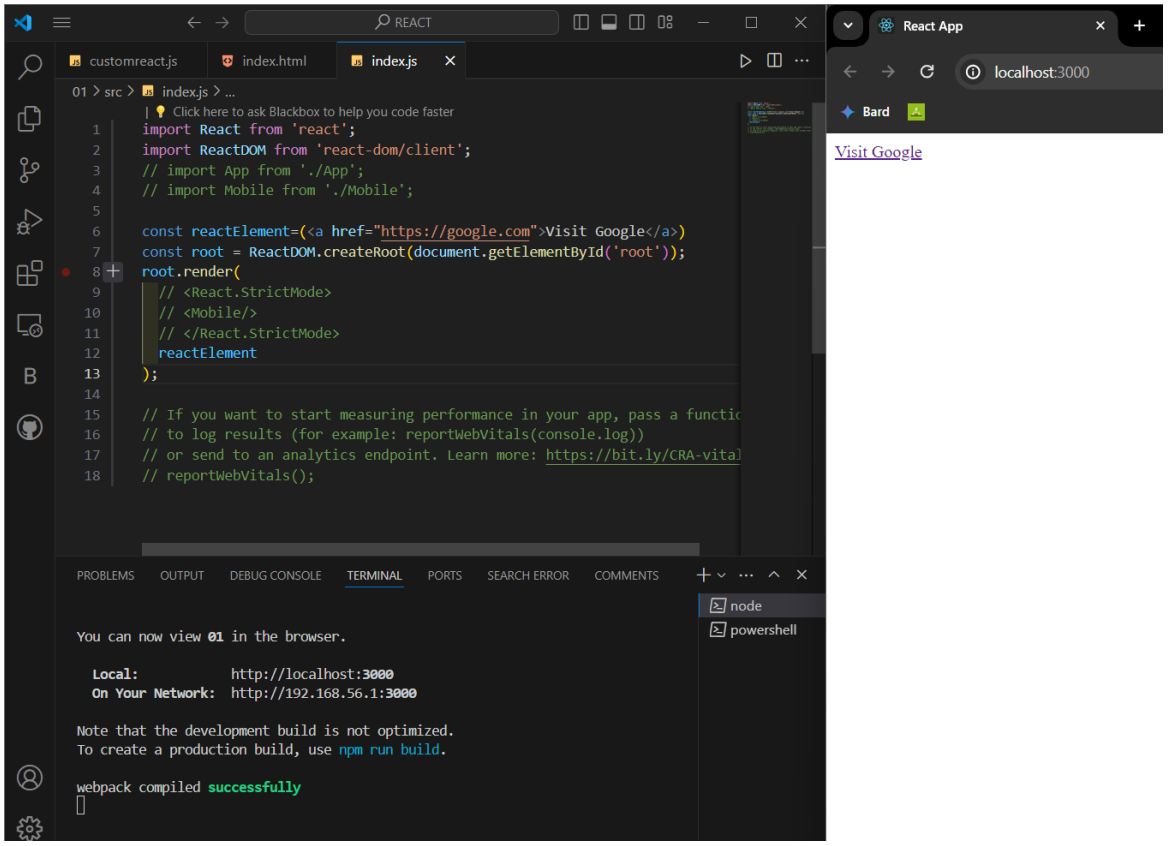
React provides a unified and efficient way to handle many aspects of front-end development, often replacing the need for multiple separate libraries or custom code. Its strengths lie in its component-based architecture, declarative UI, efficient state management, and strong community support. While React itself doesn't handle everything (e.g., routing, AJAX, real-time updates), it integrates seamlessly with complementary tools and libraries to provide a comprehensive development experience.

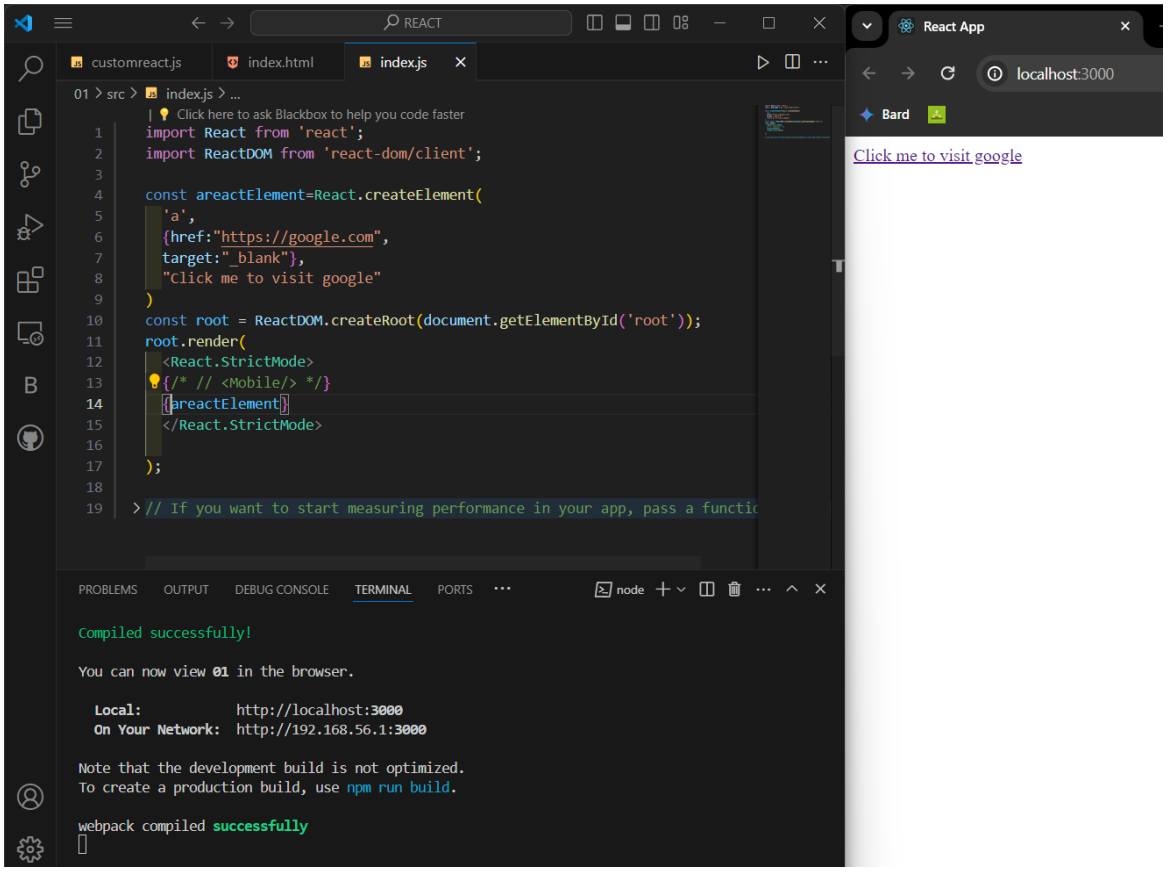
> 💡 IF we install something that dependencies we can see in package.json

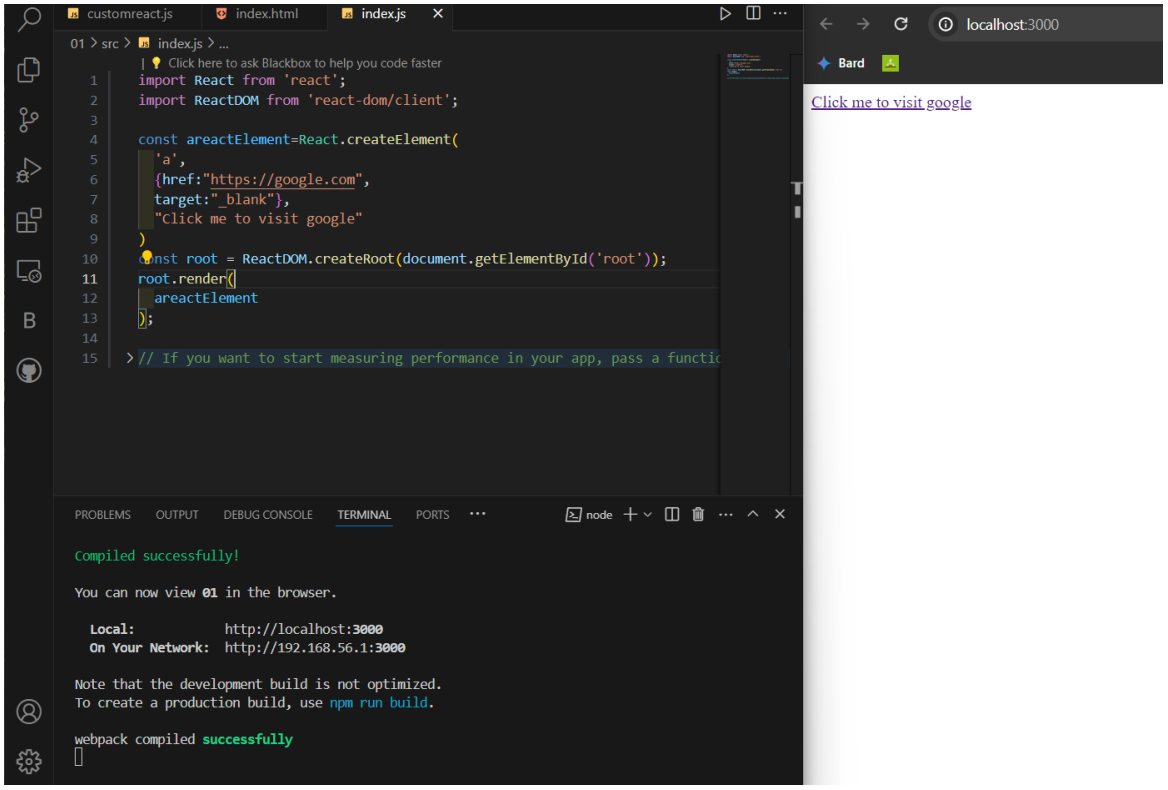
▼ BASIC

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

const areactElement=React.createElement(
  'a',
  {href:"https://google.com",
  target:"_blank"},
  "Click me to visit google"
)
const root = ReactDOM.createRoot(document.getElementById('r
root.render(
  <React.StrictMode>
  {/* // <Mobile/> */}
  {areactElement}
  </React.StrictMode>

);
```


Khushal Savalakha


REACT
customreact.js
index.html
index.js
01 > src > index.js > ...
Click here to ask Blackbox to help you code faster
import React from 'react';
import ReactDOM from 'react-dom/client';
// import App from './App';
// import Mobile from './Mobile';
const reactElement=(<a href="https://google.com">Visit Google</a>)
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(
// <React.StrictMode>
// <Mobile/>
// </React.StrictMode>
reactElement
);
// If you want to start measuring performance in your app, pass a functic
// to log results (for example: reportWebVitals(console.log))
// or send to an analytics endpoint. Learn more: https://bit.ly/CRA-vital
// reportWebVitals();
PROBLEMS
OUTPUT
DEBUG CONSOLE
TERMINAL
PORTS
SEARCH ERROR
COMMENTS
node
powershell
You can now view 01 in the browser.
Local: http://localhost:3000
On Your Network: http://192.168.56.1:3000
Note that the development build is not optimized.
To create a production build, use npm run build.
webpack compiled successfully
React App
localhost:3000
Bard
Visit Google


REACT
customreact.js
index.html
index.js
01 > src > index.js > ...
Click here to ask Blackbox to help you code faster
import React from 'react';
import ReactDOM from 'react-dom/client';
const areactElement=React.createElement(
'a',
{href:"https://google.com",
target:"_blank"},
"Click me to visit google"
)
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(
<React.StrictMode>
{/* // <Mobile/> */}
{areactElement}
</React.StrictMode>
);
// If you want to start measuring performance in your app, pass a functic
PROBLEMS
OUTPUT
DEBUG CONSOLE
TERMINAL
PORTS
node
Compiled successfully!
You can now view 01 in the browser.
Local: http://localhost:3000
On Your Network: http://192.168.56.1:3000
Note that the development build is not optimized.
To create a production build, use npm run build.
webpack compiled successfully
React App
localhost:3000
Bard
Click me to visit google



In React, the `React.createElement()` function is used to create React elements, which are the building blocks of React applications. The first argument to `React.createElement()` is the type of element you want to create. In your code, `'a'` is passed as the type, which represents an HTML anchor element (`<a>`).

Here's why you write `'a'`:

1. **Type of Element**: The first argument represents the type of element you want to create. In HTML, an anchor element is represented by the `<a>` tag, which is used for creating hyperlinks. So, `'a'` is used to specify that you want to create an anchor element.

2. **Syntax**: In React, the syntax for creating elements is different from HTML. Instead of directly writing HTML tags, you use `React.createElement()` function. So, you specify the type of HTML element you want to create as a string. In this case, `'a'` represents the `<a>` tag.

3. **Dynamic Elements**: Using a string for the element type allows you to dynamically create elements based on variables or conditions. For example, you might have a variable that determines whether to create a `<div>` or an `<a>`, and you can use that variable as the type argument for `React.createElement()`.

In your code, `'a'` is used to create an anchor element (`<a>`), which is a clickable link. The `href` attribute specifies the URL the link points to, and the `target` attribute specifies

Khushal Savalakha

where to open the link (in this case, `_blank` means it will open in a new tab). The text `"Click me to visit google"` is the content of the anchor element, which will be displayed as the clickable text on the webpage.

> if there is a 4 line html code then how React.createElement() work to create object like above
> give me new example.

Sure, let's say you have a simple HTML code with four lines:

```
<div id="container">
  <h1>Hello, World!</h1>
  <p>This is a paragraph.</p>
  <button>Click me</button>
</div>
```

Now, let's translate this HTML code into React using `React.createElement()`:

```
const containerElement = React.createElement(
  'div',
  { id: 'container' },
  React.createElement('h1', null, 'Hello, World!'),
  React.createElement('p', null, 'This is a paragrap
h.'),
  React.createElement('button', null, 'Click me')
);

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

In this example:

1. We create a `div` element as the container with an `id` of `'container'`.
2. Inside the `div`, there are three child elements:
   - An `h1` element with the text `'Hello, World!'`.
   - A `p` element with the text `'This is a paragraph.'`.

Khushal Savalakha

- A `button` element with the text `'Click me'`.

Each `React.createElement()` call corresponds to an HTML element in the original code. The first argument specifies the type of element, the second argument is an object for the element's properties (like `id` in the `div` element), and the subsequent arguments (if any) are the children elements.

So, in this example, `React.createElement()` is used to create a hierarchy of React elements representing the HTML structure, just like we have in the HTML code.

> React.createElement('h1', null, 'Hello, World!'),
> why we have to write null here .

In the `React.createElement()` function, the second argument is for the element's properties, also known as props. If an element doesn't have any props, you can pass `null` as the second argument.

In the case of the `h1` element in your example:

```
React.createElement('h1', null, 'Hello, World!')
```

There are no additional properties (props) being passed to the `h1` element, so `null` is used as a placeholder for the props argument. If you had props to pass, you would replace `null` with an object containing those props.

For example, if you wanted to add a class to the `h1` element, you could pass it as props:

```
React.createElement('h1', { className: 'heading' }, 'Hel
lo, World!')
```

But since there are no additional properties in your case, `null` is sufficient as a placeholder.

> props means id or class ?

In React, props (short for "properties") are used to pass data from one component to another. Props can contain any type of data, including strings, numbers, arrays, objects, and even functions. They are similar to attributes in HTML elements, but they are used to pass data from parent components to child components in React.

In the context of the `React.createElement()` function, props refer to the additional properties you can assign to the created React element. These properties can include

Khushal Savalakha

attributes such as `id`, `className`, `style`, `onClick`, and any other custom attributes you define.

So, when I mentioned "props" in the previous response, I was referring to properties such as `className`, `id`, `style`, etc., that you might want to assign to your React elements.

For example, if you want to add a class to an `h1` element, you would pass it as a prop like this:

```
React.createElement('h1', { className: 'heading' }, 'Hel
lo, World!')
```

Here, `{ className: 'heading' }` is an object containing the prop `className` with the value `'heading'`. Similarly, you can pass other props such as `id`, `style`, `onClick`, etc., depending on your requirements.

💡 More Complex Example

Sure, let's create a more complex example where the `<div id="container">` contains an inner `<div>` along with the existing elements:

```
<div id="container">
  <h1>Hello, World!</h1>
  <p>This is a paragraph.</p>
  <button>Click me</button>
  <div id="inner-container">
    <p>This is an inner paragraph.</p>
    <ul>
      <li>Item 1</li>
      <li>Item 2</li>
      <li>Item 3</li>
    </ul>
  </div>
</div>
```

Now, let's translate this HTML code into React using `React.createElement()`:

Khushal Savalakha

```
const innerDivElement = React.createElement(
  'div',
  { id: 'inner-container' },
  React.createElement('p', null, 'This is an inner parag
raph.'),
  React.createElement('ul', null,
    React.createElement('li', null, 'Item 1'),
    React.createElement('li', null, 'Item 2'),
    React.createElement('li', null, 'Item 3')
  )
);

const containerElement = React.createElement(
  'div',
  { id: 'container' },
  React.createElement('h1', null, 'Hello, World!'),
  React.createElement('p', null, 'This is a paragrap
h.'),
  React.createElement('button', null, 'Click me'),
  innerDivElement
);

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

In this example:

1. We create an inner `div` element with an `id` of `'inner-container'`.
2. Inside the inner `div`, there are two child elements:
   - A `p` element with the text `'This is an inner paragraph.'`.
   - A `ul` element with three `li` child elements.

Then, we include this inner `div` element as a child of the main `div` with the id `'container'`. So, the structure is preserved, and the inner `div` with its contents is nested inside the main `div`.

Khushal Savalakha

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


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

// If you want to start measuring performance in your app,
// to log results (for example: reportWebVitals(console.log
// or send to an analytics endpoint. Learn more: https://bi
// reportWebVitals();
```

`React.StrictMode` is a tool that helps in highlighting potential problems in your React application. It's not about changing how your components behave in production but rather about detecting and warning about unsafe code in development and transitioning APIs in future versions of React.

Here are some of the things `React.StrictMode` does:

1. **Identifying Unsafe Lifecycle Methods and Side Effects**: It checks your components for unsafe lifecycle methods and legacy API usage. For example, it will warn you if you use deprecated lifecycle methods or if you call `setState` inside a `render` method.
2. **Detecting Legacy Context API Usage**: If you're using the older Context API (before React 16.3), `React.StrictMode` will warn you about it. It encourages you to migrate to the new Context API for better performance and flexibility.
3. **Spotting Unsafe Updates Inside `render`**: If you accidentally cause a component to re-render during the `render` phase, `React.StrictMode` will warn you about it. This helps to catch potential bugs related to render cycles.
4. **Identifying Deprecated `findDOMNode` Usages**: It warns you if you use `ReactDOM.findDOMNode()` which is considered deprecated and might be removed in future versions of React.

Khushal Savalakha

5. **Detecting Legacy Ref APIs**: If you use legacy `ref` APIs like `String Refs`, `React.StrictMode` will warn you about it. It encourages using the new `React.createRef()` or callback refs instead.

In summary, `React.StrictMode` is a development mode tool that helps you write better React code by detecting and warning about potential problems, deprecated APIs, and unsafe patterns. It's not for use in production but rather to assist during development and debugging.

What is component?

In React, a component is a reusable piece of user interface (UI) that can contain both structure (HTML or JSX) and behavior (JavaScript logic). Components are the building blocks of React applications, allowing you to break down complex UIs into smaller, manageable pieces.

There are two main types of components in React:

1. **Function Components**: Also known as stateless functional components, these are JavaScript functions that take props (short for properties) as input and return React elements to describe what should appear on the screen. Function components are typically used for simpler UI elements without state or lifecycle methods.

   Example:

   ```
   function Welcome(props) {
     return <h1>Hello, {props.name}</h1>;
   }
   ```

2. **Class Components**: Class components are ES6 classes that extend from `React.Component`. They have a render method that returns the React elements to be rendered. Class components can have state and lifecycle methods, making them suitable for more complex UIs and interactions.

   Example:

   ```
   class Welcome extends React.Component {
     render() {
       return <h1>Hello, {this.props.name}</h1>;
     }
   }
   ```


Khushal Savalakhia

Components can be composed together to create larger UIs. For example, you might have a `Header` component, a `Sidebar` component, and a `MainContent` component, and then compose them together within a `App` component to create the overall structure of your application.

Using components promotes code reusability, maintainability, and modularity in React applications, as you can easily reuse and compose them to build complex user interfaces.

Always Component's name start with Capital Letter it means it follows Camel Case.

JSX

JSX (JavaScript XML) is a syntax extension for JavaScript that allows you to write HTML-like code directly within JavaScript. It's commonly used with React to describe the UI elements that should be rendered. JSX makes it easier to write and understand React components by blending HTML-like syntax with JavaScript.

Here's an example of JSX:

```
const element = <h1>Hello, world!</h1>;
```

In this example:

- `<h1>Hello, world!</h1>` looks like HTML but is actually JSX.
- JSX gets transpiled into regular JavaScript function calls by tools like Babel, which converts JSX into `React.createElement()` calls.

Under the hood, JSX is transformed into regular JavaScript code. The above JSX code gets transformed into:

```
const element = React.createElement('h1', null, 'Hello,
world!');
```

JSX also allows you to embed JavaScript expressions within curly braces `{}`. This allows you to dynamically generate content or attributes based on variables or expressions:

```
const name = 'John';
const element = <h1>Hello, {name}</h1>;
```

This will render `<h1>Hello, John</h1>`.

Khushal Savalakha

JSX is not required to write React applications, but it's widely used because it provides a more concise and expressive way to define React components compared to using `React.createElement()` directly. It's important to note that JSX is not HTML but a syntax extension that gets transformed into regular JavaScript function calls during the build process.

In React JSX, curly braces `{}` are used to embed JavaScript expressions within JSX syntax. This allows you to dynamically generate content, evaluate variables, or execute JavaScript code within your JSX elements.

Here are some common uses of curly braces `{}` in React JSX:

1. **Rendering Dynamic Content**:

   ```
   const name = "John";
   return <h1>Hello, {name}!</h1>;
   ```

   In this example, the value of the `name` variable is inserted dynamically into the JSX element.

2. **Executing JavaScript Expressions**:

   ```
   return <h1>2 + 2 equals {2 + 2}</h1>;
   ```

   Here, the JavaScript expression `{2 + 2}` is evaluated and the result (`4`) is rendered within the JSX element.

3. **Accessing Object Properties**:

   ```
   const user = { name: "John", age: 30 };
   return <p>User's name is {user.name} and age is {use
   r.age}</p>;
   ```

   You can access properties of JavaScript objects within JSX using curly braces.

4. **Conditional Rendering**:

   ```
   return <div>{isLogged ? <p>Welcome back!</p> : <p>Ple
   ase log in.</p>}</div>;
   ```

   Curly braces allow you to conditionally render different JSX elements or expressions based on JavaScript conditions.

Khushal Savalakha

5. **Mapping Arrays**:

```
const numbers = [1, 2, 3, 4, 5];
return (
  <ul>
    {numbers.map((number) => (
      <li key={number}>{number}</li>
    ))}
  </ul>
);
```

You can use curly braces to iterate over arrays and map them to JSX elements.

In summary, curly braces `{}` in React JSX are used to inject JavaScript expressions, variables, object properties, or execute JavaScript code within JSX elements, enabling dynamic and flexible rendering of content.

```
import './App.css';
/*function App() {
  return (
    <>
      <h1>hello iiii am Component</h1>
      <p>Hello i am khushal Savalakha</p>
    </>
    // <></> It's a Fragment tag which is used to wrap mult
  );
}*/
function App() {
  let a = 45;
  return (
    <div>
      {a}
      <br/>
      {a+5}
      <h1 className='main'>hello iiii am Component</h1>
      <p style={{backgroundColor:"orange",color:"white"}}>H
    </div>
  );
}
```


Khushal Savalakha

```
export default App;
```

```
.main{
    background-color:skyblue;
    text-decoration:underline;
    font-style: italic;
}
```

If we want to write style in javascript code then we have to use {{STYLE CODE GOES HERE..}}

DOM

DOM stands for Document Object Model. It's a programming interface for web documents. Simply put, it's a way for programs to interact with HTML, XML, or XHTML documents.

Here's a simplified explanation:

1. **Document**: The whole web page loaded in the browser is called the document. It includes everything from the `<html>` tag to the `</html>` tag.
2. **Object**: Each element in the HTML document, like `<h1>`, `<p>`, `<div>`, etc., is represented as an object in the DOM. These objects can be accessed, manipulated, and changed using programming languages like JavaScript.
3. **Model**: The DOM provides a structured representation of the document as a tree-like structure. Each HTML element is a node in this tree, and they are organized in a hierarchical manner, where one element can be the parent, child, or sibling of another element.

Here's a simple example:

Consider the following HTML document:

```
<!DOCTYPE html>
<html>
<head>
  <title>DOM Example</title>
```


Khushal Savalakha

```
</head>
<body>
  <h1>Hello, World!</h1>
  <p>This is a paragraph.</p>
</body>
</html>
```

In this HTML document:

- The `<html>` element is the root of the DOM tree.
- Inside `<html>`, there's a `<head>` element and a `<body>` element.
- `<head>` contains a `<title>` element.
- `<body>` contains an `<h1>` element and a `<p>` element.

In JavaScript, you can interact with this HTML document using the DOM. For example, you can access the `<h1>` element and change its content like this:

```
// Get the <h1> element
const heading = document.querySelector('h1');

// Change its content
heading.textContent = 'Hello, New World!';
```

This JavaScript code uses DOM methods to select the `<h1>` element and change its text content. This is just a simple example of how you can use the DOM to manipulate web documents dynamically.

```
import './App.css';
/*function App() {
  return (
    <>
      <h1>hello iiii am Component</h1>
      <p>Hello i am khushal Savalakha</p>
    </>
    // <></> It's a Fragment tag which is used to wrap mult
  );
}*/
function App() {
```


Khushal Savalakha

```
  let a = 45;
  return (
    <div>
      {a}
      <br/>
      {a+5}
      <h1 className='main'>hello iiii am Component</h1>
      <p style={{backgroundColor:"orange",color:"white"}}>H
    </div>
  );
}

export default App;
```

In the context of file paths in a React project, './' and '../' are used to specify the location of files relative to the current file or directory.

- **'./'**: This refers to the current directory. When you import or reference a file using './', you are specifying that the file is located in the same directory as the file in which the import or reference is written.
- **'../'**: This refers to the parent directory. When you import or reference a file using '../', you are specifying that the file is located in the directory one level above the current directory.

Let's illustrate with an example:

Suppose you have the following file structure:

```
my-react-app/
  ├── src/
  |   ├── components/
  |   |   ├── Header/
  |   |   |   ├── Header.js
  |   |   |   └── Header.css
  |   |   ├── Sidebar/
  |   |   |   ├── Sidebar.js
  |   |   |   └── Sidebar.css
  |   |   └── MainContent/
  |   |       ├── MainContent.js
  |   |       └── MainContent.css
```


Khushal Savalakha

```
|    ├── App.js
|    └── index.js
```

In the `Header.js` file, if you want to import the CSS file located in the same directory, you would use './':

```
import './Header.css';
```

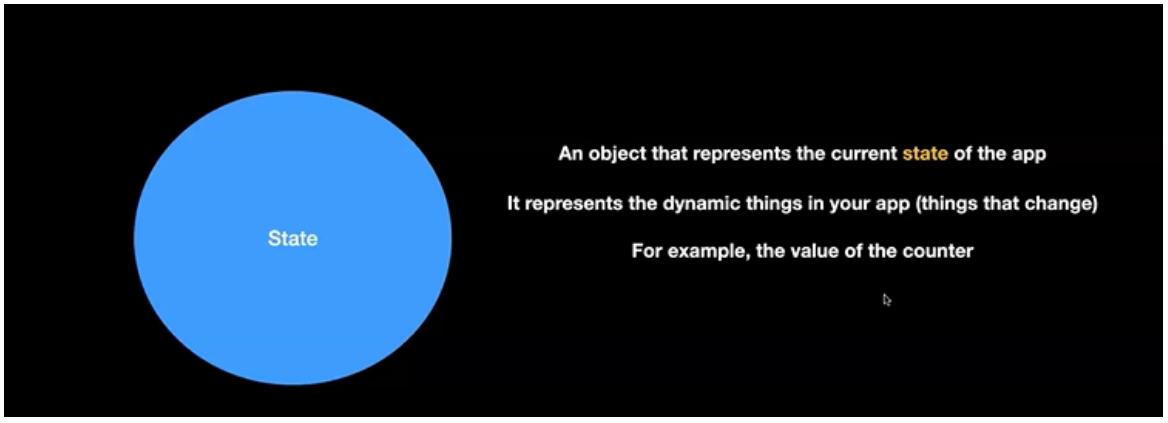
This tells the JavaScript environment to look for the `Header.css` file in the same directory as `Header.js`.

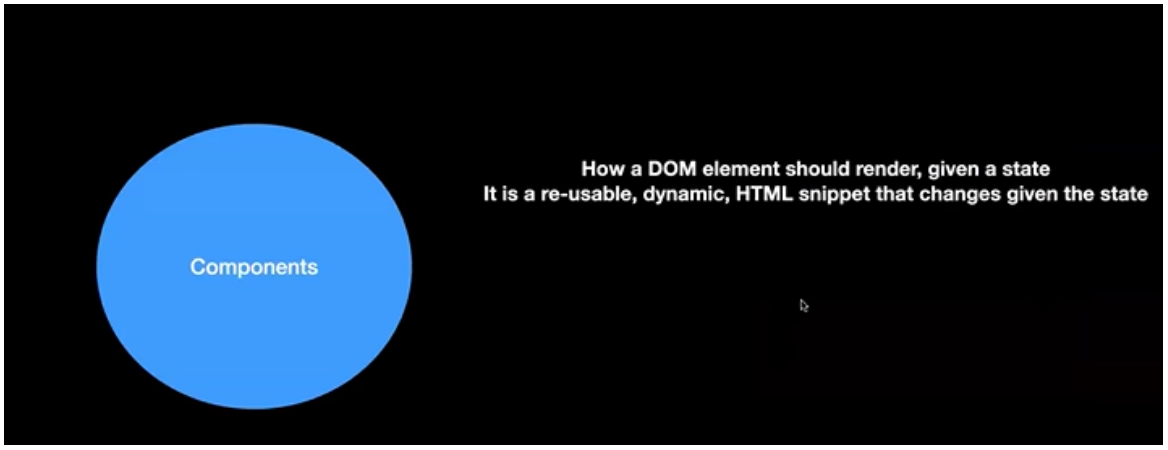
Now, if you're in the `Header.js` file and want to import something from the `MainContent` directory, you would use '../' to navigate to the parent directory and then access the `MainContent` directory:

```
import MainContent from '../MainContent/MainContent';
```

This tells the JavaScript environment to navigate up one level (to the `components` directory) and then access the `MainContent` directory.

▼ BASIC 2





In the context of the image you provided, a "component" refers to a modular and reusable piece of a web application's user interface. It encapsulates the structure (HTML), style (CSS), and behavior (JavaScript) required to render a specific part of the UI. Components are dynamic, meaning they can change their appearance or behavior based on the application's state.

Here's a more detailed breakdown:

- **Re-usable**: A component can be used multiple times throughout the application without duplicating code.
- **Dynamic**: It can update or change in response to changes in the application's state (such as user input or data changes).
- **HTML Snippet**: It represents a portion of the HTML that makes up the user interface.
- **State-driven**: The rendering of the component depends on the application's current state, ensuring that it reflects real-time data or user interactions.

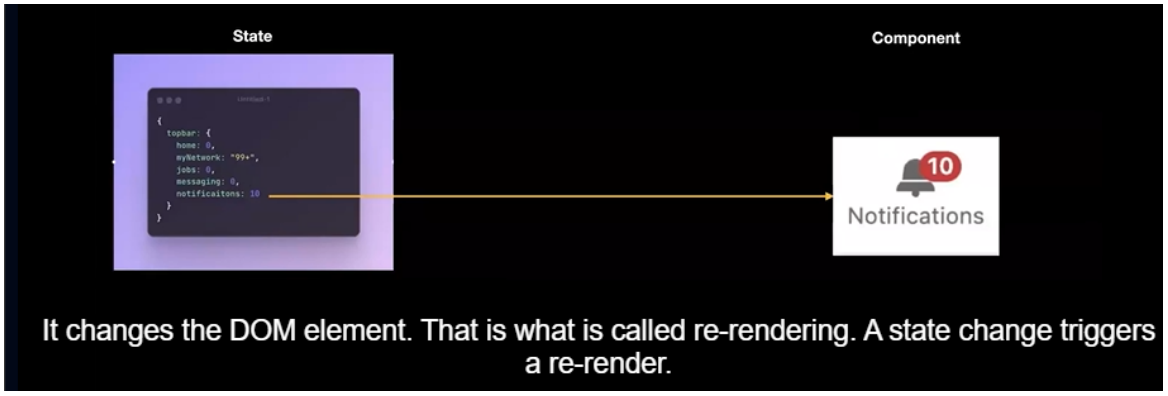
This approach is widely used in modern web development frameworks like React, Angular, and Vue.js, where components are fundamental building blocks for creating interactive and maintainable user interfaces.

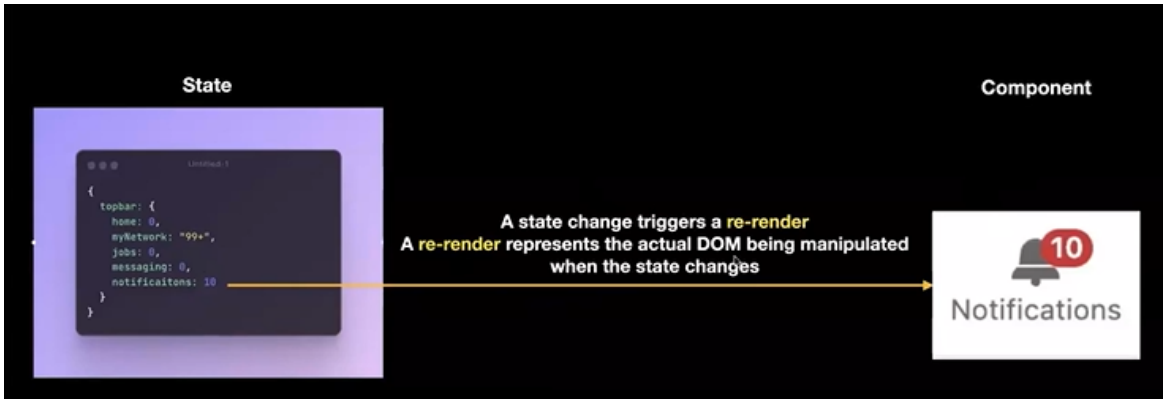
💡 how it aspects to write code ?

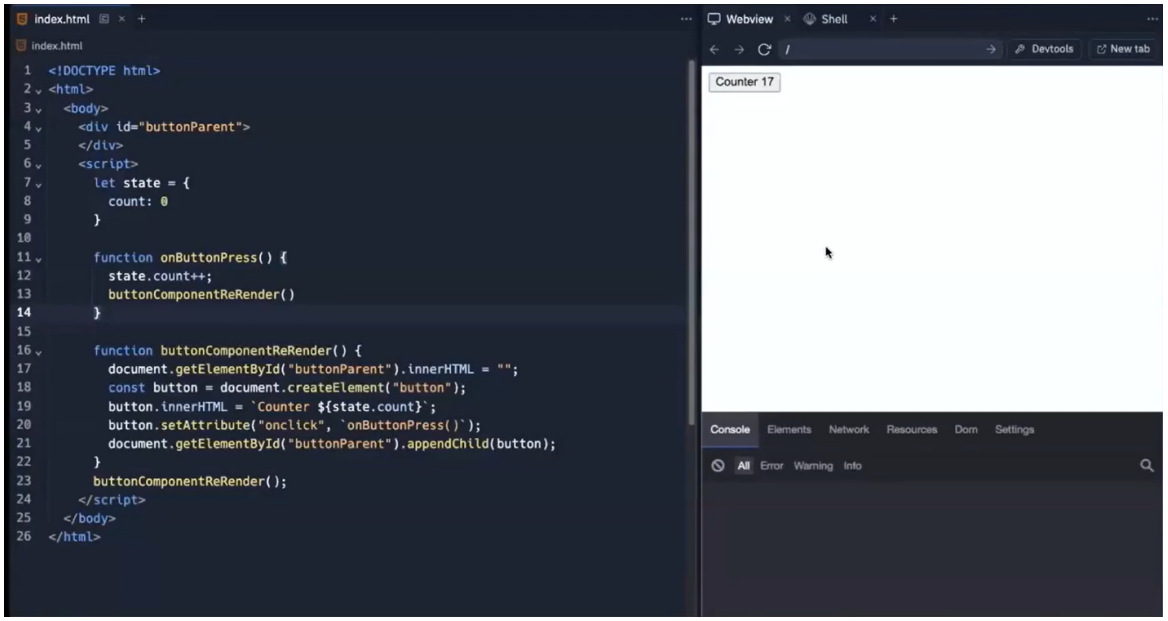
1]Give me a state of your application which means a very big object that reperesents all the dynamic parts of your application.

Khushal Savalakha

here whenever you know the backend tells the website k number of notifications have gone up.A new connedction request came in.Whatever the reason.There is some backend request that might go out of 20 seconds to check the current notificzation count.it is increased by 2 this to 12 and react re-renders.

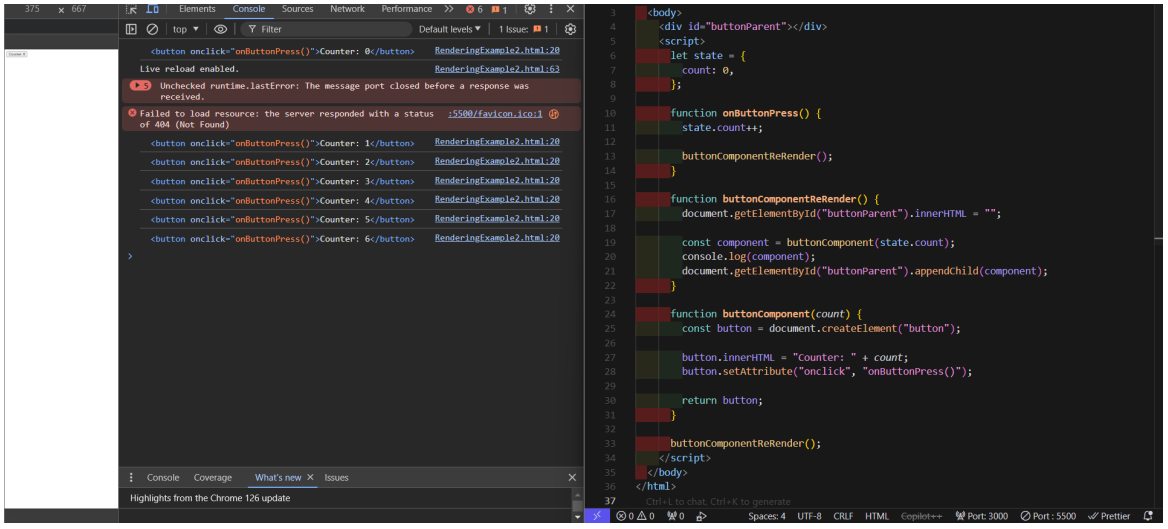


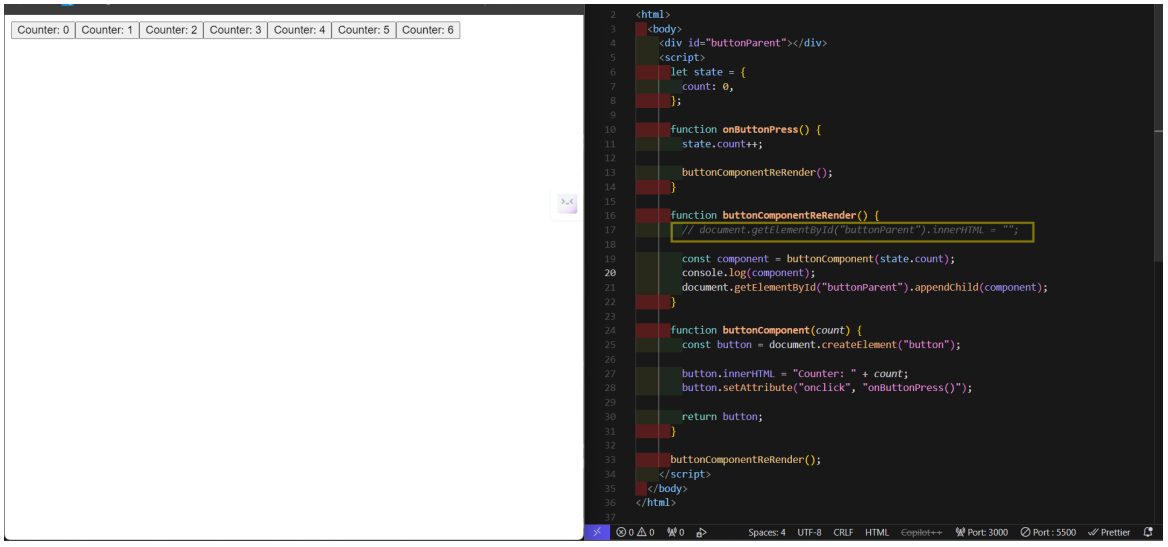





```
<!DOCTYPE html>
<html>
  <body>
    <div id="buttonParent"></div>
    <script>
      let state = {
        count: 0,
      };

      function onButtonPress() {
        state.count++;
                //A state change triggers a re-rendering.
        buttonComponentReRender();
      }//Just have to write it for calculation
      //In react you just have to write below part easily
      //don't have to write to much logic like above code

      function buttonComponentReRender() {
        document.getElementById("buttonParent").innerHTML =

        const component = buttonComponent(state.count);
        document.getElementById("buttonParent").appendChild
      }
```

```
      function buttonComponent(count) {
        const button = document.createElement("button");

        button.innerHTML = "Counter: " + count;
        button.setAttribute("onclick", "onButtonPress()");

        return button;
      }

      buttonComponentReRender();
    </script>
  </body>
</html>
```





Khushal Savalakha

here it is creating new button every time because we deleted

```
document.getElementById("buttonParent").innerHTML =
//it will set content of this tag to empty.
```

> 💡 it means initially a function called and it will append the child which has button count while clicking on button it will again call the function and add child with same name and previous child will be overridden here am i true ?

Here previous child is not overidden here we are deleting it by

```
document.getElementById("buttonParent").innerHTML =
```

Yes, you are correct. Let's break down the functionality step by step:

1. **Initial State**:
   - The `state` object is initialized with `count: 0`.
2. **Initial Render**:
   - The `buttonComponentReRender` function is called initially, which:
     - Clears the inner HTML of the `div` with `id="buttonParent"`.
     - Creates a button element using the `buttonComponent` function, passing the current `count` (which is initially 0).
     - Appends this button to the `div`.
3. **Button Click**:
   - When the button is clicked, the `onButtonPress` function is triggered, which:
     - Increments the `count` in the `state` object by 1.
     - Calls `buttonComponentReRender` again.
4. **Re-render on Click**:
   - Each time `buttonComponentReRender` is called:
     - It clears the existing content of the `div` (removing the previous button).
     - Creates a new button with the updated `count`.

Khushal Savalakha

    - Appends this new button to the `div`.

Thus, each button click results in the previous button being removed and replaced with a new button that has the updated count. This ensures that only one button is visible at any time, with the count reflecting the number of times the button has been clicked.

Here's a step-by-step summary of what happens during each button click:

- The button's `onclick` attribute triggers the `onButtonPress` function.
- `onButtonPress` increments the `count` and calls `buttonComponentReRender`.
- `buttonComponentReRender` clears the existing button and appends a new button with the updated count.

So yes, each click on the button will override the previous button with a new one that has an updated count.

> 💡 it means components every time rendering by user it will send the object and it will update on the main page am i true ?

Yes, you are correct. Every time the component is rendered by the user interaction, the object (in this case, the state) is updated, and this updated information is used to render the component on the main page. Here's a more detailed explanation:

1. **Initial State**: The application starts with an initial state defined in the `state` object. For instance, `state = { count: 0 }`.
2. **Initial Rendering**: The `buttonComponentReRender` function is called to render the initial component:
    - It clears the current content of the `div` with `id="buttonParent"`.
    - It creates a new button component using the `buttonComponent` function, passing the current state (`state.count`).
    - This new button is then appended to the `div`.
3. **User Interaction**: When the user clicks the button, the `onButtonPress` function is triggered:
    - This function updates the state by incrementing `state.count`.
    - After updating the state, it calls the `buttonComponentReRender` function to update the DOM with the new state.

Khushal Savalakha

4. **Re-rendering**:
    - The `buttonComponentReRender` function clears the existing button from the `div`.
    - It creates a new button component with the updated state (`state.count`).
    - This new button is appended to the `div`, replacing the old button.

This cycle of updating the state and re-rendering the component ensures that the component on the main page always reflects the current state.

In summary:

- The component's rendering is driven by the state.
- Every time the state changes (due to user interaction or other reasons), the component is re-rendered to reflect the new state.
- The updated component is then displayed on the main page, replacing the previous component.

This approach allows for dynamic and responsive user interfaces, where the view consistently matches the underlying state.

> 💡 it means state is set information like object ?

Yes, the state in this context is an object that holds information about the current status of the application or component. This state object can have various properties that represent different pieces of information or data that the component needs to function correctly.

Here's a recap using the code example you provided:

1. **State Object**:

```
let state = {
  count: 0,
};
```

    - The state is initialized as an object with a property `count` set to `0`.

2. **State Update**:
    - When the button is pressed, the `onButtonPress` function updates the state:

Khushal Savalakha

```
function onButtonPress() {
  state.count++;
  buttonComponentReRender();
}
```

- This increments the `count` property in the state object.

3. **Re-render with State**:

   - The `buttonComponentReRender` function re-renders the component using the updated state:

```
function buttonComponentReRender() {
  document.getElementById("buttonParent").innerHTM
L = "";
  const component = buttonComponent(state.count);
  document.getElementById("buttonParent").appendCh
ild(component);
}
```

   - It clears the previous content and appends a new button with the current state's `count`.

4. **Component Creation Using State**:

   - The `buttonComponent` function creates a button using the current state:

```
function buttonComponent(count) {
  const button = document.createElement("button");
  button.innerHTML = "Counter: " + count;
  button.setAttribute("onclick", "onButtonPress
()");
  return button;
}
```

   - The `count` from the state is used to set the button's inner HTML.

In summary:

- The **state** is an object that stores information about the component's current status.

Khushal Savalakha

- **Updating the state** involves changing properties within this object (e.g., incrementing `count`).
- **Re-rendering** the component involves creating new DOM elements using the updated state and updating the displayed content accordingly.

This pattern of managing state and rendering components based on state changes is a fundamental concept in modern web development frameworks like React, Vue.js, and Angular.

> 💡 it means i made a click button which is giving me count every time i clicked and this is processed in our system and our system id directly managing it but whenever we have to work with outside our system with different database if user gives any question and it will goes to server and server respond it that time react state,component and rendering are useful am i true ?

Yes, you are correct. React's state, components, and rendering are particularly useful in scenarios where your application needs to interact with a server or an external database. This is a common use case in modern web applications where user interactions often result in server requests and responses. Here's how it all fits together:

1. **Local State Management**:
   - State in React components is used to manage local data within the application. For example, maintaining the count of button clicks.
2. **User Interactions**:
   - When a user interacts with the UI (e.g., clicks a button), the state can be updated to reflect these interactions.
   - This update triggers a re-render of the relevant components to display the latest state.
3. **Server Interactions**:
   - In many applications, user interactions may also require communicating with an external server or database. For example, submitting a form, fetching data, etc.
   - React provides mechanisms to handle such interactions using lifecycle methods, hooks (like `useEffect`), or event handlers.

Example Scenario: Submitting a Question to the Server

Khushal Savalakha

Let's consider an example where a user submits a question, and this needs to be sent to the server. Here's how React's state, components, and rendering are useful:

Step-by-Step Process:

1. **Component and State Setup**:
   - Create a component with state to hold the user's input (question) and the response from the server.
2. **User Input**:
   - When the user types a question and clicks a submit button, an event handler captures this input.
3. **Server Request**:
   - The event handler sends the question to the server via an API call (using `fetch`, `axios`, etc.).
   - The server processes the question and sends back a response.
4. **State Update and Re-rendering**:
   - Once the response is received, update the state with the server's response.
   - React re-renders the component to display the response to the user.

Code Example:

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

function QuestionComponent() {
  const [question, setQuestion] = useState('');
  const [response, setResponse] = useState('');

  const handleInputChange = (e) => {
    setQuestion(e.target.value);
  };

  const handleSubmit = async () => {
    try {
      const res = await fetch('<https://api.example.com/
ask>', {
        method: 'POST',
```


Khushal Savalakha

```
        headers: {
          'Content-Type': 'application/json'
        },
        body: JSON.stringify({ question })
      });
      const data = await res.json();
      setResponse(data.answer);
    } catch (error) {
      console.error('Error:', error);
    }
  };

  return (
    <div>
      <input
        type="text"
        value={question}
        onChange={handleInputChange}
        placeholder="Ask your question"
      />
      <button onClick={handleSubmit}>Submit</button>
      {response && <p>Response: {response}</p>}
    </div>
  );
}

export default QuestionComponent;
```

Explanation:

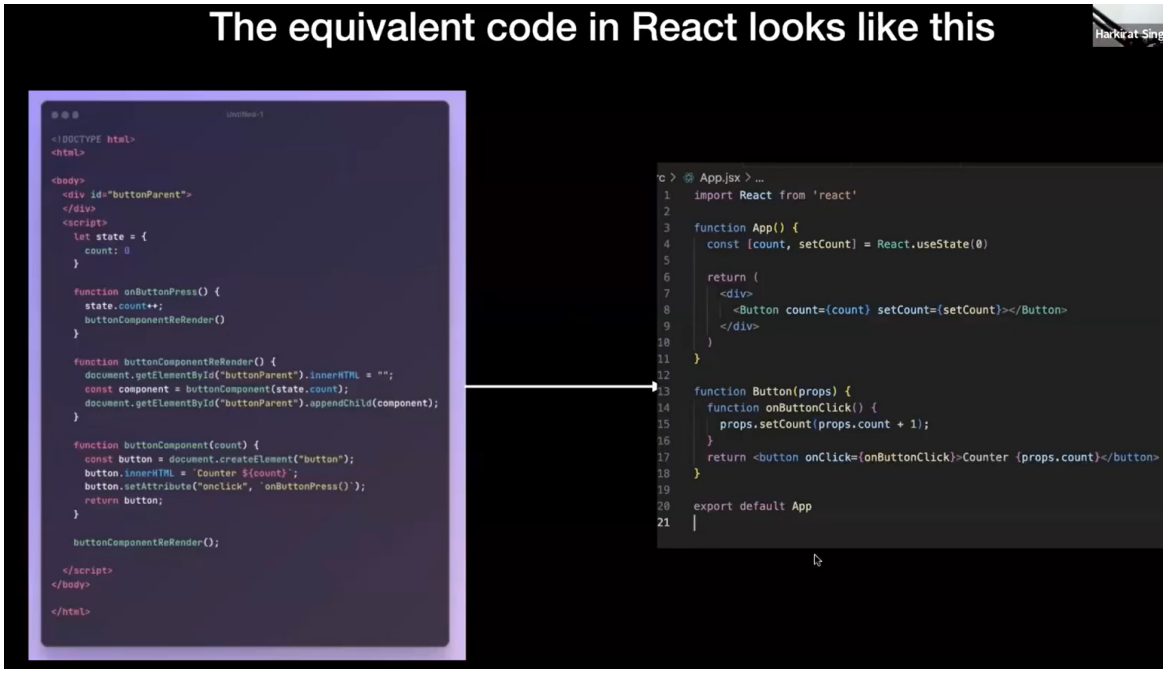
1. **State Management**:
   - `question` state holds the user's input.
   - `response` state holds the server's response.
2. **User Input Handling**:
   - `handleInputChange` updates the `question` state as the user types.
3. **Server Request**:

Khushal Savalakha

- `handleSubmit` sends the user's question to the server.
- Upon receiving the response, it updates the `response` state.

4. **Re-rendering**:
   - When the state is updated, React re-renders the component to display the updated response.

In this way, React's state management, component structure, and rendering capabilities make it effective for building interactive UIs that communicate with external servers and databases, providing a seamless user experience.



▼ Components

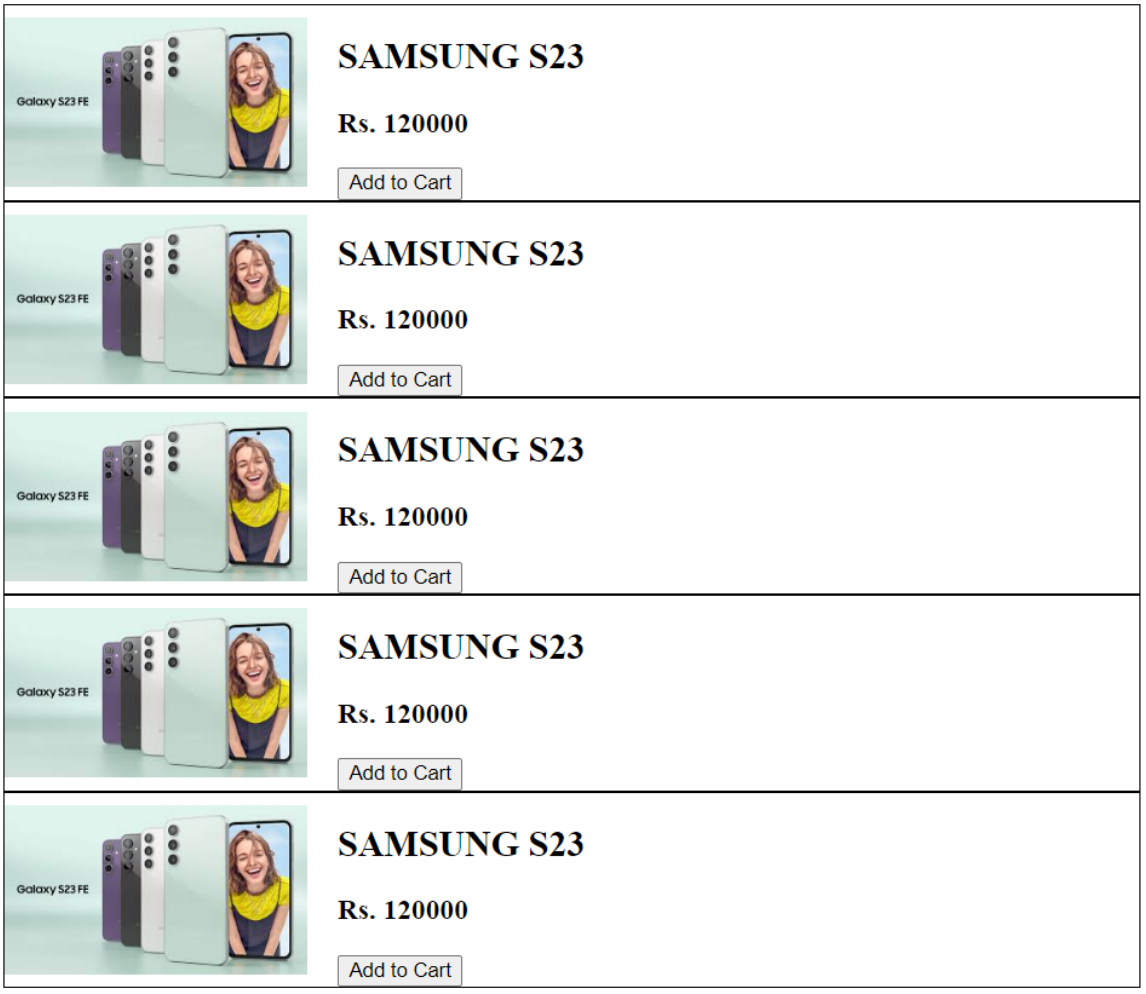
```
import React from 'react'
import Mobile_List from './Mobile_List'

export default function Mobile() {
  return (
    <div>
      <Mobile_List/>
      <Mobile_List/>
      <Mobile_List/>
      <Mobile_List/>
      <Mobile_List/>
    </div>
```


Khushal Savalakha

```
  )
}
//Here we don't have to write 1 thing too many times to dis
```

```
import React from "react";
import "./Mobile_List.css";
export default function Mobile_List() {
  const image = (
    <img
      src="https://encrypted-tbn0.gstatic.com/images?q=tbn:
      alt="sorry"
      className="image"
    ></img>
  );
  const img_url =
    "https://encrypted-tbn0.gstatic.com/images?q=tbn:ANd9Gc
  return (
    <div className="main">
      {/* {image} */}
      <img src={img_url} alt="sorry" className="image"></im
      <div>
        <h2>SAMSUNG S23</h2>
        <h3>Rs. 120000</h3>
        <button>Add to Cart</button>
      </div>
    </div>
  );
}
```


Khushal Savalakha




```
import React from 'react'

export default function T3_practice() {
    let data="h1"
  return (
    <div>
      <{data}>Hello<{data}/>
    </div>
  )
}

above code gives error
 return (
   6 |      <div>
>  7 |        <{data}>Hello<{data}/>
```


Khushal Savalakha

```
     |         ^
   8 |     </div>
   9 |   )
  10 | }

explain me why it gives error ?
```

In JSX, tags like `<h1>`, `<div>`, etc., need to be known and defined at compile-time (the time when the code is being compiled) because JSX gets transformed into JavaScript function calls by a compiler (like Babel). This transformation requires knowing exactly what HTML tags or React components to generate in the final output.

When you use a dynamic variable like `{data}` within the JSX tags:

```
<{data}>Hello<{data}/>
```

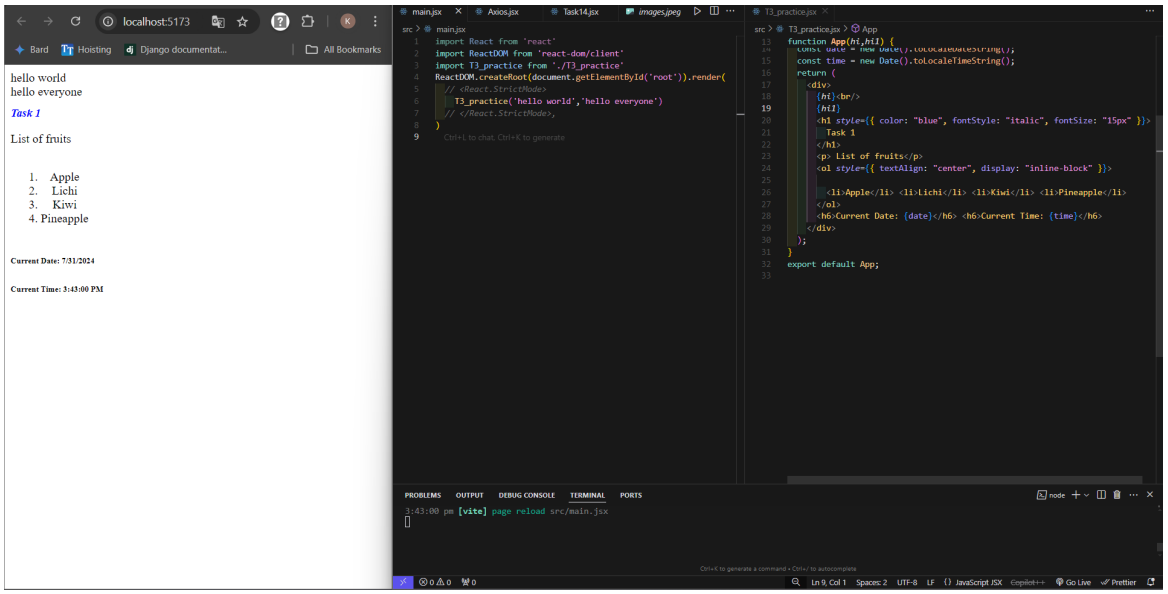
the compiler doesn't know what `<{data}>` should be converted into because `{data}` could be anything (e.g., a string, a number, an object) and it only gets its value at runtime (the time when the code is actually running in the browser). Since the compiler can't predict the value of `{data}`, it throws an error.

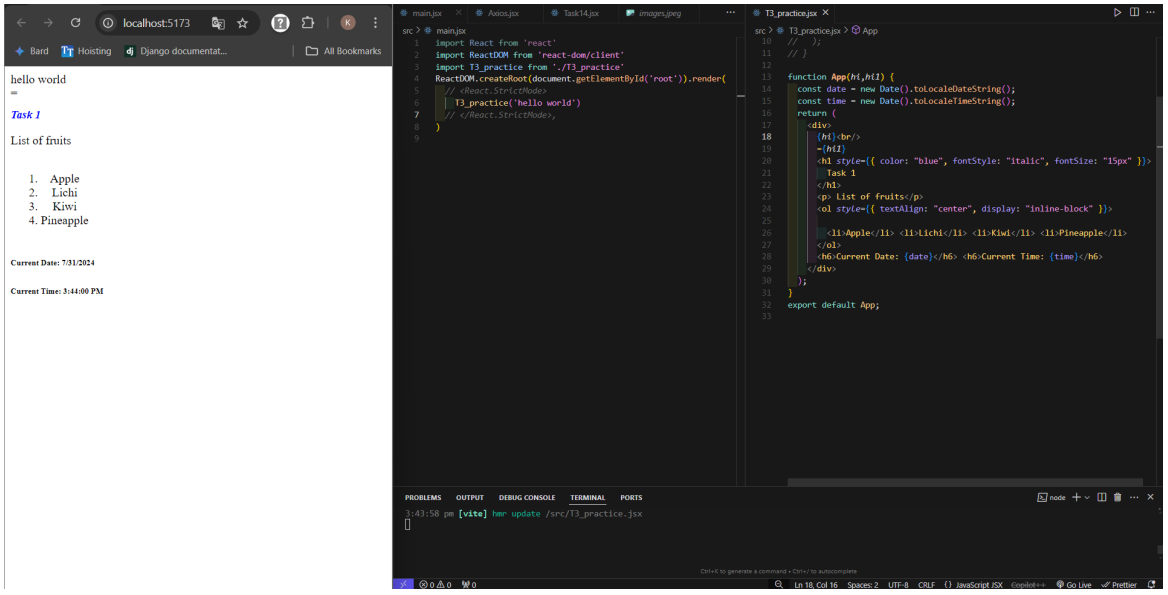
However, you can assign the tag name to a variable and then use that variable in the JSX, because this keeps the dynamic part in the JavaScript layer where it's evaluated at runtime:

```
import React from 'react';

export default function T3_practice() {
  let data = "h1";
  const Tag = data;  // Tag will be evaluated at runtime

  return (
    <div>
      <Tag>Hello</Tag>  // JSX now understands what Tag
is at runtime
    </div>
  );
}
```


Khushal Savalakha

In this way, `Tag` is evaluated at runtime and JSX can correctly create the desired element. This method works because `Tag` is now a variable containing the string `"h1"` which JSX can use to create the appropriate HTML element when the code runs.





▼ Props

```
import React from 'react'
import Mobile_List from './Mobile_List'

export default function Mobile() {
  return (
    <div>
      <Mobile_List greed="Hello Everyone"/>
```


Khushal Savalakha

```
    </div>
  )
}
```

```
import React from "react";
import "./Mobile_List.css";
export default function Mobile_List(props) {
  // const image = (
  //   <img
  //     src="https://encrypted-tbn0.gstatic.com/images?q=t
  //     alt="sorry"
  //     className="image"
  //   ></img>
  // );
  const img_url =
    "https://encrypted-tbn0.gstatic.com/images?q=tbn:ANd9Gc
  return (
    <div className="main">
        {console.log(props)}
        {props.greed}
      {/* {image} */}
      <img src={img_url} alt="sorry" className="image"></im
      <div>
        <h2>SAMSUNG S23</h2>
        <h3>Rs. 120000</h3>
        <button>Add to Cart</button>
      </div>
    </div>
  );
}
```

Hello Everyone

SAMSUNG S23

Rs. 120000

Add to Cart

Khushal Savalakha

In this React code snippet, `props` are being used to pass data from a parent component (`Mobile`) to a child component (`Mobile_List`).

In the `Mobile` component, the `<Mobile_List greed="Hello Everyone"/>` line passes a prop named `greed` with the value `"Hello Everyone"` to the `Mobile_List` component.

In the `Mobile_List` component, `props` is an object that contains all the properties passed from the parent component (`Mobile`). In this case, it contains the `greed` property with the value `"Hello Everyone"`.

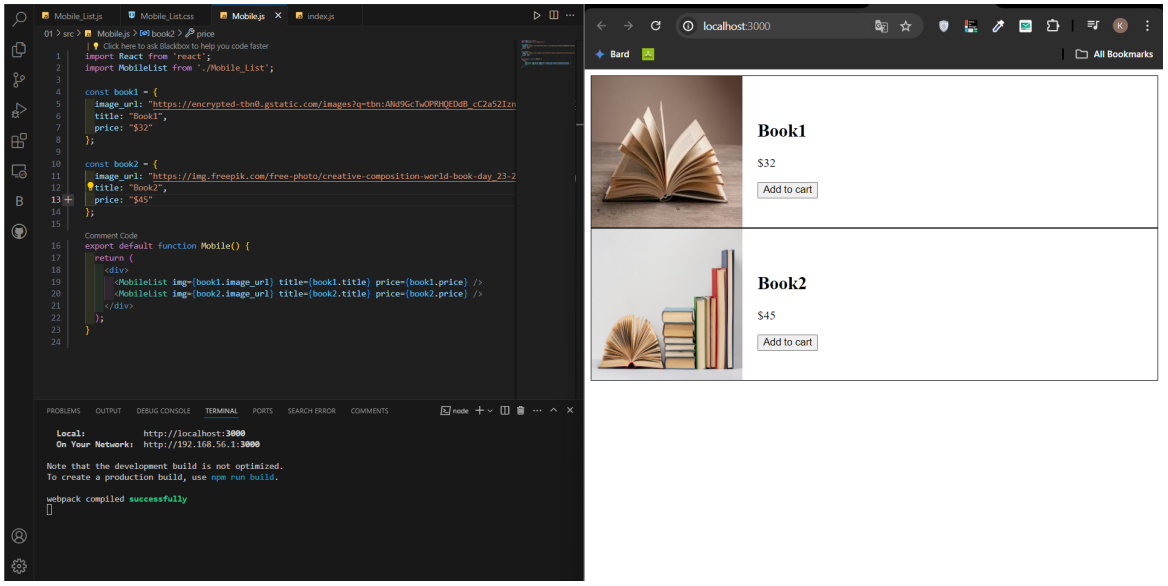
So, `props.greed` in the `Mobile_List` component refers to the value `"Hello Everyone"` that was passed from the `Mobile` component. This allows components to communicate and share data in a React application.

```
import React from 'react';
import MobileList from './Mobile_List';

const book1 = {
  image_url: "https://encrypted-tbn0.gstatic.com/images?q=t
  title: "Book1",
  price: "$32"
};

const book2 = {
  image_url: "https://img.freepik.com/free-photo/creative-c
  title: "Book2",
  price: "$45"
};

export default function Mobile() {
  return (
    <div>
      <MobileList img={book1.image_url} title={book1.title}
      <MobileList img={book2.image_url} title={book2.title}
    </div>
  );
}
```

```
import React from 'react';
import './Mobile_List.css';
```


Khushal Savalakha

```
export default function Mobile_List(props) {
  return (
    <div className='main'>
      <img src={props.img} alt='sorry'></img>
      <div>
      <h2>{props.title}</h2>
      <p>{props.price}</p>
      <button>Add to cart</button>
      </div>
    </div>
  )
}
```



or

```
import React from 'react';
import './Mobile_List.css';

export default function Mobile_List(props) {
  const { img, title, price } = props; // Object destructur

  return (
```


Khushal Savalakha

```
    <div className='main'>
      <img src={img} alt='sorry' />
      <div>
        <h2>{title}</h2>
        <p>{price}</p>
        <button>Add to cart</button>
      </div>
    </div>
  );
}
```

Sure, you can create an array of book objects and then iterate over it to render each book's data using the `MobileList` component. Here's how you can do it:

```
import React from 'react';
import MobileList from './Mobile_List';

// Array of book objects
const books = [
  {
    image_url: "<https://encrypted-tbn0.gstatic.com/imag
es?q=tbn:ANd9GcTwOPRHQEDdB_cC2a52Izn5Gd6aKLQzGIr3mj_-BXK
0ig&s>",
    title: "Book1",
    price: "$32"
  },
  {
    image_url: "<https://img.freepik.com/free-photo/crea
tive-composition-world-book-day_23-2148883765.jpg>",
    title: "Book2",
    price: "$45"
  },
  // Add more book objects here
];

export default function Mobile() {
  return (
    <div>
```


Khushal Savalakha

```
      {books.map((book, index) => (
        <MobileList key={index} img={book.image_url} tit
le={book.title} price={book.price} />
      ))}
    </div>
  );
}
```

In this code:

- We create an array called `books` containing multiple book objects.
- We use the `map` function to iterate over each book object in the `books` array.
- For each book object, we render a `MobileList` component with the book's `image_url`, `title`, and `price` passed as props.
- Each `MobileList` component is given a unique `key` prop, which is necessary when rendering lists of components in React. Here, we use the index of each book as the key.

USING JSON FILE

```
[
    {
      "image_url": "https://encrypted-tbn0.gstatic.com/imag
      "title": "Book1",
      "price": "$32"
    },
    {
      "image_url": "https://img.freepik.com/free-photo/crea
      "title": "Book2",
      "price": "$45"
    }
  ]
```

```
import React from 'react';
import MobileList from './Mobile_List';
import books from './books.json'
```


Khushal Savalakha

```
export default function Mobile() {
  return (
    <div>
      {books.map(book => (
        <MobileList  img={book.image_url} title={book.title
      ))}
    </div>
  );
}
```

The `map()` function in JavaScript is a built-in array method that creates a new array populated with the results of calling a provided function on every element in the calling array. It does not change the original array.

Full Syntax of `map()`

```
array.map(callback(currentValue, index, array), thisAr
g);
```

Parameters:

1. **callback:** A function that is called for every element of the array. Each time `callback` executes, the returned value is added to the new array.
   - **currentValue:** The current element being processed in the array.
   - **index (optional):** The index of the current element being processed in the array.
   - **array (optional):** The array `map` was called upon.
2. **thisArg (optional):** Value to use as `this` when executing `callback`.

Example of `map()`

Here's a simple example to illustrate how `map()` works:

```
const numbers = [1, 2, 3, 4, 5];

// Using map to create a new array by multiplying each e
lement by 2
const doubled = numbers.map((number, index, array) => {
  console.log(`Processing index: ${index}, value: ${numb
```


Khushal Savalakha

```
er}, in array: ${array}`);
  return number * 2;
});

console.log(doubled); // Output: [2, 4, 6, 8, 10]
```

Example in Your React Code

In your React code, `map()` is used to iterate over the `books` array and render a `MobileList` component for each book object.

```
import React from 'react';
import MobileList from './Mobile_List';

// Array of book objects
const books = [
  {
    image_url: "<https://encrypted-tbn0.gstatic.com/imag
es?q=tbn:ANd9GcTwOPRHQEDdB_cC2a52Izn5Gd6aKLQzGIr3mj_-BXK
0ig&s>",
    title: "Book1",
    price: "$32"
  },
  {
    image_url: "<https://img.freepik.com/free-photo/crea
tive-composition-world-book-day_23-2148883765.jpg>",
    title: "Book2",
    price: "$45"
  },
  // Add more book objects here
];

export default function Mobile() {
  return (
    <div>
      {books.map((book, index) => (
        <MobileList key={index} img={book.image_url} tit
le={book.title} price={book.price} />
      ))}
```

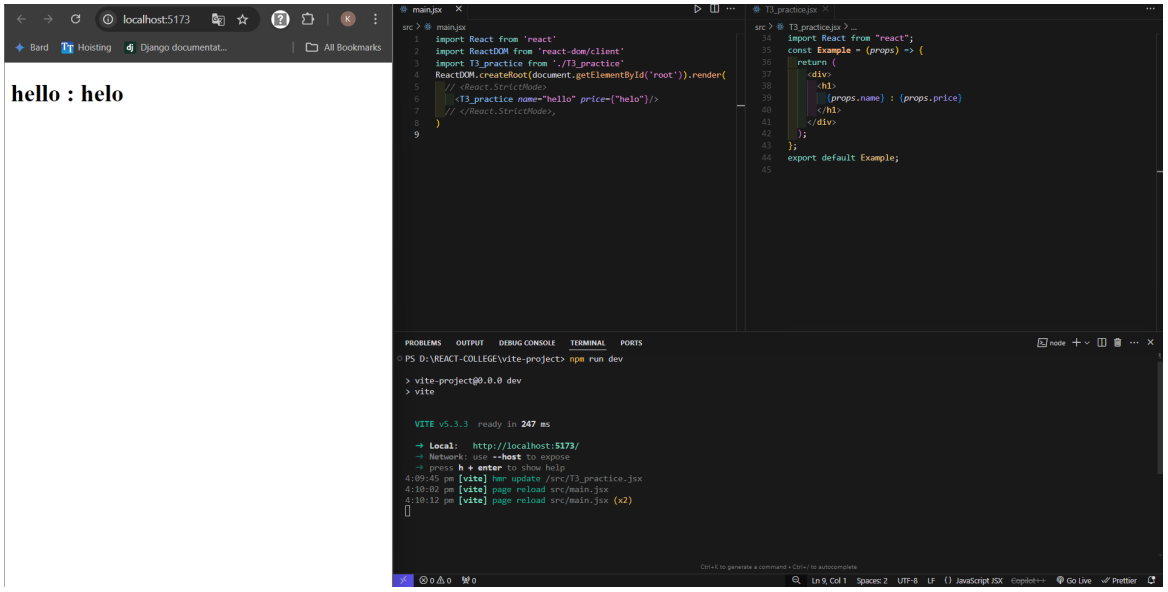

Khushal Savalakha

```
      </div>
    );
  }
```

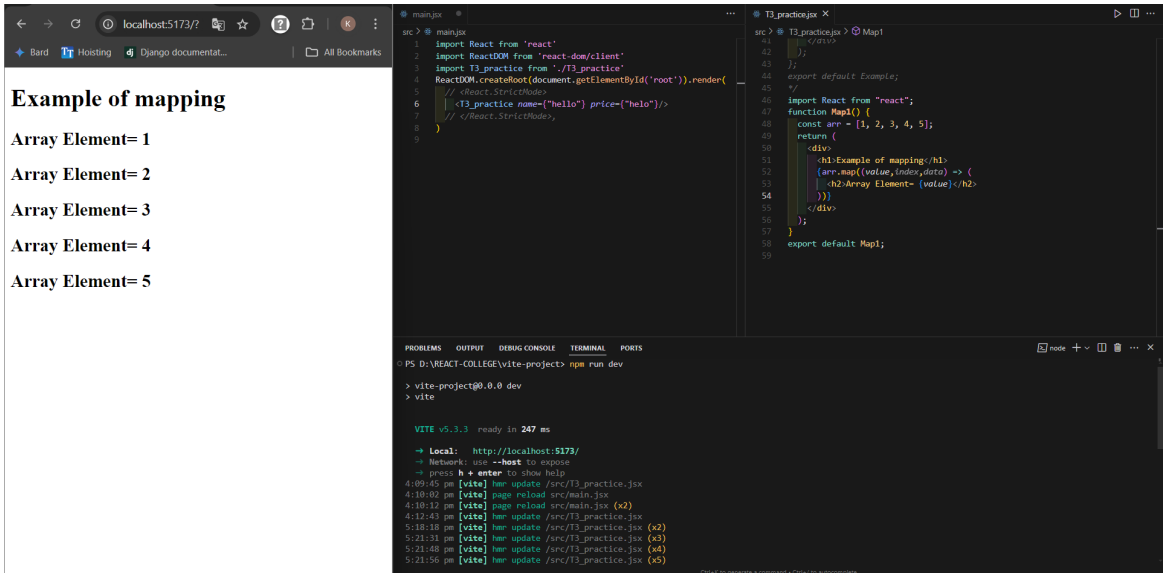
In this code:

- `books.map()` iterates over each book object in the `books` array.
- For each book, it returns a `MobileList` component.
- The `key` prop is used to uniquely identify each element in the list.
- `img`, `title`, and `price` are passed as props to the `MobileList` component.

This use of `map()` is common in React for rendering lists of components dynamically based on an array of data.



Khushal Savalakha


localhost:5173/?
Bard
Hoisting
Django documentat...
All Bookmarks
Example of mapping
Array Element= 1
Array Element= 2
Array Element= 3
Array Element= 4
Array Element= 5
main.jsx
import React from 'react'
import ReactDOM from 'react-dom/client'
import T3_practice from './T3_practice'
ReactDOM.createRoot(document.getElementById('root')).render(
// <React.StrictMode>
<T3_practice name={"hello"} price={"helo"}/>
// </React.StrictMode>,
)
T3_practice.jsx

);
};
export default Example;
*/
import React from "react";
function Map1() {
const arr = [1, 2, 3, 4, 5];
return (

<h1>Example of mapping</h1>
{arr.map((value,index,data) => (
<h2>Array Element= {value}</h2>
))}

);
}
export default Map1;
PROBLEMS
OUTPUT
DEBUG CONSOLE
TERMINAL
PORTS
PS D:\REACT-COLLEGE\vite-project> npm run dev
> vite-project@0.0.0 dev
> vite
VITE v5.3.3 ready in 247 ms
Local: http://localhost:5173/
Network: use --host to expose
press h + enter to show help
4:09:45 pm [vite] hmr update /src/T3_practice.jsx
4:10:02 pm [vite] page reload src/main.jsx
4:10:12 pm [vite] page reload src/main.jsx (x2)
4:12:43 pm [vite] hmr update /src/T3_practice.jsx
5:18:18 pm [vite] hmr update /src/T3_practice.jsx (x2)
5:21:31 pm [vite] hmr update /src/T3_practice.jsx (x3)
5:21:48 pm [vite] hmr update /src/T3_practice.jsx (x4)
5:21:56 pm [vite] hmr update /src/T3_practice.jsx (x5)

```
/*function map() {
const arr = [1, 2, 3, 4, 5, 3, 6, 4, 3, 1];
return (
<div>
<h1>Example of mapping with condition</h1>
{arr.map((value) => {
if (value > 3) {
return <h2>Array Elements= {value}</h2>;
}
})}
</div>
);
}
export default map;
/
```

```
import React from 'react';

function Map() {
const arr = [1, 2, 3, 4, 5, 3, 6, 4, 3, 1];

return (
<div>
<h1>Example of mapping with condition</h1>
{arr.map((value, index) => (
(value > 3) && <h2 key={index}>Array Elements = {value}</h2>
))}
</div>
);
}

export default Map;
```

write difference between both code with proper explaination in table form .also write code in table.

Certainly! Here is a table comparing the two code snippets along with their respective explanations:

Aspect	Original Code	Updated Code

Khushal Savalakha

Syntax	`function map() { const arr = [1, 2, 3, 4, 5, 3, 6, 4, 3, 1];  return (  <div>  <h1>Example of mapping with condition</h1>  {arr.map((value) => {  if (value > 3) {  return <h2>Array Elements= {value}</h2>;  }  })} </div>  ); } export default map;`	`import React from 'react';  function Map() { const arr = [1, 2, 3, 4, 5, 3, 6, 4, 3, 1];   return (  <div>  <h1>Example of mapping with condition</h1>  {arr.map((value, index) => (  (value > 3) && <h2 key={index}>Array Elements = {value}</h2>  ))} </div>  ); }  export default Map;`
Conditional Rendering	Uses an `if` statement inside the `.map()` callback function to conditionally return an element.	Uses the logical AND (`&&`) operator to conditionally render elements based on the condition.
Return Statements	Explicitly uses `return` inside the `if` block to render elements.	Uses short-circuit evaluation to implicitly return the element when the condition is met.
Handling Unmet Conditions	Does not explicitly handle the case where the condition is not met; implicitly returns `undefined`.	Returns `false` (which React treats as nothing to render) when the condition is not met.
Code Structure	More verbose, with explicit `return` inside a block.	More concise, with a single-line conditional rendering expression.
Key Prop	Does not use a key prop, which can cause issues with React's list reconciliation.	Includes a `key` prop to uniquely identify each rendered element, improving React's performance and reducing potential bugs.

Code Comparison:

```
function map() {
  const arr = [1, 2, 3, 4, 5, 3, 6, 4, 3, 1];
  return (
    <div>
      <h1>Example of mapping with condition</h1>
      {arr.map((value) => {
        if (value > 3) {
          return <h2>Array Elements= {value}</h2>;
        }
      })}
```


Khushal Savalakha

```
    </div>
  );
}
export default map;
```

```
import React from 'react';

function Map() {
  const arr = [1, 2, 3, 4, 5, 3, 6, 4, 3, 1];

  return (
    <div>
      <h1>Example of mapping with condition</h1>
      {arr.map((value, index) => (
        (value > 3) && <h2 key={index}>Array Elements = {va
      ))}
    </div>
  );
}

export default Map;
```

Detailed Differences:

- **Conditional Rendering**:
  - **Original Code**: Uses `if` statement to check the condition inside `.map()`.
  - **Updated Code**: Uses the `&&` operator to conditionally render elements.
- **Return Statements**:
  - **Original Code**: Explicitly returns the JSX element when the condition is met.
  - **Updated Code**: Implicitly returns the JSX element using short-circuit evaluation.
- **Handling Unmet Conditions**:
  - **Original Code**: Implicitly returns `undefined` if the condition is not met.
  - **Updated Code**: Uses `&&` to ensure that `false` is returned when the condition is not met, which React ignores in rendering.

Khushal Savalakha

- **Code Structure**:
  - **Original Code**: More verbose due to the explicit `if` and `return` statements.
  - **Updated Code**: More concise, leveraging JavaScript's short-circuit evaluation.
- **Key Prop**:
  - **Original Code**: Does not include a `key` prop.
  - **Updated Code**: Includes a `key` prop for better React performance and avoiding potential rendering issues.

Certainly! The `ArrayMap_condition` component uses both the `.filter()` and `.map()` functions to conditionally render elements based on an array of values. Here's a step-by-step explanation of how it works:

Code Breakdown:

```
const ArrayMap_condition = () => {
  const arr1 = [1, 2, 3, 4, 5, 6, 7, 8, 9];

  return (
    <div>
      <h1>using map/filter function</h1>
      {
        arr1.filter((value) => value <= 6).map((value) =
> {
          return <h1 key={value}>array values = {value}
</h1>;
        })
      }
    </div>
  );
}

export default ArrayMap_condition;
```

Explanation:

1. **Array Declaration**:

Khushal Savalakha

```
const arr1 = [1, 2, 3, 4, 5, 6, 7, 8, 9];
```

- `arr1` is an array containing numbers from 1 to 9.

2. **Filter Operation**:

```
arr1.filter((value) => value <= 6)
```

- The `.filter()` method creates a new array containing only the elements that satisfy the condition provided in the callback function.
- In this case, `value <= 6` is the condition, so only the numbers less than or equal to 6 will be included in the new array.
- The result of this filter operation will be `[1, 2, 3, 4, 5, 6]`.

3. **Map Operation**:

```
.map((value) => {
  return <h1 key={value}>array values = {value}</h1>;
})
```

- The `.map()` method creates a new array by applying the provided function to each element of the array generated by the `.filter()` method.
- For each value in the filtered array `[1, 2, 3, 4, 5, 6]`, a new `<h1>` element is created with the text `"array values = {value}"`.
- The `key` prop is added to each `<h1>` element to uniquely identify each element in the list, which helps React manage updates more efficiently.

4. **Rendering**:

```
<div>
  <h1>using map/filter function</h1>
  {
    arr1.filter((value) => value <= 6).map((value) =>
{
      return <h1 key={value}>array values = {value}</
h1>;
    })
```


Khushal Savalakha

```
  }
</div>
```

- The `return` statement renders a `div` containing an `<h1>` header and the list of `<h1>` elements generated by the `.filter()` and `.map()` operations.
- The header `<h1>using map/filter function</h1>` is displayed at the top, followed by the filtered and mapped array values.

Summary:

- **Filtering**: The `.filter()` method is used to select elements from the array that meet a specific condition (values less than or equal to 6).
- **Mapping**: The `.map()` method is used to transform each element of the filtered array into a React component (`<h1>` element).
- **Rendering**: The transformed elements are then rendered inside a `div` element in the component.

This approach is efficient for displaying a subset of data and transforming it into a desired format for rendering.

▼ {} use

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

export default function Mobile() {
  const books = [
    {
      image_url:
        "https://encrypted-tbn0.gstatic.com/images?q=tbn
      title: "Book1",
      price: "$32",
    },
    {
      image_url:
        "https://img.freepik.com/free-photo/creative-com
      title: "Book2",
      price: "$45",
```


Khushal Savalakha

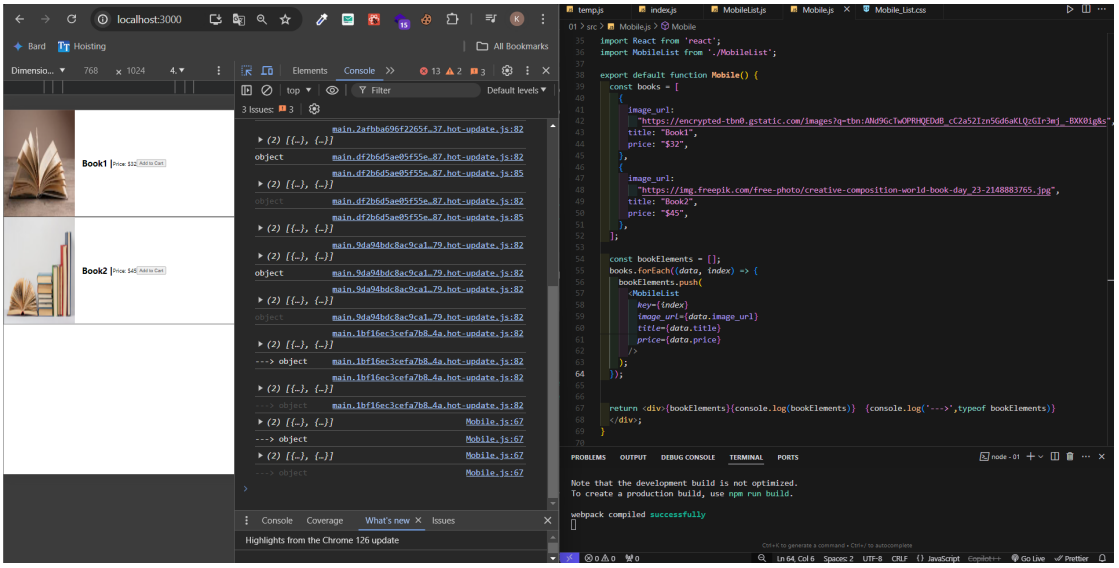
```
    },
  ];

  const bookElements = [];
  books.forEach((data, index) => {
    bookElements.push(
      <MobileList
        key={index}
        image_url={data.image_url}
        title={data.title}
        price={data.price}
      />
    );
  });


  return <div>{bookElements}{console.log(bookElements)}
  </div>;
}
```

```
import React from "react";
import "./Mobile_List.css";

function Book({ image_url, title, price }) {
  return (
    <div className="main">
      <img
        src={image_url}
        alt="sorry"
        style={{ height: "300px", width: "300px" }}
      ></img>
      <h2>{title} |</h2>
      <p>Price: {price}</p>
      <button>Add to Cart</button>
    </div>
  );
}
```


Khushal Savalakha

```
export default Book;
```

```
.main{
    display:flex;
    align-items:center;
    border:1px solid black;
    flex-direction: row;
}

img{
    flex:0 0 500px;
    height:200px;
    width:200px;
    margin-right:20px;
    display: block;
    max-width:200px;
}
```



Khushal Savalakha

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

export default function Mobile() {
  const books = [
    {
      image_url:
        "https://encrypted-tbn0.gstatic.com/images?q=tbn:ANd9GcTwOPRHQEDdB_cC2a52Izn5Gd6aKLQzGIr3mj
      title: "Book1",
      price: "$32",
    },
    {
      image_url:
        "https://img.freepik.com/free-photo/creative-composition-world-book-day_23-2148883765.jpg",
      title: "Book2",
      price: "$45",
    },
  ];

  const bookElements = [];
  books.forEach(({data, index) => {
    bookElements.push(
      <MobileList
        key={index}
        image_url={data.image_url}
        title={data.title}
        price={data.price}
      />
    );
  });


  return <div>{bookElements}{console.log(bookElements)}  {console.log('--->',typeof bookElements)}
  </div>;
}
```

```
import React from "react";
import "./Mobile_list.css";

function Book({ image_url, title, price }) {
  return (
    <div className="main">
      <img
        src={image_url}
        alt="sorry"
        style={{ height: "300px", width: "300px" }}
      ></img>
      <h2>{title} |</h2>
      <p>Price: {price}</p>
      <button>Add to Cart</button>
    </div>
  );
}

export default Book;
```

```
.main{
    display:flex;
    align-items:center;
    border:1px solid black;
    flex-direction: row;
}

img{
    flex:0 0 500px;
    height:200px;
    width:200px;
    margin-right:20px;
    display: block;
    max-width:200px;
}
```

```
PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL   PORTS

Note that the development build is not optimized.
To create a production build, use npm run build.

webpack compiled successfully
```

JSX in Return Statement: When you include `{bookElements}` inside the `return` statement, JSX processes the array and renders each `MobileList` component as a child of the `<div>`.

here i don't understand what is jsx and how it renders each MobileList component as child of div?

Sure, let's break it down step by step.

What is JSX?

- **JSX (JavaScript XML)**: JSX is a syntax extension for JavaScript, commonly used with React to describe what the UI should look like. It looks similar to HTML but is written within JavaScript.
- **Purpose of JSX**: It allows you to write HTML-like code inside your JavaScript files, making it easier to visualize the structure of your components.

How JSX Works

1. **JSX Syntax**:

   ```
   const element = <h1>Hello, world!</h1>;
   ```

   This looks like HTML but is actually JavaScript. Under the hood, this JSX code is transformed into regular JavaScript objects.

2. **Transformation**:

Khushal Savalakha

```
const element = React.createElement('h1', null, 'He
llo, world!');
```

JSX is compiled to `React.createElement` calls which create JavaScript objects representing elements.

Rendering Elements with JSX

When you write a component that uses JSX to render other components, React processes and renders these components as described. Let's look at your example in detail:

Example Breakdown

1. The `Mobile.js` Component:

- **Books Array**: You have an array of book objects:

```
const books = [
  { image_url: "url1", title: "Book1", price: "$32"
},
  { image_url: "url2", title: "Book2", price: "$45"
}
];
```

- **Mapping to Create `MobileList` Components**:

```
const bookElements = books.map((data, index) => (
  <MobileList
    key={index}
    image_url={data.image_url}
    title={data.title}
    price={data.price}
  />
));
```

This `map` function iterates over each item in the `books` array and creates a `MobileList` component for each item, passing the relevant data as props.

- **Returning the JSX**:

Khushal Savalakha

```
return <div>{bookElements}</div>;
```

This line is where JSX shines. Let's break it down:

How JSX Renders the Components

1. **Inside the `return` Statement**:

```
return <div>{bookElements}</div>;
```

   - The `<div>` is a JSX element.
   - `{bookElements}` is a JavaScript expression. In JSX, you can embed any JavaScript expression by wrapping it in curly braces `{}`.

2. **What `{bookElements}` Contains**:

   - After the `map` function runs, `bookElements` is an array of `MobileList` components:

```
[
  <MobileList key={0} image_url="url1" title="Bo
ok1" price="$32" />,
  <MobileList key={1} image_url="url2" title="Bo
ok2" price="$45" />
]
```

3. **Embedding the Array**:

   - When you include `{bookElements}` inside the JSX, React understands that you want to render each item in the `bookElements` array.
   - React will render each `MobileList` component as a child of the `<div>`.

Visual Explanation

1. **Books Array**:

```
const books = [
  { image_url: "url1", title: "Book1", price: "$32"
},
  { image_url: "url2", title: "Book2", price: "$45"
```


Khushal Savalakha

```
}
];
```

2. **Mapping to Create Components**:

```
const bookElements = books.map((data, index) => (
  <MobileList
    key={index}
    image_url={data.image_url}
    title={data.title}
    price={data.price}
  />
));
```

This creates an array of `MobileList` components:

```
[
  <MobileList key={0} image_url="url1" title="Book
1" price="$32" />,
  <MobileList key={1} image_url="url2" title="Book
2" price="$45" />
]
```

3. **Rendering in JSX**:

```
return <div>{bookElements}</div>;
```

When React encounters `{bookElements}`, it processes this array and renders each `MobileList` component as a child of the `<div>`. The result is:

```
<div>
  <div class="main">
    <img src="url1" alt="sorry" style="height: 300p
x; width: 300px;" />
    <h2>Book1</h2>
    <p>Price: $32</p>
    <button>Add to Cart</button>
  </div>
```


Khushal Savalakha

```
  <div class="main">
    <img src="url2" alt="sorry" style="height: 300p
x; width: 300px;" />
    <h2>Book2</h2>
    <p>Price: $45</p>
    <button>Add to Cart</button>
  </div>
</div>
```

Summary

- **JSX**: Allows you to write HTML-like code in JavaScript files.
- **Embedding Expressions**: You can embed JavaScript expressions (like arrays) inside JSX using `{}`.
- **Rendering Arrays**: When you embed an array of components inside JSX, React renders each component in the array as if you wrote them out manually.
- **Result**: Each `MobileList` component receives its own props and renders as a child of the `<div>`.

So, React takes care of iterating over the array of components (`bookElements`) and renders them individually as children of the `<div>` in your JSX.

💡 it means {} is the symbol of jsx ?

Yes, in the context of JSX (JavaScript XML), curly braces `{}` are used to embed JavaScript expressions within JSX. This allows you to include dynamic content, such as variables, function calls, or expressions, directly inside your JSX markup. Here's a detailed explanation of how `{}` works in JSX:

Embedding JavaScript Expressions in JSX

1. **Using Variables**:
   You can include JavaScript variables within your JSX using `{}`.

   ```
   const name = "John";
   return <h1>Hello, {name}!</h1>;
   ```


Khushal Savalakha

This will render:

```
<h1>Hello, John!</h1>
```

2. **Using Expressions**:
   You can perform calculations or call functions within `{}`.

```
const age = 30;
return <p>Next year, you will be {age + 1} years ol
d.</p>;
```

   This will render:

```
<p>Next year, you will be 31 years old.</p>
```

3. **Rendering Arrays**:
   When you pass an array of elements inside `{}`, React will render each element.

```
const items = ["Apple", "Banana", "Cherry"];
return (
  <ul>
    {items.map((item, index) => (
      <li key={index}>{item}</li>
    ))}
  </ul>
);
```

   This will render:

```
<ul>
  <li>Apple</li>
  <li>Banana</li>
  <li>Cherry</li>
</ul>
```

Example in Your Context

Khushal Savalakha

Let's revisit your example to see how `{}` is used to embed the array of components:

1. **Books Array**:

```
const books = [
  { image_url: "url1", title: "Book1", price: "$32"
},
  { image_url: "url2", title: "Book2", price: "$45"
}
];
```

2. **Mapping to Create Components**:

```
const bookElements = books.map((data, index) => (
  <MobileList
    key={index}
    image_url={data.image_url}
    title={data.title}
    price={data.price}
  />
));
```

   Here, `bookElements` is an array of `MobileList` components.

3. **Embedding the Array in JSX**:

```
return <div>{bookElements}</div>;
```

   The curly braces `{}` allow you to embed the `bookElements` array directly within the JSX. React processes this array and renders each `MobileList` component inside the `<div>`.

Summary

- **Curly Braces `{}` in JSX**: They are used to embed JavaScript expressions within JSX. This can include variables, expressions, function calls, and arrays.
- **Dynamic Content**: The use of `{}` allows JSX to be dynamic and responsive to changes in data or state.

Khushal Savalakha

- **Rendering Arrays**: When you embed an array of elements within `{}`, React will iterate over the array and render each element as part of the JSX.

Practical Example

Let's put it all together with a simplified example:

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

export default function Mobile() {
  const books = [
    { image_url: "url1", title: "Book1", price: "$32"
},
    { image_url: "url2", title: "Book2", price: "$45"
}
  ];

  const bookElements = books.map((data, index) => (
    <MobileList
      key={index}
      image_url={data.image_url}
      title={data.title}
      price={data.price}
    />
  ));

  return <div>{bookElements}</div>;
}
```

In this example:

- `{bookElements}` is the embedded JavaScript expression.
- `bookElements` is an array of `MobileList` components created from the `books` array.
- React renders each `MobileList` component as a child of the `<div>`.

This demonstrates how JSX and curly braces `{}` work together to enable dynamic and flexible UI rendering in React.

Khushal Savalakha

💡 it means {} is to use js,html and XMl data in at one place am i true ?

Yes, that's essentially correct. Curly braces `{}` in JSX allow you to embed and execute JavaScript expressions within your HTML-like JSX syntax. This capability enables you to dynamically render content based on your JavaScript code, making it a powerful tool for building interactive and dynamic user interfaces. Here's a concise summary:

Purpose of `{}` in JSX

- **Embedding JavaScript Expressions**: You can include variables, expressions, function calls, and other JavaScript logic within your JSX markup.
- **Dynamic Content Rendering**: This allows the content to be dynamic and responsive to changes in state, props, or other JavaScript data.
- **Integration with HTML/XML-like Syntax**: JSX allows you to write HTML/XML-like syntax directly within JavaScript, making it easier to visualize the structure of your components.

Examples

1. **Using Variables**:

```
const name = "John";
return <h1>Hello, {name}!</h1>;
```

Renders:

```
<h1>Hello, John!</h1>
```

2. **Using Expressions**:

```
const age = 30;
return <p>Next year, you will be {age + 1} years ol
d.</p>;
```

Renders:

Khushal Savalakha

```
<p>Next year, you will be 31 years old.</p>
```

3. **Calling Functions**:

```
function getGreeting(name) {
  return `Hello, ${name}!`;
}
return <h1>{getGreeting('John')}</h1>;
```

Renders:

```
<h1>Hello, John!</h1>
```

4. **Rendering Arrays**:

```
const items = ["Apple", "Banana", "Cherry"];
return (
  <ul>
    {items.map((item, index) => (
      <li key={index}>{item}</li>
    ))}
  </ul>
);
```

Renders:

```
<ul>
  <li>Apple</li>
  <li>Banana</li>
  <li>Cherry</li>
</ul>
```

Integration Example from Your Code

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


Khushal Savalakhia

```
export default function Mobile() {
  const books = [
    { image_url: "url1", title: "Book1", price: "$32"
},
    { image_url: "url2", title: "Book2", price: "$45"
}
  ];

  const bookElements = books.map((data, index) => (
    <MobileList
      key={index}
      image_url={data.image_url}
      title={data.title}
      price={data.price}
    />
  ));

  return <div>{bookElements}</div>;
}
```

- **Books Array**: An array of book objects.
- **Mapping to Create Components**: Using `map` to iterate over the `books` array and create a `MobileList` component for each book.
- **Embedding Array in JSX**: `{bookElements}` is embedded within the JSX to render all `MobileList` components as children of the `<div>`.

Summary

- `{}` **in JSX**: Used to embed JavaScript expressions in JSX, allowing for dynamic and flexible UI rendering.
- **Combining JavaScript and HTML/XML-like Syntax**: JSX allows you to write JavaScript and HTML/XML-like syntax together, making it easier to build and visualize components.

So yes, the `{}` in JSX is a way to use JavaScript within your JSX, enabling dynamic and interactive user interfaces that integrate seamlessly with HTML/XML-like syntax.

Yes, outside of curly braces `{}`, JSX resembles HTML, not JavaScript. Inside curly braces `{}`, you embed JavaScript expressions for dynamic content and logic within

Khushal Savalakha

the JSX markup.

▼ Projects

> what is the use of projects in react ?

Sure, let's consider a simple React project structure for a hypothetical e-commerce website. We'll create components for a Header, Sidebar, and MainContent. Here's how the project structure might look:

```
my-react-app/
  ├── src/
  │   ├── components/
  │   │   ├── Header/
  │   │   │   ├── Header.js
  │   │   │   └── Header.css
  │   │   ├── Sidebar/
  │   │   │   ├── Sidebar.js
  │   │   │   └── Sidebar.css
  │   │   └── MainContent/
  │   │       ├── MainContent.js
  │   │       └── MainContent.css
  │   ├── App.js
  │   └── index.js
  ├── public/
  │   └── index.html
  ├── package.json
  └── README.md
```

Here's what each file or directory contains:

1. **src/**: This directory contains all the source code for our React application.
    - **components/**: This directory holds all our reusable React components.
        - **Header/**: This directory contains files related to the Header component.
            - **Header.js**: The JavaScript file defining the Header component.
            - **Header.css**: The CSS file for styling the Header component.

Khushal Savalakha

    - **Sidebar/**: Similar to Header, this directory contains files for the Sidebar component.
    - **MainContent/**: Similar to Header and Sidebar, this directory contains files for the MainContent component.
  - **App.js**: The main component of our React application where we'll compose the Header, Sidebar, and MainContent components.
  - **index.js**: The entry point of our React application where we'll render the main App component into the DOM.
2. **public/**: This directory contains static assets like the HTML file that serves as the entry point for our React application.
  - **index.html**: The HTML file where we'll mount our React application.
3. **package.json**: This file holds various metadata and dependencies for our project.
4. **README.md**: A markdown file containing information about our project.

Now, let's provide a simple code example for one of the components, say the Header component:

```
// Header.js
import React from 'react';
import './Header.css';

const Header = () => {
  return (
    <header className="header">
      <h1>My E-commerce Website</h1>
      {/* Add other header content here */}
    </header>
  );
};

export default Header;
```

```
/* Header.css */
.header {
  background-color: #333;
```


Khushal Savalakha

```
  color: #fff;
  padding: 20px;
}
```

In this example, the Header component is a simple functional component that renders a header with a title. We've also added basic styling for the header using CSS.

Similarly, you can create components for Sidebar and MainContent following the same structure and principles. Then, compose these components in the `App.js` file to build your complete application.

> It means Projects is one type of Structure ?

Yes, that's correct! In the context of software development, particularly in frameworks like React, "projects" refer to the overall structure and organization of your codebase. Think of a project as a container or framework within which you develop your application.

The project structure defines how your files and directories are organized, how components interact with each other, and how different parts of your application fit together. It encompasses aspects such as directory layout, naming conventions, component hierarchy, and file organization.

Having a well-defined project structure is crucial for several reasons:

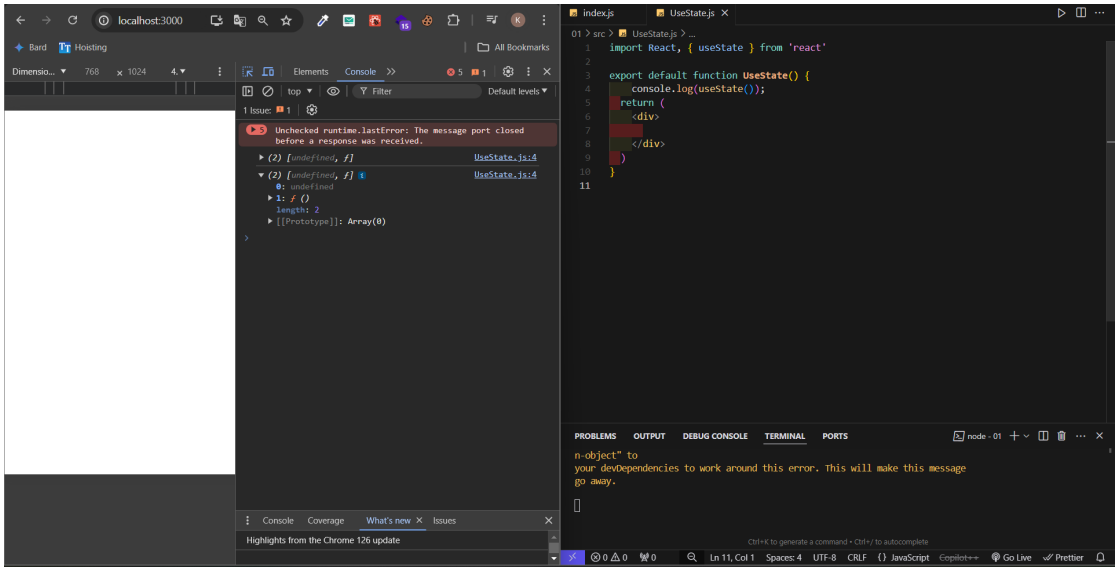
1. **Maintainability**: A clear and organized project structure makes it easier to understand, maintain, and extend your codebase over time.
2. **Collaboration**: It facilitates collaboration among team members by providing a standardized layout and conventions that everyone can follow.
3. **Scalability**: A good project structure allows your application to scale gracefully as it grows in size and complexity.
4. **Efficiency**: It promotes efficiency by streamlining development workflows, reducing cognitive overhead, and enabling better code reuse.

So, in essence, a project structure is a foundational aspect of software development, providing the framework within which you build and maintain your applications.

▼ Hooks

▼ useState

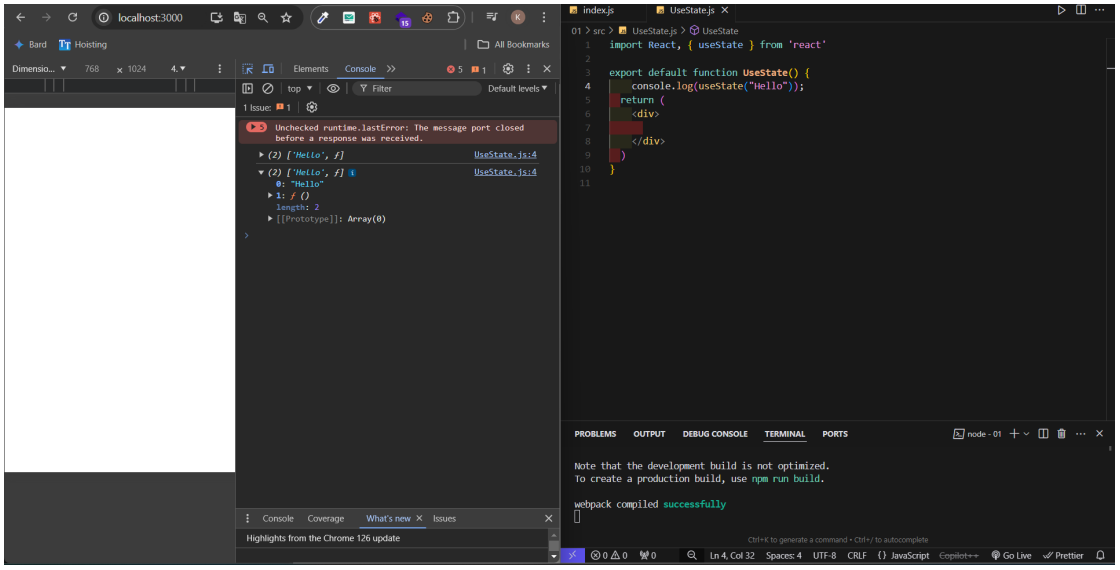
Khushal Savalakha



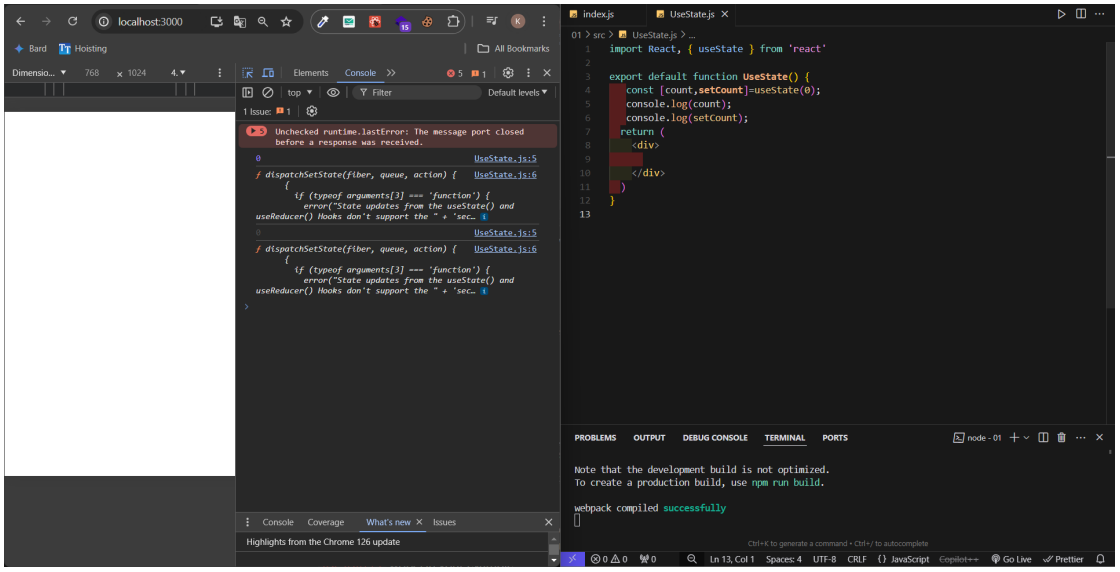
It means useState() simply returns an Array.Which has 2 elements.

1]Undefined

2]Function



Khushal Savalakha



In React, `useState()` is a Hook that allows you to add state to functional components. It enables you to declare state variables and update them within your components. Here's how `useState()` works in your example:

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

const App1 = () => {
    // useState() returns an array with two elements:
    // 1. The current state value (count)
    // 2. A function to update the state value (setCou
nt)
    const [count, setCount] = useState(0);

    console.log(count); // Logs the current value of c
ount
    console.log(setCount); // Logs the function used t
o update count

    const handleInc = () => {
        // When the + button is clicked, setCount is c
alled with the new value of count
        setCount(count + 1);
    };

    const handleDec = () => {
        // When the - button is clicked, setCount is c
```


Khushal Savalakha

```
alled with the new value of count
        setCount(count - 1);
    };

    return (
        <div>
            <h1>{count}</h1>
            {/* When the + button is clicked, handleIn
c function is called */}
            <button onClick={handleInc}>+</button>
            {/* When the - button is clicked, handleDe
c function is called */}
            <button onClick={handleDec}>-</button>
        </div>
    );
};

export default App1;
```

Explanation:

- `useState(0)` initializes the state variable `count` to 0. The `useState()` function returns an array with two elements:
  1. `count`: the current value of the state variable.
  2. `setCount`: a function that allows you to update the value of `count`.
- `handleInc` function increments the value of `count` by 1 when the + button is clicked. It calls `setCount` with the updated value of `count`.
- `handleDec` function decrements the value of `count` by 1 when the - button is clicked. It also calls `setCount` with the updated value of `count`.
- In the JSX, the value of `count` is displayed inside an `<h1>` tag. Two buttons are rendered: one for incrementing `count` and the other for decrementing it. The `onClick` attribute of each button is set to call the respective event handler functions (`handleInc` and `handleDec`) when clicked.

So, `useState()` allows you to manage state within functional components in React, making it possible to create dynamic and interactive UIs.

Khushal Savalakha

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

const App1=()=>{
    // console.log(useState("Hello"))
    const[count,setCount]=useState(0);
    console.log(count);
    console.log(setCount);

    const handleInc=()=>{
        setCount(count+1);
    }
    const handleDec=()=>{
        setCount(count-1);
    }
  return (
    <div>
      <h1>{count}</h1>
      <button onClick={handleInc}>+</button>
      <button onClick={handleDec}>-</button>
    </div>
  )
}

export default App1
```

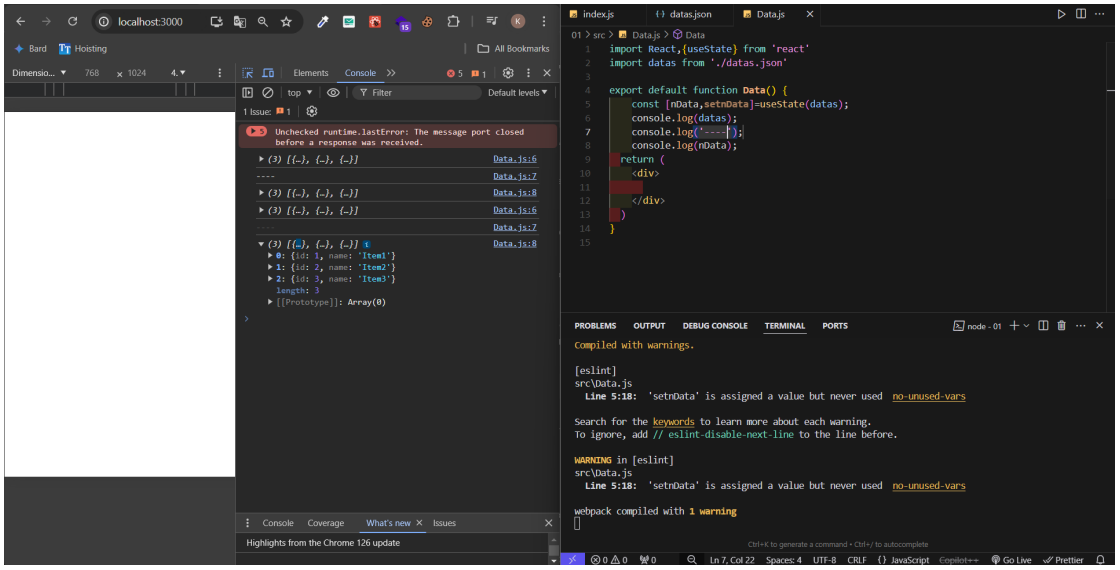
```
import React,{useState} from 'react'
import datas from './datas.json'

export default function Data() {
    const [nData,setnData]=useState(datas);
    // console.log(datas);
    // console.log('----');
    // console.log(nData);
  return (
    <div>
      <ul>
        {
          datas.map((items)=>
            <li key={items.id}>
              {items.name}
            </li>
          )
        }
      </ul>
    </div>
  )
}
```

```
WARNING in [eslint]
src\Data.js
  Line 5:12:  'nData' is assigned a value but never used    no-unused-vars
  Line 5:18:  'setnData' is assigned a value but never used  no-unused-vars

webpack compiled with 1 warning
```

```
import React, { useState } from "react";
import datas from "./datas.json";
export default function Data() {
//   const [nData, setnData] = useState(datas);
  // console.log(datas)
  return (
    <div>
    <ul>
      {datas.map((items) => ( // Use parentheses instead of curly braces
        <li key={items.id}>
          {console.log(items.id)}
        </li>
      ))}
    </ul>
  </div>
);
}
```

```
Compiled successfully!

You can now view 01 in the browser.

  Local:            http://localhost:3000
  On Your Network:  http://192.168.56.1:3000

Note that the development build is not optimized.
To create a production build, use npm run build.

webpack compiled successfully
```


Khushal Savalakha




```
import React, { useState } from "react";
import datas from "./datas.json";
export default function Data() {
//   const [nData, setnData] = useState(datas);
  // console.log(datas)
  return (
    <div>
    <ul>
      {datas.map((items) => ( // Use parentheses instead
        <li key={items.id}>
          {console.log(items.id)}
        </li>
      ))}
    </ul>
  </div>
);
}
```

```
import React, { useState } from "react";
import datas from "./datas.json";
export default function Data() {
  const [nData, setnData] = useState(datas);
  // console.log(datas)
```

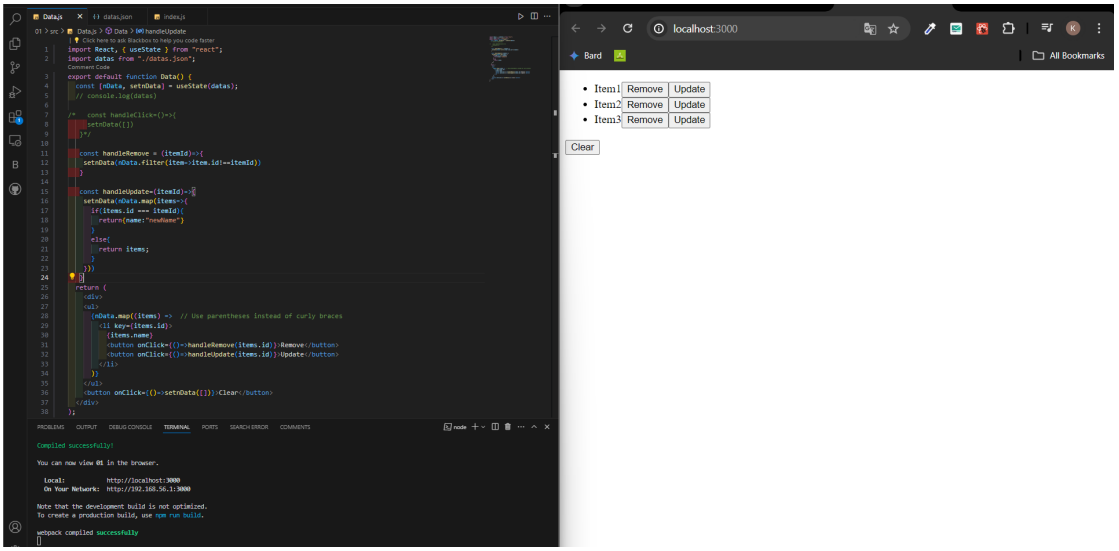

Khushal Savalakha

```
/*   const handleClick=()=>{
     setnData([])
   }*/

   const handleRemove = (itemId)=>{
    setnData(nData.filter(item=>item.id!==itemId))
   }

   const handleUpdate=(itemId)=>{
    setnData(nData.map(items=>{
      if(items.id === itemId){
        return{name:"newName"}
      }
      else{
        return items;
      }
    }))
   }
  return (
    <div>
    <ul>
      {nData.map((items) =>  // Use parentheses instead
        <li key={items.id}>
          {items.name}
          <button onClick={()=>handleRemove(items.id)}>R
          <button onClick={()=>handleUpdate(items.id)}>U
        </li>
      )}
    </ul>
    <button onClick={()=>setnData([])}>Clear</button>
  </div>
);
}
```

```
//datas.json
[
    {"id":1,"name":"Item1"},
    {"id":2,"name":"Item2"},
```


Khushal Savalakhia

```
    {"id":3,"name":"Item3"}
]
```



Bard

- newName Remove Update
- Item2 Remove Update
- Item3 Remove Update

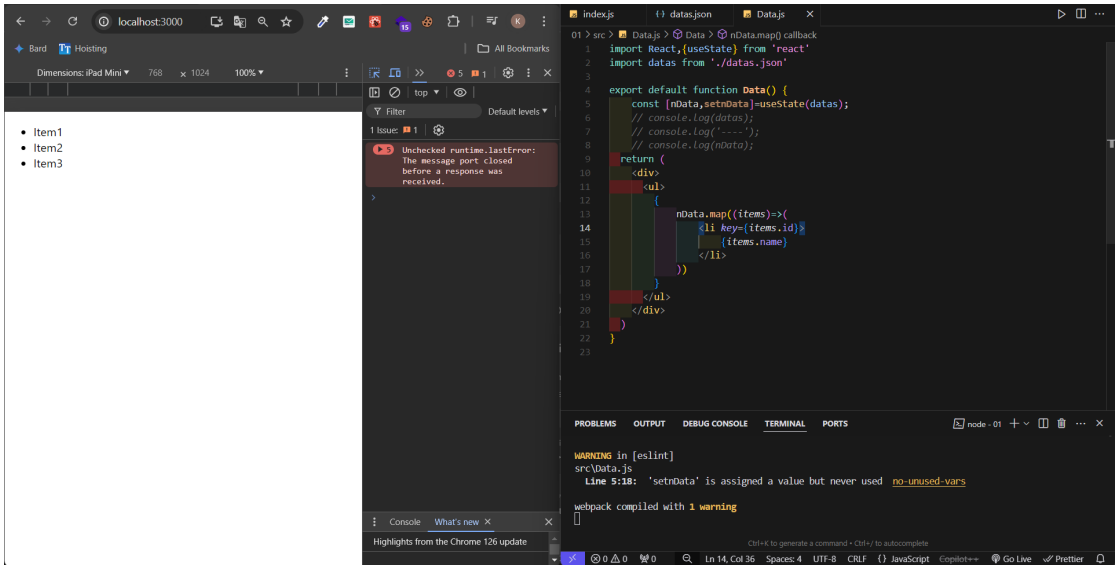
Clear

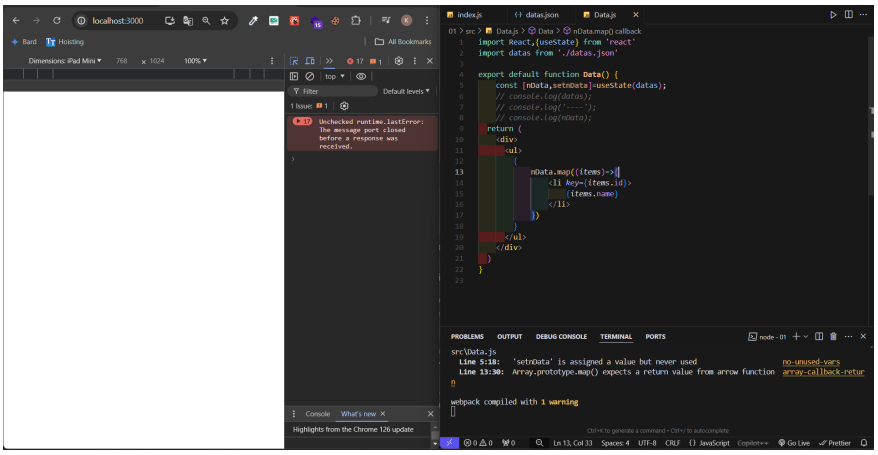
Bard

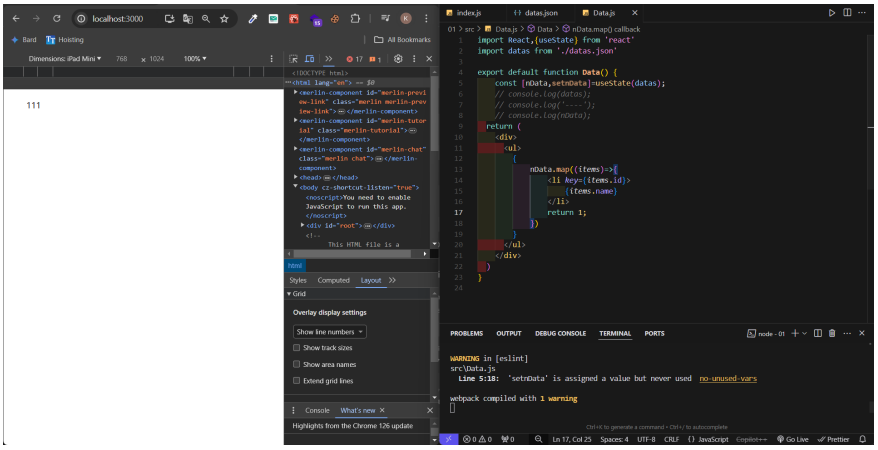
- newName Remove Update
- Item3 Remove Update

Clear

Khushal Savalakha







In JavaScript, arrow functions have different syntax forms that affect whether and how they return values. Understanding these forms is key to using arrow functions effectively, especially in React when working with JSX. Let's explore these concepts:

Arrow Function Syntax Forms

Khushal Savalakha

1. **No Parameters**:

```
() => expression
```

2. **Single Parameter**:

```
param => expression
```

3. **Multiple Parameters**:

```
(param1, param2, ...) => expression
```

Implicit Return vs. Explicit Return

- **Implicit Return**: If the function body is a single expression without curly braces `{}`, the value of that expression is automatically returned.

```
// Implicit return with parentheses
const add = (a, b) => a + b;
```

- **Explicit Return**: If the function body is enclosed in curly braces `{}`, you need to use the `return` statement to return a value.

```
// Explicit return with curly braces
const add = (a, b) => {
  return a + b;
};
```

Applying to JSX in React

In React, when using the `map` method to render lists, you need to ensure the arrow function used returns the JSX elements correctly.

Common Pitfall: Missing Return

Consider your original code:

```
nData.map((items) => {
  <li key={items.id}>
```


Khushal Savalakha

```
    {items.name}
  </li>
});
```

Here, the arrow function uses curly braces `{}`, so you must explicitly return the JSX:

```
nData.map((items) => {
  return (
    <li key={items.id}>
      {items.name}
    </li>
  );
});
```

Alternatively, you can use an implicit return with parentheses:

```
nData.map((items) => (
  <li key={items.id}>
    {items.name}
  </li>
));
```

Examples to Illustrate Concepts

1. **Implicit Return Example**:

```
// Using implicit return
const numbers = [1, 2, 3];
const doubled = numbers.map(num => num * 2);
console.log(doubled); // [2, 4, 6]
```

Sure! Let's break down the line of code:

Explanation of `nData.map((items) => ( ... ))`

This line of code is using the `map` method to iterate over the `nData` array, and for each item in the array, it is returning a JSX `<li>` element. Here's a step-by-step breakdown:

1. `nData.map`:

Khushal Savalakha

- `nData` is an array (initialized using `useState` with data from `datas.json`).
- The `map` method creates a new array by calling a provided function on every element in the calling array.

2. **Arrow Function**:
   - `(items) => ( ... )` is an arrow function.
   - This arrow function is the callback provided to the `map` method.
   - It takes one argument, `items`, which represents each element in the `nData` array.
3. **Implicit Return**:
   - The arrow function uses parentheses `()` instead of curly braces `{}`, which means it will implicitly return the value inside the parentheses.
   - This is useful for returning single expressions without needing an explicit `return` statement.
4. **JSX Element**:
   - Inside the parentheses, a JSX `<li>` element is defined.
   - `<li key={items.id}> {items.name} </li>`:
     - `<li>`: The list item element in HTML.
     - `key={items.id}`: The `key` attribute is used by React to identify each element uniquely. It helps in efficiently updating the list when elements change.
     - `{items.name}`: This is a JSX expression that outputs the `name` property of the current `items` object.

Full Code Context

Here's the complete code for better understanding:

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

export default function Data() {
  const [nData, setnData] = useState(datas);

  return (
```


Khushal Savalakhia

```
    <div>
      <ul>
        {
          nData.map((items) => (
            <li key={items.id}>
              {items.name}
            </li>
          ))
        }
      </ul>
    </div>
  );
}
```

Detailed Steps

1. **Initialization**:
   - `const [nData, setnData] = useState(datas);`
   - Initializes `nData` with data from `datas.json`.
2. **Rendering**:
   - `<ul>` is an unordered list in HTML.
   - `{ nData.map((items) => ( ... )) }`:
     - This part iterates over `nData`.
     - For each `items` in `nData`, it creates a `<li>` element.
     - The `key` attribute is set to `items.id` to uniquely identify each list item.
     - The content of the `<li>` element is set to `items.name`.
3. **Output**:
   - The `map` method returns an array of `<li>` elements.
   - These elements are rendered inside the `<ul>`.

Summary

- `map` **Method**: Iterates over the `nData` array.
- **Arrow Function**: Takes each `items` and returns a JSX `<li>` element.

Khushal Savalakha

- **Implicit Return**: The JSX inside parentheses is returned for each `items`.
- **JSX**: Constructs the HTML-like structure to be rendered by React.

This pattern is common in React for rendering lists of items dynamically. The `key` attribute is crucial for React's reconciliation process, helping it to track and update elements efficiently.

1. **Explicit Return Example**:

```
// Using explicit return
const numbers = [1, 2, 3];
const doubled = numbers.map(num => {
  return num * 2;
});
console.log(doubled); // [2, 4, 6]
```

2. **React Example with JSX**:

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

export default function Data() {
  const [nData, setnData] = useState(datas);

  return (
    <div>
      <ul>
        {
          nData.map((items) => (
            <li key={items.id}>
              {items.name}
            </li>
          ))
        }
      </ul>
    </div>
  );
}
```


Khushal Savalakha

Summary

- **Arrow Functions**: Can have implicit or explicit returns depending on whether curly braces `{}` are used.
- **Implicit Return**: No curly braces `{}`, the expression after `=>` is returned.
- **Explicit Return**: Use curly braces `{}` and a `return` statement inside them.
- **React JSX**: Ensure your arrow functions in `map` return JSX elements correctly, either implicitly or explicitly, to render lists.

By understanding these principles, you can avoid common pitfalls and write more concise and correct code, especially when working with React components and JSX.

```
import React,{useState} from 'react'
import datas from './datas.json'

export default function Data() {
    const [nData,setnData]=useState(datas);
    // console.log(datas);
    // console.log('----');
    // console.log(nData);
  return (
    <div>
      <ul>
        {
            nData.map((items)=>
                <li key={items.id}>
                    {items.name}
                </li>
            )
        }
      </ul>
    </div>
  )
}

here we don't have to write return statement because it
```


Khushal Savalakha

Yes, you are correct. In your updated code, the arrow function implicitly returns the JSX element because there are no curly braces `{}`. When you omit curly braces, the arrow function returns the result of the expression immediately following the `=>` symbol.

Correct Code Explanation

Your code is using an implicit return, which is concise and works perfectly for this use case:

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

export default function Data() {
  const [nData, setnData] = useState(datas);

  return (
    <div>
      <ul>
        {
          nData.map((items) =>
            <li key={items.id}>
              {items.name}
            </li>
          )
        }
      </ul>
    </div>
  );
}
```

Breakdown

1. **useState Initialization**:

   ```
   const [nData, setnData] = useState(datas);
   ```

   - Initializes the state `nData` with the contents of `datas.json`.

2. **JSX Return**:

Khushal Savalakha

```
return (
  <div>
    <ul>
      {
        nData.map((items) =>
          <li key={items.id}>
            {items.name}
          </li>
        )
      }
    </ul>
  </div>
);
```

- Inside the `map` function, each `items` element is implicitly returned as a `<li>` element without needing an explicit `return` statement.

Implicit Return

- **Without Curly Braces**:

```
nData.map((items) =>
  <li key={items.id}>
    {items.name}
  </li>
)
```

  - This implicitly returns the JSX inside the arrow function.

- **With Curly Braces** (Explicit Return Needed):

```
nData.map((items) => {
  return (
    <li key={items.id}>
      {items.name}
    </li>
  );
})
```


Khushal Savalakha

- When curly braces are used, you must include an explicit `return` statement.

Summary

Your current code correctly uses an implicit return within the arrow function to render the list items. This is a concise and clean way to write the `map` function for rendering JSX elements in React.

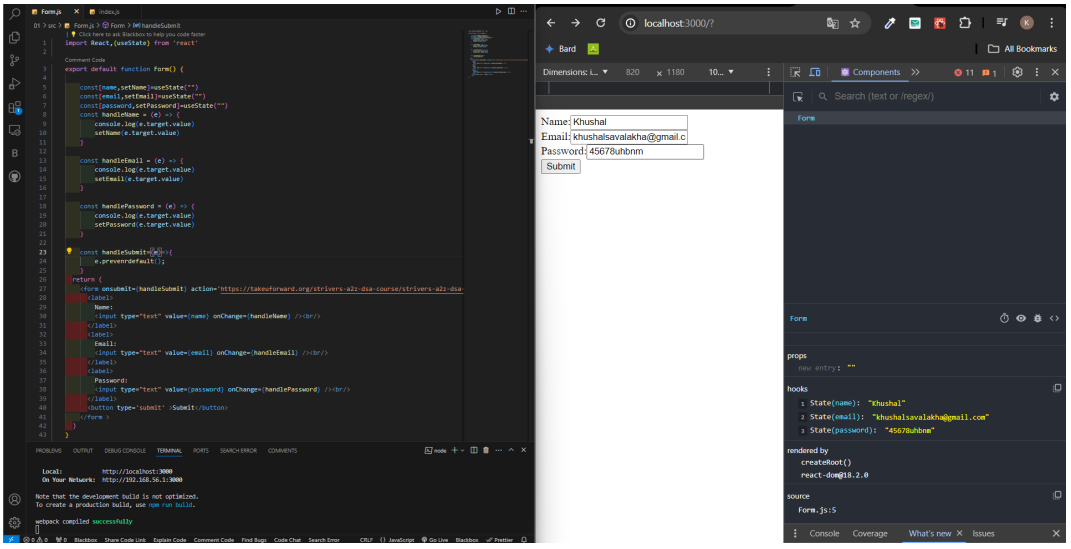
> Form

▼ Multiple States:

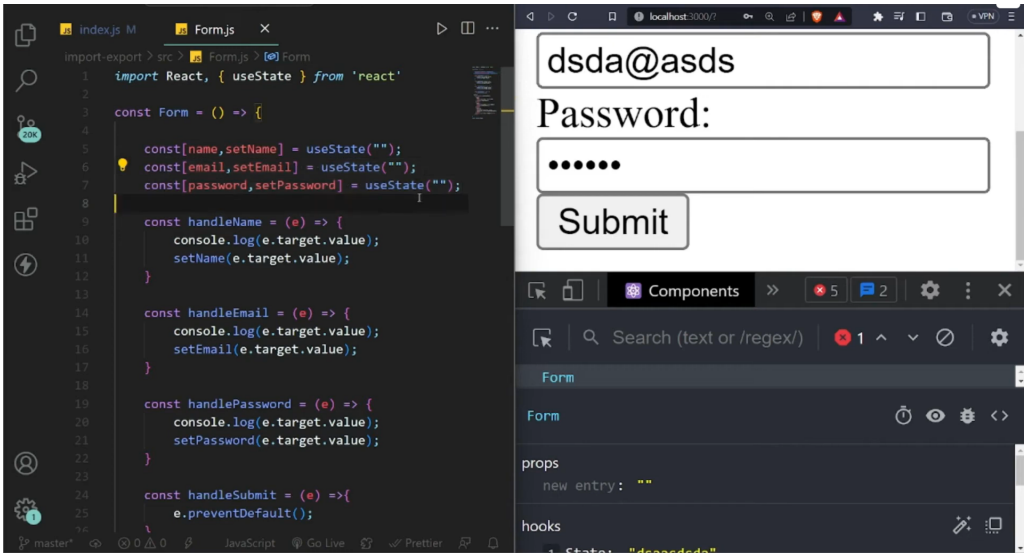
```
import React,{useState} from 'react'

export default function Form() {

    const[name,setName]=useState("")
    const[email,setEmail]=useState("")
    const[password,setPassword]=useState("")
    const handleName = (e) => {
        console.log(e.target.value)
        setName(e.target.value)
    }

    const handleEmail = (e) => {
        console.log(e.target.value)
        setEmail(e.target.value)
    }

    const handlePassword = (e) => {
        console.log(e.target.value)
        setPassword(e.target.value)
    }

    const handleSubmit=(e)=>{
        e.prevenrdefault();
```


Khushal Savalakha

```
      }
    return (
      <form onsubmit={handleSubmit} action='https://take
        <label>
          Name:
          <input type="text" value={name} onChange={hand
        </label>
        <label>
          Email:
          <input type="text" value={email} onChange={har
        </label>
        <label>
          Password:
          <input type="text" value={password} onChange={
        </label>
        <button type='submit' >Submit</button>
      </form >
    )
  }
```



Khushal Savalakha

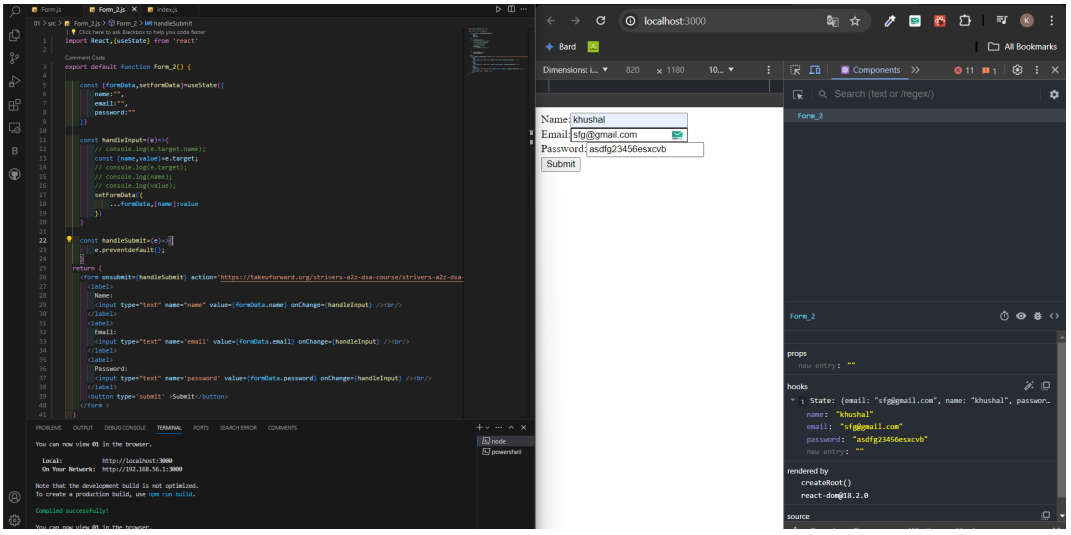


▼ Multiple States to One States

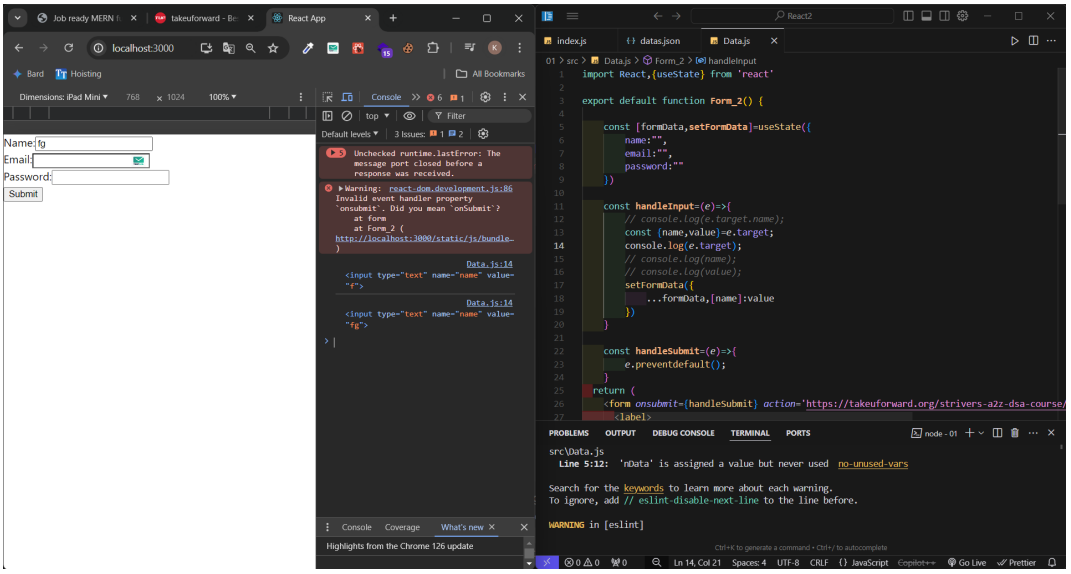
```
import React,{useState} from 'react'

export default function Form_2() {

    const [formData,setFormData]=useState({
        name:"",
        email:"",
        password:""
    })

    const handleInput=(e)=>{
        // console.log(e.target.name);
        const {name,value}=e.target;
        // console.log(e.target);
        // console.log(name);
        // console.log(value);
        setFormData({
            ...formData,[name]:value
        })
    }

    const handleSubmit=(e)=>{
        e.preventdefault();
    }
  return (
```

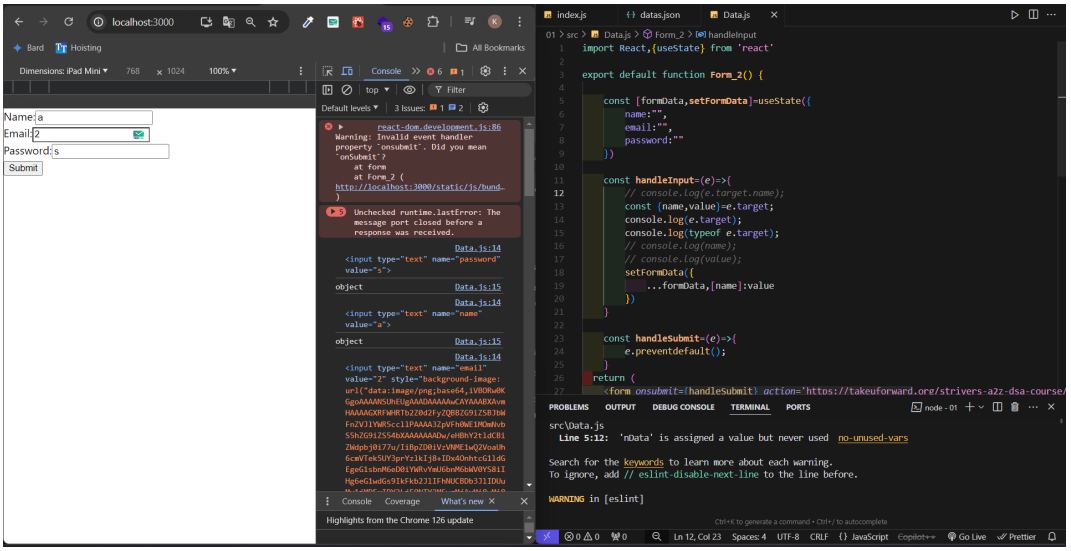

Khushal Savalakha

```
      <form onsubmit={handleSubmit} action='https://take
        <label>
          Name:
          <input type="text" name="name" value={formData
        </label>
        <label>
          Email:
          <input type="text" name='email' value={formDat
        </label>
        <label>
          Password:
          <input type="text" name='password' value={form
        </label>
        <button type='submit' >Submit</button>
      </form >
    )
  }
```



Khushal Savalakha





In JavaScript, when you destructure an object, the variable names inside the curly braces `{}` are matched to the property names of the object being destructured based on their keys.

In the line:

```
const { name, value } = e.target;
```

JavaScript looks at the structure of `e.target`, which is an object representing the input element that triggered the event. It then matches the variable names `name` and `value` inside the curly braces `{}` with the property names of the object `e.target`.

For example, if `e.target` has properties like `{ name: "inputName", value: "inputValue" }`, then destructuring `{ name, value }` would essentially assign

Khushal Savalakha

`"inputName"` to the `name` variable and `"inputValue"` to the `value` variable.

So, in the context of an input event (`e`), the `name` property typically refers to the `name` attribute of the input element, and the `value` property refers to the current value entered into the input field.

JavaScript matches these property names (`name` and `value`) from `e.target` to the corresponding variable names (`name` and `value`) inside the curly braces during destructuring, allowing you to conveniently access these properties using the variable names in subsequent code.

This way, you can easily access properties of an object without explicitly referencing the object's property names.

i don't understand setFormData({
  ...formData,
  [name]: value
});

how this is working ?

Let's break down how `setFormData` works in the provided code:

```
setFormData({
    ...formData,
    [name]: value
});
```

This line of code is updating the state `formData` with a new object. Here's how it works:

1. `{ ...formData }`: This part spreads the current state `formData` into a new object. The spread operator (`...`) creates a new object containing all the key-value pairs of the existing `formData` state.
2. `[name]: value`: This part dynamically updates the property specified by the `name` variable (which represents the `name` attribute of the input field that triggered the change event) with the new `value`.

- `[name]` is enclosed in square brackets `[]`, which indicates dynamic property assignment in JavaScript. It allows you to use a variable (`name` in this case) as the key for the object property.

Khushal Savalakha

- `value` represents the new value entered into the input field associated with the `name` attribute.

So, when you put `{ ...formData }` before `[name]: value`, you're essentially creating a new object that:

- Contains all the existing key-value pairs from `formData` (`...formData`).
- Updates the property specified by `[name]` with the new `value`.

For example, let's say `formData` initially looks like this:

```
{
    name: "",        // initial value for name
    email: "",       // initial value for email
    password: ""     // initial value for password
}
```

If the user enters "John" into the `name` input field, the `handleInput` function will be called, and `setFormData` will be executed with the following parameters:

```
setFormData({
    ...formData,  // Spread the existing state
    [name]: value // Update the 'name' property wit
h the new value
});
```

This will result in `formData` being updated to:

```
{
    name: "John",    // Updated value for name
    email: "",       // Unchanged value for email
    password: ""     // Unchanged value for password
}
```

So, `setFormData` effectively updates the state by merging the existing state (`formData`) with the updated property (`[name]: value`). This approach ensures that only the specific field being changed is updated in the `formData` state, while the rest of the state remains unchanged.

> i don't understand why we have to write ...formData ?

Khushal Savalakha

The spread operator `...` in JavaScript is used to create a shallow copy of an object or array. In the context of React state management, using the spread operator allows you to create a new object with the same properties as the existing object, making it easier to update specific properties without mutating the original state directly.

Here's why we use `...formData` in the `setFormData` function:

1. **Preserving Existing State**: When you're updating the state in React, it's important to preserve the existing state while making changes. Directly modifying the state can lead to unexpected behavior, as React relies on immutable data to efficiently detect changes and trigger re-renders.
2. **Immutability**: React state should be treated as immutable. Instead of modifying the existing state directly, you create a new state object with the desired changes. This ensures that React can accurately detect and manage state updates.

By using `...formData`, you're essentially creating a new object that contains all the key-value pairs of the existing `formData` state. This new object represents the updated state, with the addition of any new properties or updates specified after the spread operator.

For example, let's consider the `formData` state object:

```
{
    name: "",
    email: "",
    password: ""
}
```

If you only wanted to update the `name` property, you could do so by creating a new object with the same properties as `formData`:

```
{
    ...formData,    // Spread the existing state
    name: "John"    // Update the 'name' property
}
```

This results in a new object:

Khushal Savalakha

```
{
    name: "John",
    email: "",
    password: ""
}
```

By spreading `formData` before updating the `name` property, you ensure that the new object maintains all the properties of the original state (`email` and `password`) while only modifying the specified property (`name`). This approach adheres to the principles of immutability and ensures predictable state management in React components.

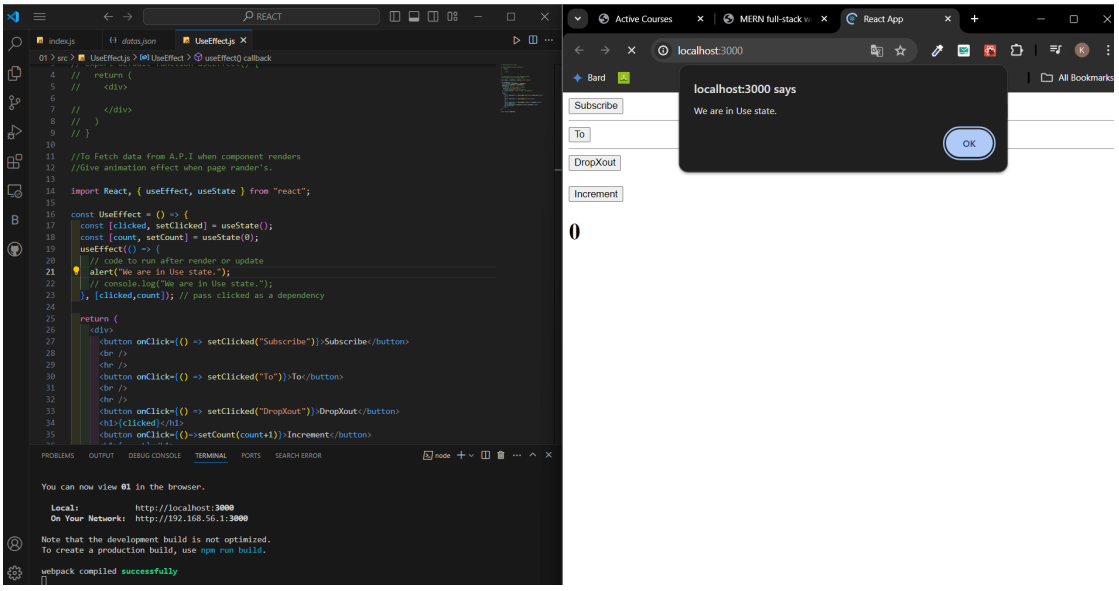
▼ useEffect

rfc,refc

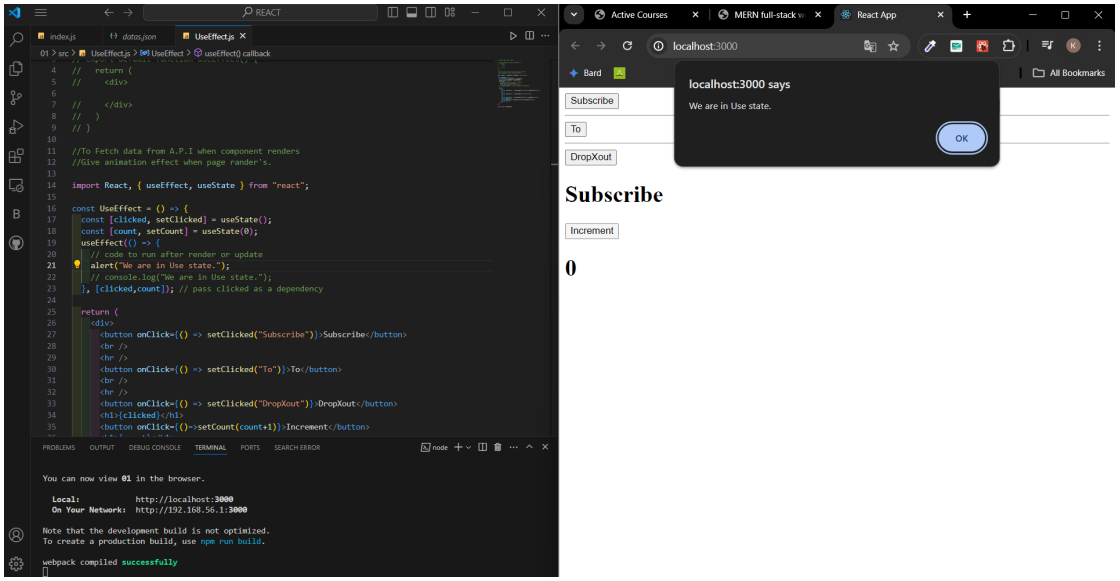
```
// import React from 'react'

// export default function useEffect() {
//   return (
//     <div>

//     </div>
//   )
// }

//To Fetch data from A.P.I when component renders
//Give animation effect when page rander's.


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

const UseEffect = () => {
  const [clicked, setClicked] = useState();
  const [count, setCount] = useState(0);
  useEffect(() => {
    // code to run after render or update
    // alert("We are in Use state.");
    console.log("We are in Use state.");
```

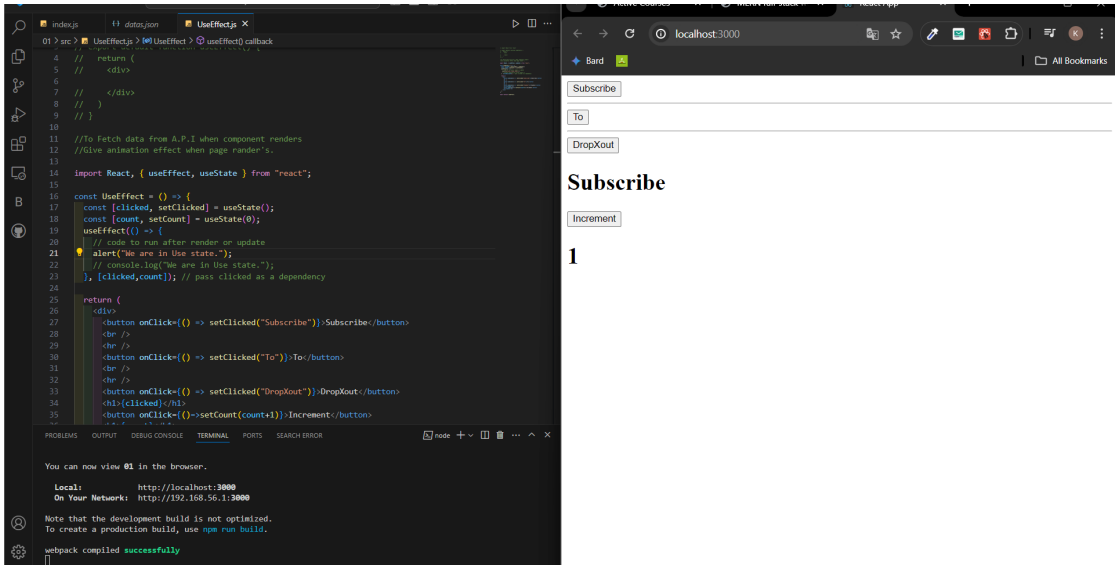

Khushal Savalakha

```
  }, [clicked,count]);
  // pass clicked and count as a dependency so it will s

  return (
    <div>
      <button onClick={() => setClicked("Subscribe")}>Su
      <br />
      <hr />
      <button onClick={() => setClicked("To")}>To</butto
      <br />
      <hr />
      <button onClick={() => setClicked("DropXout")}>Dro
      <h1>{clicked}</h1>
      <button onClick={()=>setCount(count+1)}>Increment<
      <h1>{count}</h1>
    </div>
  );
};

export default UseEffect;
```



Khushal Savalakhia





This code is a React functional component named `UseEffect`. Let's break down what each part does:

1. **Imports**:
   - `React`: This imports the necessary modules from React, including the `useEffect` and `useState` hooks.
2. **Function Component**:
   - `UseEffect`: This is the functional component being defined.
3. **State Variables**:
   - `clicked`: This state variable is created using the `useState` hook. It holds the value of the button that was clicked.

Khushal Savalakha

- `count`: Another state variable created using the `useState` hook. It keeps track of the number of times the "Increment" button is clicked.

4. **Effect Hook**:
   - `useEffect`: This hook is called within the component. It runs after every render or update. In this case, it displays an alert message saying "We are in Use state." whenever either the `clicked` or `count` state variables change. The dependency array `[clicked, count]` specifies that the effect should only re-run if either `clicked` or `count` changes.
5. **JSX**:
   - Inside the `return` statement, JSX elements are rendered.
   - Three buttons are displayed: "Subscribe", "To", and "DropXout". Each button has an `onClick` event handler that sets the `clicked` state variable to a specific string value when clicked.
   - Two `<h1>` elements display the values of the `clicked` and `count` state variables.
   - There's an "Increment" button that, when clicked, increments the `count` state variable by 1.

Overall, this component demonstrates the usage of the `useState` and `useEffect` hooks in React to manage state and perform side effects respectively. It also illustrates how to specify dependencies for the `useEffect` hook to control when it should re-run.

> 💡 useEffect is running before return code of component and it also provide it's return statement which is recognized as cleaner of previous code and it also do action on provided array of useState am I true ? If anything else about useEffect then explain.

You are partially correct in your understanding of `useEffect`. Here are some clarifications and additional details:

1. **Execution Timing**: `useEffect` runs after the render is committed to the screen, not before. It's a post-render effect.
2. **Cleanup Function**: The function you return from `useEffect` is indeed used for cleanup. This cleanup runs before the effect is re-executed (if dependencies change) and when the component unmounts.

Khushal Savalakha

3. **Dependencies**: The array you provide as the second argument to `useEffect` dictates when the effect runs. If any value in this array changes between renders, the effect re-runs.

Here's an example to illustrate:

```
useEffect(() => {
  // Effect code here (e.g., fetching data)
  return () => {
    // Cleanup code here (e.g., unsubscribing, clearin
g intervals)
  };
}, [dependency1, dependency2]); // Dependencies array
```

1. **Purpose**: `useEffect` is used for managing side effects such as data fetching, subscriptions, and manually changing the DOM.
2. **No Dependencies**: If the dependencies array is empty (`[]`), the effect runs only once after the initial render.
3. **No Array**: If you omit the dependencies array, the effect runs after every render.

Feel free to ask if you have more questions about `useEffect` or any other React concepts!

> 💡 It means it's working is after component render first function execute that outside return statement code executed in which useEffect is included which work is cleaning before execution of another task, fetching data , work of provided dependencies which executed when provided useState executed and it's not necessary to write return statement all time in useEffect am I true?

Yes, you're correct with some additional clarifications:

1. **Effect Execution**: After the component renders, the `useEffect` function is executed. It performs tasks such as fetching data or subscribing to services.
2. **Cleanup**: If the effect returns a function, this function is used for cleanup. This cleanup runs before the effect is re-executed on subsequent renders (when dependencies change) or when the component unmounts.

Khushal Savalakha

3. **Dependencies**: The effect runs only when the dependencies change.
4. **Optional Cleanup**: It's not necessary to always include a return statement for cleanup in `useEffect`. If no cleanup is needed, you can omit it.

So, the `useEffect` hook allows you to manage side effects in your components with control over execution and cleanup based on dependencies.

▼ clean up

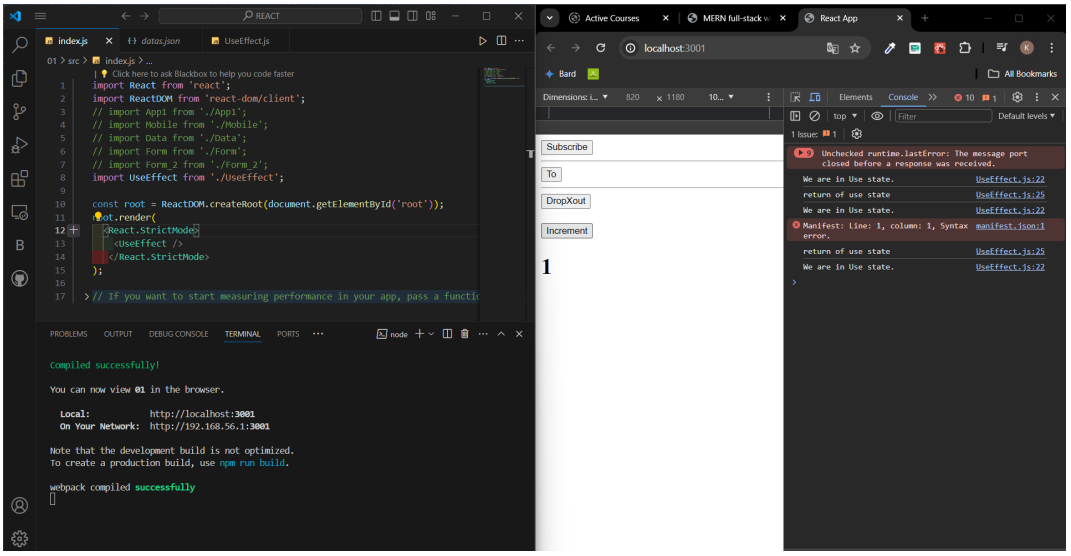
```
import React from 'react';
import ReactDOM from 'react-dom/client';
// import App1 from './App1';
// import Mobile from './Mobile';
// import Data from './Data';
// import Form from './Form';
// import Form_2 from './Form_2';
import UseEffect from './UseEffect';

const root = ReactDOM.createRoot(document.getElementBy
root.render(
  // <React.StrictMode>
    <UseEffect />
  // </React.StrictMode>
);

// If you want to start measuring performance in your
// to log results (for example: reportWebVitals(consol
// or send to an analytics endpoint. Learn more: https
// reportWebVitals();
```

```
// import React from 'react'

// export default function useEffect() {
//   return (
//     <div>

//     </div>
//   )
```


Khushal Savalakha

```
// }

//To Fetch data from A.P.I when component renders
//Give animation effect when page rander's.

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

const UseEffect = () => {
  const [clicked, setClicked] = useState();
  const [count, setCount] = useState(0);
  useEffect(() => {
    // code to run after render or update
    // alert("We are in Use state.");
    console.log("We are in Use state.");
    return(
      ()=>{
        console.log("return of use state");
      }
    )
  }, [clicked,count]); // pass clicked as a dependency

  return (
    <div>
      <button onClick={() => setClicked("Subscribe")}>
      <br />
      <hr />
      <button onClick={() => setClicked("To")}>To</but
      <br />
      <hr />
      <button onClick={() => setClicked("DropXout")}>D
      <h1>{clicked}</h1>
      <button onClick={()=>setCount(count+1)}>Incremer
      <h1>{count}</h1>
    </div>
  );
};

export default UseEffect;
```


Khushal Savalakha



In your `UseEffect` component, you use the `useEffect` hook to execute code after the component renders or updates. The `useEffect` hook can also return a cleanup function, which runs before the component is removed from the DOM or before the effect runs again on subsequent updates. Let's break down how this works in your code:

Code Explanation

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

const UseEffect = () => {
  const [clicked, setClicked] = useState();
  const [count, setCount] = useState(0);

  useEffect(() => {
    // Code to run after render or update
    console.log("We are in Use state.");

    // Return a cleanup function
    return () => {
      console.log("return of use state");
    };
  }, [clicked, count]); // Pass clicked and count a
s dependencies

  return (
```


Khushal Savalakha

```
    <div>
      <button onClick={() => setClicked("Subscrib
e")}>Subscribe</button>
      <br />
      <hr />
      <button onClick={() => setClicked("To")}>To</
button>
      <br />
      <hr />
      <button onClick={() => setClicked("DropXou
t")}>DropXout</button>
      <h1>{clicked}</h1>
      <button onClick={() => setCount(count + 1)}>I
ncrement</button>
      <h1>{count}</h1>
    </div>
  );
};

export default UseEffect;
```

useEffect Hook

- **Execution of Effect**:
  - When the component first renders, the `useEffect` hook runs the code inside it. In this case, it logs "We are in Use state." to the console.
- **Dependencies Array**:
  - The dependencies array `[clicked, count]` tells React to re-run the effect whenever `clicked` or `count` changes.
- **Cleanup Function**:
  - The function returned from the `useEffect` hook is the cleanup function.
  - This cleanup function runs before the effect runs again on updates, and before the component unmounts.
  - In your code, it logs "return of use state" to the console.

Example Sequence

Khushal Savalakha

1. **Initial Render**:
   - The component renders for the first time.
   - `useEffect` runs, logging "We are in Use state.".
   - No cleanup function runs because it's the initial render.
2. **Update State**:
   - When you click any of the buttons, the state updates (`clicked` or `count`).
   - Before running the effect again, React runs the cleanup function, logging "return of use state".
   - Then the effect runs again, logging "We are in Use state.".
3. **Unmounting Component**:
   - If the component is removed from the DOM, the cleanup function runs one last time.

Key Points

- **Effect**: The code inside the `useEffect` callback runs after the render.
- **Cleanup**: The function returned by `useEffect` runs before the next effect execution and before the component unmounts.
- **Dependencies**: The array `[clicked, count]` specifies when to re-run the effect and cleanup.

Visual Representation

```
useEffect(() => {
  // Runs after initial render and after every upda
te when dependencies change
  console.log("We are in Use state.");

  return () => {
    // Runs before the next effect execution or whe
n component unmounts
    console.log("return of use state");
  };
}, [clicked, count]); // Dependencies
```


Khushal Savalakha

Summary

- **Initial Render**: Logs "We are in Use state.".
- **Update (`clicked` or `count`)**:
  - Logs "return of use state".
  - Logs "We are in Use state.".
- **Unmount**: Logs "return of use state".

This ensures side effects (like fetching data or subscribing to events) are managed correctly, with proper cleanup to prevent memory leaks or unwanted behavior.

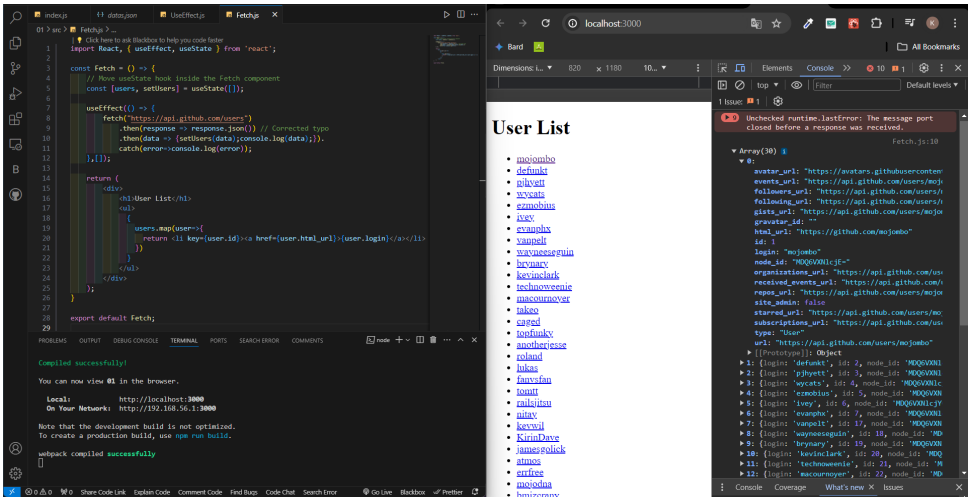
▼ Fetch using useEffect

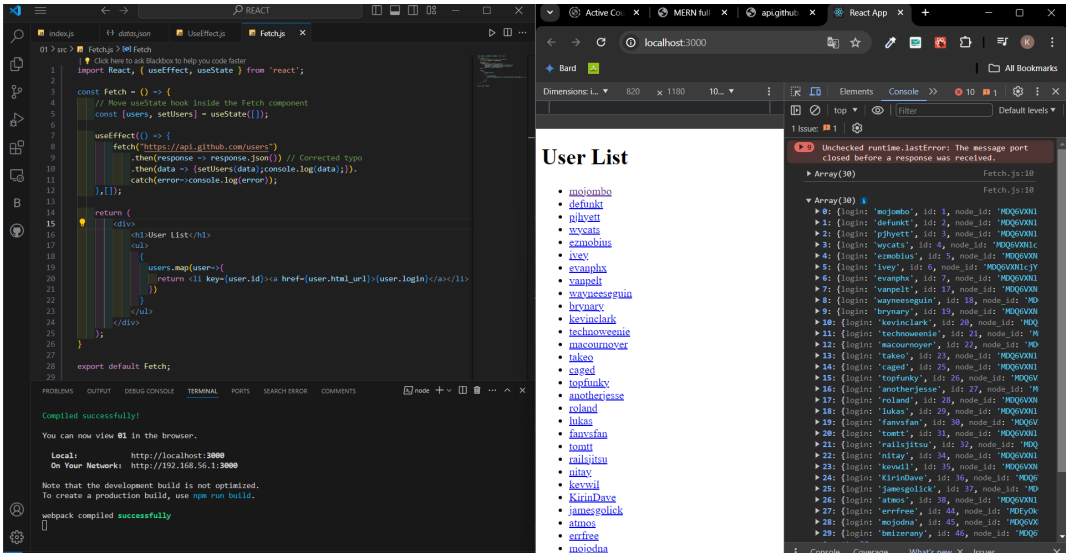
https://api.github.com/users

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

const Fetch = () => {
    // Move useState hook inside the Fetch component
    const [users, setUsers] = useState([]);

    useEffect(() => {
        fetch("https://api.github.com/users")
            .then(response => response.json())
            .then(data => {setUsers(data);console.log(
            catch(error=>console.log(error));
    },[]);

    return (
        <div>
            <h1>User List</h1>
            <ul>
              {
                users.map(user=>{
                  return <li key={user.id}><a href={us
                })
              }
            </ul>
        </div>
```


Khushal Savalakha

```
    );
}

export default Fetch;
```





Sure, let's break down how this code works:

1. **Import Statements**:
   - `import React, { useEffect, useState } from 'react';`: Imports the necessary functions and hooks from the React library.
2. **Functional Component `Fetch`**:
   - `const Fetch = () => { ... }`: Defines a functional component named `Fetch`.

Khushal Savalakha

3. **State Management with `useState`**:
   - `const [users, setUsers] = useState([]);`: Initializes a state variable `users` as an empty array and a function `setUsers` to update the state.
4. **Fetching Data with `useEffect`**:
   - `useEffect(() => { ... }, []);`: Utilizes the `useEffect` hook to perform side effects in function components. The empty dependency array `[]` ensures that the effect runs only once when the component mounts.
   - `fetch("<https://api.github.com/users>")`: Initiates a GET request to the GitHub API endpoint that returns a list of users.
   - `.then(response => response.json())`: Converts the response from JSON format to a JavaScript object.
   - `.then(data => {setUsers(data);console.log(data);})`: Updates the `users` state with the fetched data and logs the data to the console.
   - `.catch(error => console.log(error));`: Handles any errors that occur during the fetch operation and logs them to the console.
5. **Rendering User List**:
   - The JSX within the `return` statement renders a `<div>` containing an `<h1>` with the text "User List" and a `<ul>` element.
   - Within the `<ul>` element, it maps over the `users` array and renders a list item `<li>` for each user. Each list item contains an anchor `<a>` element with the user's GitHub username (`user.login`) as the text and a link to their GitHub profile (`user.html_url`) as the `href` attribute.
6. **Key Prop for List Items**:
   - Each `<li>` element generated by the `.map()` function is assigned a unique `key` prop using the `user.id`. This helps React efficiently identify which items have changed, added, or removed during re-rendering.
7. **Exporting the Component**:
   - `export default Fetch;`: Exports the `Fetch` component as the default export, allowing it to be imported and used in other parts of the application.

Overall, this component

Khushal Savalakha

> <ul>
> {
> users.map(user=>{
> return <li key={
> user.id}><a href={user.html_url}>{user.login}</a></li>
> })
> }
> </ul>

> i don't understand here why we have to return it and what is the reason behind it to assign key?

In React, when you use the `map()` function to iterate over an array and generate JSX elements dynamically, each JSX element generated must have a unique `key` prop assigned to it. This `key` prop helps React identify which items have changed, been added, or been removed during re-rendering.

Let's break down the code snippet you provided:

```
<ul>
  {
    users.map(user => {
      return <li key={user.id}><a href={user.html_u
rl}>{user.login}</a></li>
    })
  }
</ul>
```

- `users.map(user => { ... })`: This iterates over each item in the `users` array and returns a new array of JSX elements.
- `<li key={user.id}><a href={user.html_url}>{user.login}</a></li>`: This JSX represents a list item (`

<li>`) containing an anchor (`

<a>`) element. For each user in the` users `array, it creates a list item with a unique` key `assigned to it (` key={user.id} `). The` user.id` is typically a unique identifier provided by the API, ensuring that each list item has a distinct key.

Khushal Savalakha

The reason behind assigning a `key` to each list item is to help React efficiently update the DOM when the array of items changes. When React renders a list, it needs a way to differentiate between items and their respective DOM elements. By providing a unique `key` prop, React can optimize updates by only re-rendering or reordering the necessary list items instead of re-rendering the entire list. This improves performance and reduces unnecessary DOM manipulation.

In summary, the `return` statement within the `map()` function is used to generate JSX elements dynamically for each item in the `users` array, and assigning a unique `key` prop to each list item helps React efficiently manage and update the list.

> it means when we fetch data from json in array which are object .here we have array in which we add the data but while printing that jason data from array it still have problem of missing data am i true ?

Yes, you're correct. When fetching data from an external API, especially when it returns an array of objects like JSON data, it's essential to handle it properly in React.

Here's why:

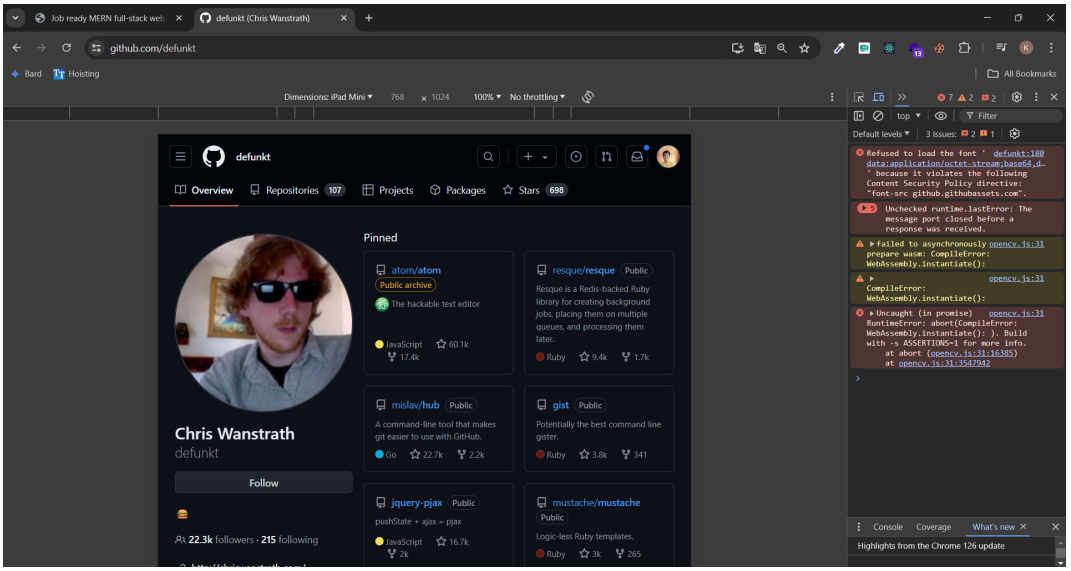
1. **Asynchronous Nature**: Fetching data from an API is an asynchronous operation. It means that when you make the fetch request, the JavaScript code continues to execute without waiting for the response. Once the response arrives, it triggers the `.then()` block. During this time, React might have already rendered the component without the data.
2. **Initial State**: React components typically render with some initial state. If you're fetching data to update that state, you initially render the component with empty or placeholder data. When the actual data arrives, you update the state, triggering a re-render to display the fetched data.
3. **Handling Missing Data**: It's crucial to handle scenarios where the data may not be available immediately. This includes displaying loading indicators or placeholders until the data arrives, as well as handling errors if the fetch operation fails.

In the provided code, the `users` state is initially an empty array. When the fetch operation completes successfully, the fetched data is stored in the `users` state

Khushal Savalakha

using `setUsers(data)`. React automatically re-renders the component with the updated state, displaying the fetched user data.

However, if there's a delay in fetching the data or if the fetch operation fails, the component might render without the expected data. It's essential to handle such scenarios gracefully, ensuring a good user experience.



▼ multiple return

https://www.notion.so/Java-Script-1-146bab5249d147f3ae045ef4a99c6eb1?pvs=4#f0669a0d5cf640e38f27d2358b8d2ffd

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

const MultipleReturn = () => {
    const [users, setUsers] = useState([]);
    const [isLoading, setIsLoading] = useState(true);
    const [error, setError] = useState(null);

    useEffect(() => {
        async function fetchUsers() {
            try {
                const response = await fetch('https://
                const data = await response.json();
                setUsers(data);
                setIsLoading(false);
            } catch (error) {
                setError(error);
```


Khushal Savalakha

```
                setIsLoading(false);
            }
        }
        fetchUsers();
    }, []);

    if (isLoading) {
        return <p>Loading...</p>;
    }
    if (error) {
        return <p>Error: {error.message}</p>;
    }
    return (
        <div>
            <h1>Users List</h1>
            <ul>
                {users.map(user => (
                    <li key={user.id}>{user.login}</li
                ))}
            </ul>
        </div>
    );
}

export default MultipleReturn;
```

💡 it means here await is waiting until data is fetched from link and then other process happen am i true ?

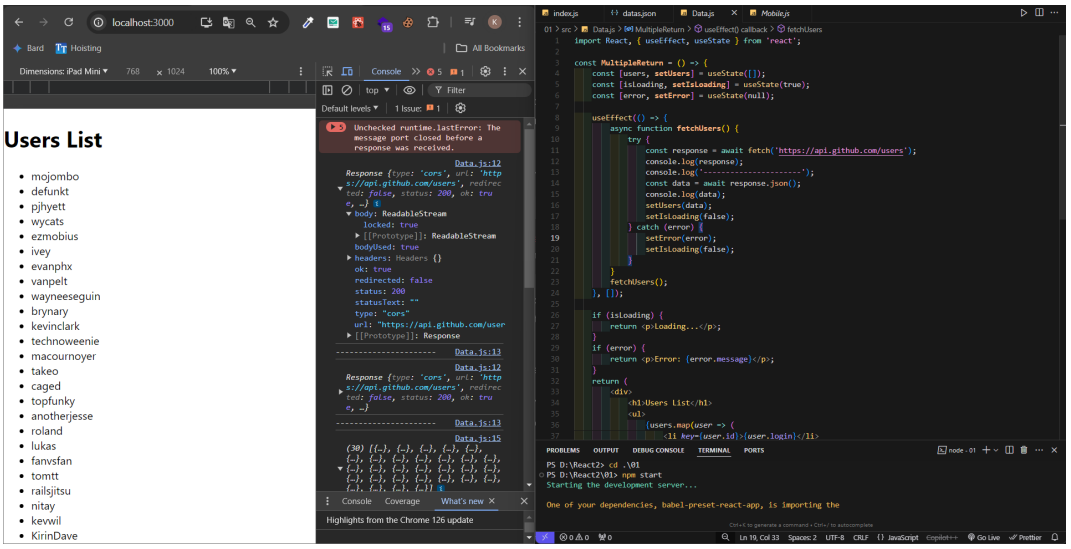
Yes, you're correct. In the provided `useEffect` code, `await` is used to pause the execution of the function until the data is fetched from the provided URL. Here's a more detailed breakdown:

1. **await fetch('https://api.github.com/users')**: This line pauses the execution of `fetchUsers` until the `fetch` call completes and returns a response object.
2. **await response.json()**: This line waits for the response to be parsed as JSON. Only after the JSON parsing is complete, it assigns the result to `data`.

Khushal Savalakha

3. After both `await` calls are complete, the subsequent lines (`setUsers(data)` and `setIsLoading(false)`) are executed.

This ensures that each step waits for the previous asynchronous operation to complete before proceeding.



> const response = await fetch('https://api.github.com/users');
> console.log(response);here which data in response ?

When you use `await fetch('<https://api.github.com/users>');`, the `response` object contains the HTTP response from the fetch request. This includes several pieces of information:

1. **status**: The HTTP status code (e.g., 200 for success, 404 for not found).
2. **statusText**: The status message corresponding to the status code.
3. **ok**: A boolean indicating whether the response was successful (status code in the range 200-299).
4. **headers**: The response headers.
5. **url**: The URL of the response.
6. **body**: The response body, which is a ReadableStream.

To see the actual user data, you need to parse the JSON body of the response using `await response.json()`. Here is an example of logging the response object and the user data:

Khushal Savalakha

```
const response = await fetch('<https://api.github.com/users>');
console.log(response); // Logs the response object with metadata
const data = await response.json();
console.log(data); // Logs the user data
```

In summary, `response` contains metadata about the HTTP response, and you need to call `.json()` on it to get the actual data.

💡 then what is response.json() do ?

The `response.json()` method is used to parse the body of the HTTP response as JSON. When you call `response.json()`, it returns a promise that resolves with the result of parsing the response body text as JSON.

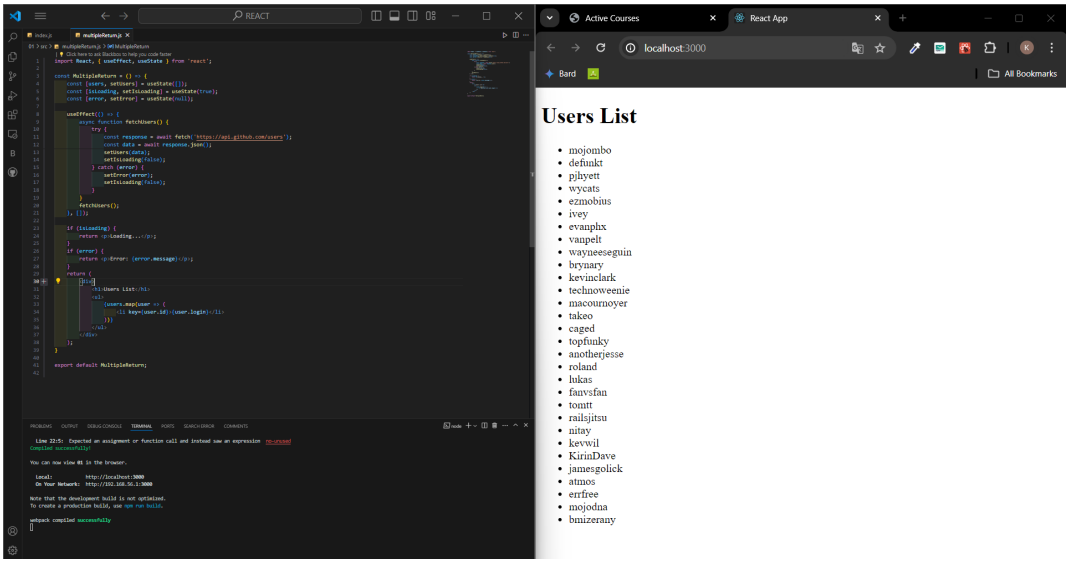
Here's how it works in the context of your code:

```
const response = await fetch('<https://api.github.com/users>');
const data = await response.json();
```

1. `fetch('<https://api.github.com/users>')` makes a network request to the GitHub API and returns a response object.
2. `response.json()` reads the response body and parses it as JSON.
3. `data` contains the parsed JSON data, which in this case is an array of user objects.

In essence, `response.json()` converts the response body into a JavaScript object or array, making it easier to work with in your code.

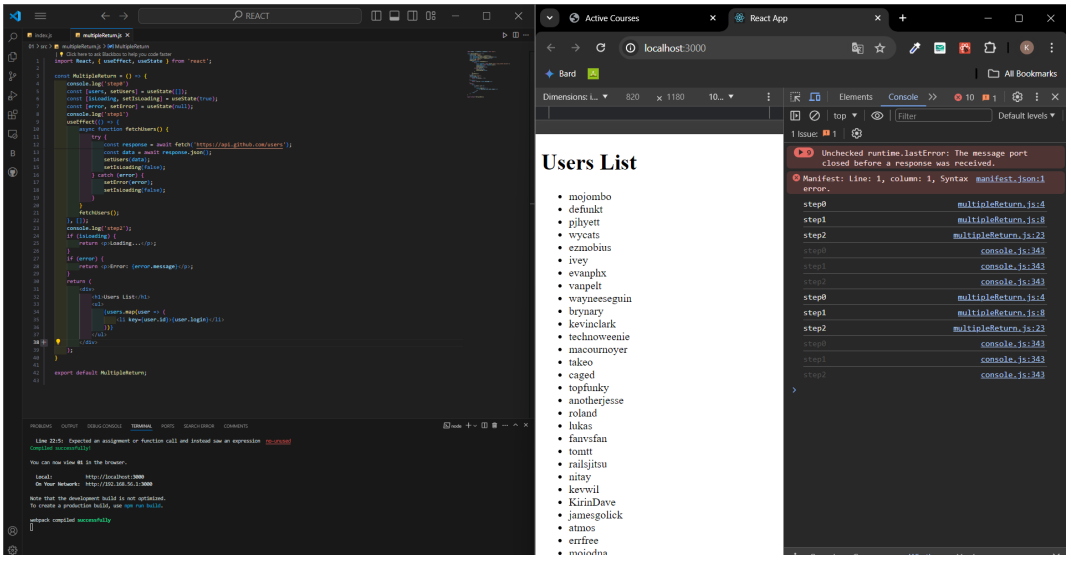
Khushal Savalakha




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

const MultipleReturn = () => {
    console.log('step0')
    const [users, setUsers] = useState([]);
    const [isLoading, setIsLoading] = useState(true);
    const [error, setError] = useState(null);
    console.log('step1')
    useEffect(() => {
        async function fetchUsers() {
            try {
                const response = await fetch('https://
                const data = await response.json();
                setUsers(data);
                setIsLoading(false);
            } catch (error) {
                setError(error);
                setIsLoading(false);
            }
        }
        fetchUsers();
    }, []);
    console.log('step2');
    if (isLoading) {
        return <p>Loading...</p>;
```


Khushal Savalakha

```
    }
    if (error) {
        return <p>Error: {error.message}</p>;
    }
    return (
        <div>
            <h1>Users List</h1>
            <ul>
                {users.map(user => (
                    <li key={user.id}>{user.login}</li
                ))}
            </ul>
        </div>
    );
}

export default MultipleReturn;
```



▼ Explaination 1

Sure, let's break down the code:

1. **Importing React and useState, useEffect Hooks**:

```
import React, { useEffect, useState } from 're
act';
```


Khushal Savalakha

Here, we're importing React and two of its built-in Hooks: `useState` and `useEffect`. These Hooks are used for managing state and performing side effects in functional components.

2. **Functional Component Declaration**:

```
const MultipleReturn = () => {
```

This code defines a functional component named `MultipleReturn`. It's a stateless component, meaning it doesn't have its own internal state.

3. **State Initialization**:

```
const [users, setUsers] = useState([]);
const [isLoading, setIsLoading] = useState(tru
e);
const [error, setError] = useState(null);
```

Here, we're initializing three state variables using the `useState` Hook:

- `users`: An array that will hold the fetched GitHub users' data.
- `isLoading`: A boolean flag to indicate whether data is being fetched (initially set to `true`).
- `error`: A variable to store any potential errors that occur during the data fetching process.

4. **Fetching Data with useEffect Hook**:

```
useEffect(() => {
    async function fetchUsers() {
        try {
            const response = await fetch('<htt
ps://api.github.com/users>');
            const data = await response.json
();
            setUsers(data);
            setIsLoading(false);
        } catch (error) {
            setError(error);
            setIsLoading(false);
```


Khushal Savalakha

```
        }
    }
    fetchUsers();
}, []);
```

The `useEffect` Hook is used to perform side effects in functional components. Here, we're using it to fetch data from the GitHub API when the component mounts (empty dependency array `[]`). Inside the `useEffect` callback, we define an `async` function `fetchUsers()` to fetch data from the API. If the fetch is successful, we update the `users` state with the fetched data and set `isLoading` to `false`. If an error occurs during the fetch, we set the `error` state and also set `isLoading` to `false`.

5. **Conditional Rendering**:

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

Here, we're using conditional rendering to display different content based on the current state of the component. If `isLoading` is `true`, we display a loading message. If an `error` occurred during data fetching, we display an error message with the error details.

6. **Rendering User List**:

```
return (
    <div>
        <h1>Users List</h1>
        <ul>
            {users.map(user => (
                <li key={user.id}>{user.login}
</li>
            ))}
        </ul>
```


Khushal Savalakha

```
    </div>
);
```

If neither `isLoading` nor `error` is `true`, we render a list of GitHub users. We use the `map` function to iterate over the `users` array and render a list item (`

li`) for each user, displaying their` login`username. Each list item has a unique` key`attribute set to the` user.id`to help React efficiently manage the list items. Finally, the list is wrapped in a` div` with a heading (` h1`) indicating it's a Users List.

> in this i don't understand on which step condition's are executes ?

In the provided code, the conditions for checking whether to display the loading message or the error message are evaluated during the component's rendering phase. Let's break down when each condition is executed:

1. **Initial Render**:
   When the component is initially rendered, the `useEffect` Hook runs because it's executed after the initial render. However, since the `isLoading` state is initially set to `true`, the first condition `if (isLoading)` will evaluate to `true`. As a result, the loading message `<p>Loading...</p>` will be rendered to the screen.

2. **Data Fetching**:
   While the loading message is being displayed, the `fetchUsers()` function inside the `useEffect` Hook is asynchronously fetching data from the GitHub API. Once the data is fetched, the `setUsers(data)` and `setIsLoading(false)` functions are called to update the `users` state with the fetched data and set `isLoading` to `false`.

3. **Re-Render**:
   After the state updates caused by `setUsers(data)` and `setIsLoading(false)`, a re-render of the component is triggered. During this re-render, the conditions for displaying the loading message or error message are evaluated again.

4. **Condition Evaluation**:

Khushal Savalakha

- If the data fetching was successful and `isLoading` is set to `false`, the condition `if (isLoading)` will evaluate to `false`, and the loading message will no longer be displayed.
- If an error occurred during data fetching, the `catch` block will set the `error` state, and `isLoading` will be set to `false`. Consequently, the condition `if (error)` will evaluate to `true`, and the error message `<p>Error: {error.message}</p>` will be rendered to the screen.
- If neither `isLoading` nor `error` is `true`, the conditions for displaying the loading or error message will be `false`, and the user list will be rendered.

In summary, the conditions for displaying the loading message or error message are evaluated during each render cycle based on the current state of the component (`isLoading` and `error` states). Depending on the state changes caused by data fetching or error handling, different content will be rendered to the screen accordingly.

💡 Depth Explaination

💡 here in this code when components renders first it will go to assign states then useEffect and then it will goes to async function fetch users while fetching data it will show loading.. and then it will go to return statement after completion of javascript data and print data am i true ?

Your understanding is mostly correct, but let's break it down step-by-step to clarify the flow of execution in the code:

1. **Component Initialization**:
   - The component starts by initializing the state variables: `users` (an empty array), `isLoading` (set to `true`), and `error` (set to `null`).
2. **Initial Render**:
   - The component renders for the first time. At this point, the `useEffect` hook is scheduled to run after the render is committed to the DOM.

Khushal Savalakha

- Because `isLoading` is `true`, the component displays the "Loading..." message.

3. **useEffect Execution**:
   - After the initial render, the `useEffect` hook executes. This is where the `fetchUsers` async function is defined and immediately called.
   - The `fetchUsers` function attempts to fetch data from the GitHub API.
4. **Fetching Data**:
   - While fetching the data, the "Loading..." message is still displayed because `isLoading` is still `true`.
   - `await fetch('<https://api.github.com/users>')` sends the request and waits for the response.
   - `await response.json()` parses the JSON data from the response.
5. **Updating State**:
   - If the fetch is successful, `setUsers(data)` updates the `users` state with the fetched data.
   - `setIsLoading(false)` sets `isLoading` to `false`, indicating that loading is complete.
   - If there is an error during the fetch, `setError(error)` updates the `error` state, and `setIsLoading(false)` sets `isLoading` to `false`.
6. **Re-render**:
   - Once the state updates, the component re-renders.
   - If `isLoading` is now `false`, the "Loading..." message is not displayed.
   - If `error` is not `null`, the error message is displayed.
   - If there is no error and `users` contains data, the component displays the list of users.

Here is the updated code for reference:

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

const MultipleReturn = () => {
    const [users, setUsers] = useState([]);
    const [isLoading, setIsLoading] = useState(tru
```


Khushal Savalakha

```
e);
    const [error, setError] = useState(null);

    useEffect(() => {
        async function fetchUsers() {
            try {
                const response = await fetch('<http
s://api.github.com/users>');
                console.log(response);
                console.log('----------------------
');
                const data = await response.json();
                console.log(data);
                setUsers(data);
                setIsLoading(false);
            } catch (error) {
                setError(error);
                setIsLoading(false);
            }
        }
        fetchUsers();
    }, []);

    if (isLoading) {
        return <p>Loading...</p>;
    }
    if (error) {
        return <p>Error: {error.message}</p>;
    }
    return (
        <div>
            <h1>Users List</h1>
            <ul>
                {users.map(user => (
                    <li key={user.id}>{user.login}
</li>
                ))}
            </ul>
```


Khushal Savalakha

```
        </div>
    );
}

export default MultipleReturn;
```

So, in summary:

- The component first initializes states and displays "Loading...".
- The `useEffect` hook fetches data asynchronously.
- State updates trigger re-renders to display either the fetched data or an error message.

▼ Quick Revise

💡 what is side effect in useEffect ?

In React, the `useEffect` hook is used to handle side effects in functional components. A side effect in this context refers to any operation that affects something outside the scope of the function being executed. These operations can include:

- **Data Fetching**: Making network requests to fetch data from an API.
- **Subscriptions**: Setting up subscriptions (like WebSocket or event listeners) and cleaning them up when the component unmounts.
- **Manual DOM Manipulation**: Directly manipulating the DOM outside of React's normal rendering cycle.
- **Timers**: Setting and clearing intervals or timeouts.

Understanding Side Effects in `useEffect`

Basic Syntax

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

function MyComponent() {
  useEffect(() => {
```


Khushal Savalakha

```
    // Your side effect code here
  }, []); // Dependency array
}
```

Common Side Effects

1. **Data Fetching**

```
useEffect(() => {
  async function fetchData() {
    const response = await fetch('<https://api.e
xample.com/data>');
    const data = await response.json();
    console.log(data);
  }
  fetchData();
}, []); // Empty dependency array means this eff
ect runs once after the initial render
```

2. **Subscriptions and Cleanups**

```
useEffect(() => {
  const handleResize = () => {
    console.log(window.innerWidth);
  };

  window.addEventListener('resize', handleResiz
e);

  return () => {
    window.removeEventListener('resize', handleR
esize);
  };
}, []); // Cleanup function is called when the c
omponent unmounts
```

3. **Timers**

Khushal Savalakha

```
useEffect(() => {
  const intervalId = setInterval(() => {
    console.log('This will run every second!');
  }, 1000);

  return () => {
    clearInterval(intervalId);
  };
}, []); // Cleanup the interval on component unm
ount
```

Dependency Array

The dependency array is the second argument to `useEffect`. It specifies when the effect should be re-run. The effect will run after the initial render and then whenever any of the dependencies change.

- **Empty Dependency Array (`[]`)**: The effect runs only once after the initial render.
- **No Dependency Array**: The effect runs after every render.
- **Specific Dependencies**: The effect runs when any of the specified dependencies change.

```
useEffect(() => {
  // Effect code here
}, [dependency1, dependency2]); // Effect runs when
dependency1 or dependency2 change
```

Cleanup Function

If the effect returns a function, React will run it when it is time to clean up, for example, before unmounting the component or before running the effect next time (if dependencies have changed).

```
useEffect(() => {
  const handleResize = () => {
    console.log(window.innerWidth);
  };
```


Khushal Savalakha

```
  window.addEventListener('resize', handleResize);

  return () => {
    window.removeEventListener('resize', handleResi
ze);
  };
}, []); // Cleanup function is called when the comp
onent unmounts
```

Summary

Side effects are operations that interact with the outside world or cause observable changes outside the scope of the component's rendering process. The `useEffect` hook provides a way to handle these side effects in functional components, ensuring that they are executed at the appropriate times and that any necessary cleanup is performed to avoid memory leaks and other issues. By understanding and correctly using the dependency array and cleanup functions, you can effectively manage side effects in your React applications.

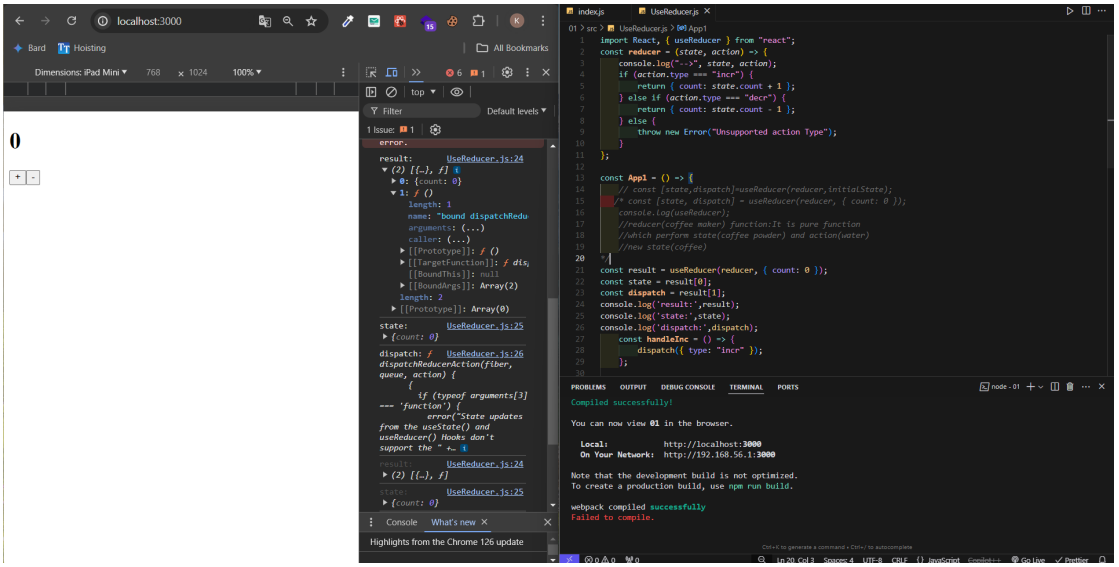
▼ useReducer

```
import React, { useReducer } from "react";
const reducer = (state, action) => {
    console.log("-->", state, action);
    if (action.type === "incr") {
        return { count: state.count + 1 };
    } else if (action.type === "decr") {
        return { count: state.count - 1 };
    } else {
        throw new Error("Unsupported action Type");
    }
};

const App1 = () => {
    // const [state,dispatch]=useReducer(reducer,initial
   /* const [state, dispatch] = useReducer(reducer, { co
    console.log(useReducer);
    //reducer(coffee maker) function:It is pure function
```


Khushal Savalakha

```
    //which perform state(coffee powder) and action(wate
    //new state(coffee)
*/
const result = useReducer(reducer, { count: 0 });
const state = result[0];
const dispatch = result[1];
console.log('result:',result);
console.log('state:',state);
console.log('dispatch:',dispatch);
    const handleInc = () => {
        dispatch({ type: "incr" });
    };

    const handleDec = () => {
        dispatch({ type: "decr" });
    };

    return (
        <div>
            <h1>{state.count}</h1>
            {/* When the + button is clicked, handleInc
            <button onClick={handleInc}>+</button>
            {/* When the - button is clicked, handleDec
            <button onClick={handleDec}>-</button>
        </div>
    );
};

export default App1;
```


Khushal Savalakha



Certainly! Let's break down the syntax and behavior of the `useReducer` hook in more detail to understand how it works.

Understanding `useReducer`

The `useReducer` hook in React is used to manage state in a more predictable way, especially for complex state logic. The function signature is:

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

- `reducer`: A function that determines how the state should change based on an action.
- `initialState`: The initial state value.

How `useReducer` Works Internally

1. **Initialization**:
   - When `useReducer` is called, it sets up the state with the `initialState` and creates a `dispatch` function that can be used to send actions to the `reducer`.
2. **Return Values**:
   - `useReducer` returns an array with two elements:
     - The current state.
     - The `dispatch` function.

Khushal Savalakha

Breaking Down the Code

Let's break down the exact code snippet:

```
const [state, dispatch] = useReducer(reducer, { count:
0 });
```

1. Calling `useReducer`

When you call `useReducer(reducer, { count: 0 })`, React performs the following steps:

- **Initial State Setup**: React initializes the state to `{ count: 0 }`.
- **Dispatch Function Creation**: React creates a `dispatch` function that, when called, will take an action and use it to update the state by calling the `reducer`.

2. Destructuring the Return Value

The return value of `useReducer` is an array containing:

- The current state.
- The `dispatch` function.

```
const result = useReducer(reducer, { count: 0 });
// result is [state, dispatch]
```

This array is then destructured into two separate variables:

- `state`: Holds the current state.
- `dispatch`: The function used to send actions to the reducer.

So, the line:

```
const [state, dispatch] = useReducer(reducer, { count:
0 });
```

is effectively doing this:

```
const result = useReducer(reducer, { count: 0 });
const state = result[0];
```


Khushal Savalakha

```
const dispatch = result[1];
```

Understanding the Flow

1. **Initial State Assignment**:
   - `useReducer` initializes the state to `{ count: 0 }`.
2. **Dispatch Function**:
   - `useReducer` creates a `dispatch` function.
3. **Destructuring**:
   - The array `[state, dispatch]` is destructured into two variables:
     - `state` gets the initial state value `{ count: 0 }`.
     - `dispatch` gets the function to send actions to the reducer.

Example Flow

1. **Initial Call**:

```
const [state, dispatch] = useReducer(reducer, { cou
nt: 0 });
```

   - `state` is `{ count: 0 }`.
   - `dispatch` is a function to send actions to the reducer.
2. **Using `dispatch`**:

```
const handleInc = () => {
    dispatch({ type: "incr" });
};
```

   - `handleInc` calls `dispatch` with an action `{ type: "incr" }`.
   - `dispatch` sends the action to the reducer function.
3. **Reducer Function**:

```
const reducer = (state, action) => {
    if (action.type === "incr") {
        return { count: state.count + 1 };
```


Khushal Savalakha

```
    } else if (action.type === "decr") {
        return { count: state.count - 1 };
    } else {
        throw new Error("Unsupported action type");
    }
};
```

- The reducer receives the current state and the action.
- It returns a new state based on the action type.

4. **State Update and Re-render**:
   - React updates the state with the new value returned by the reducer.
   - React re-renders the component with the updated state.

Visualization

Let's visualize the process step by step:

1. **Initial Setup**:

```
const [state, dispatch] = useReducer(reducer, { cou
nt: 0 });
```

   - State: `{ count: 0 }`
   - Dispatch: `dispatch` function

2. **Button Click (`+`)**:

```
const handleInc = () => {
    dispatch({ type: "incr" });
};
```

3. **Dispatch Action**:

```
dispatch({ type: "incr" });
```

4. **Reducer Execution**:

Khushal Savalakha

```
const reducer = (state, action) => {
    if (action.type === "incr") {
        return { count: state.count + 1 };
    }
    // Other cases
};
```

5. **State Update**:
   - New state: `{ count: 1 }`
6. **Re-render**:

```
<h1>{state.count}</h1> // Displays 1
```

By understanding these steps, you can see how `useReducer` initializes the state, how the `dispatch` function works, and how the reducer updates the state based on actions.

> 💡 it means here dispatch has reducer function which checks type and if here we have to intialize mutiple values to states then useReducer is required to main those am i true ?

Yes, you're on the right track! Let's break it down more clearly:

How `dispatch` Works with the Reducer

The `dispatch` function does not directly contain the reducer function, but it is tightly connected to it. Here's how it works:

1. `dispatch` **Function**:
   - When you call `dispatch(action)`, React internally calls the reducer function, passing the current state and the action as arguments.
2. **Reducer Function**:
   - The reducer function takes the current state and the action, processes them, and returns a new state based on the action type.

Managing Multiple State Values

Khushal Savalakha

`useReducer` is particularly useful when you have complex state logic or multiple state values that are related. Here's why:

1. **Single State Object**:
   - With `useReducer`, you can manage multiple state values in a single state object. This makes it easier to keep related state values together and manage them in a more organized way.
2. **Complex State Logic**:
   - If the logic for updating your state is complex (e.g., it involves multiple conditions or multiple steps), using a reducer function helps keep that logic encapsulated and easier to manage.

Example with Multiple State Values

Suppose you have a component that needs to manage two related state values: `count` and `text`.

Step-by-Step Example

1. **Define Initial State**:
   - Define an initial state object that contains all the state values you want to manage.

```
const initialState = { count: 0, text: '' };
```

2. **Define Reducer Function**:
   - The reducer function takes the current state and an action, and returns a new state based on the action type.

```
const reducer = (state, action) => {
    switch (action.type) {
        case 'increment':
            return { ...state, count: state.count +
1 };
        case 'decrement':
            return { ...state, count: state.count -
1 };
        case 'setText':
```


Khushal Savalakha

```
            return { ...state, text: action.payload
};
        default:
            throw new Error('Unsupported action typ
e');
    }
};
```

3. **Use** `useReducer`:
   - Call `useReducer` with the reducer function and the initial state.

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

4. **Dispatch Actions**:
   - Use the `dispatch` function to send actions to the reducer.

```
const increment = () => dispatch({ type: 'incremen
t' });
const decrement = () => dispatch({ type: 'decremen
t' });
const setText = (newText) => dispatch({ type: 'setT
ext', payload: newText });
```

5. **Render Component**:
   - Use the state values in your component.

```
return (
    <div>
        <h1>Count: {state.count}</h1>
        <button onClick={increment}>+</button>
        <button onClick={decrement}>-</button>
        <input
            type="text"
            value={state.text}
            onChange={(e) => setText(e.target.valu
e)}
```


Khushal Savalakha

```
            />
        </div>
    );
```

Full Example Code

Here's the complete example in one place:

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

const initialState = { count: 0, text: '' };

const reducer = (state, action) => {
    switch (action.type) {
        case 'increment':
            return { ...state, count: state.count + 1
};
        case 'decrement':
            return { ...state, count: state.count - 1
};
        case 'setText':
            return { ...state, text: action.payload };
        default:
            throw new Error('Unsupported action typ
e');
    }
};

const App = () => {
    const [state, dispatch] = useReducer(reducer, init
ialState);

    const increment = () => dispatch({ type: 'incremen
t' });
    const decrement = () => dispatch({ type: 'decremen
t' });
    const setText = (newText) => dispatch({ type: 'set
Text', payload: newText });
```


Khushal Savalakha

```
    return (
        <div>
            <h1>Count: {state.count}</h1>
            <button onClick={increment}>+</button>
            <button onClick={decrement}>-</button>
            <input
                type="text"
                value={state.text}
                onChange={(e) => setText(e.target.valu
e)}
            />
        </div>
    );
};

export default App;
```

Summary

- `dispatch` **Function**: When called, it triggers the reducer function with the current state and the provided action.
- **Managing Multiple State Values**: `useReducer` is useful for managing multiple related state values in a single state object, making state management more organized and predictable.
- **Reducer Logic**: Encapsulates complex state update logic, making the component easier to understand and maintain.

By using `useReducer`, you can effectively manage complex state and actions, keeping your component logic clear and well-organized.

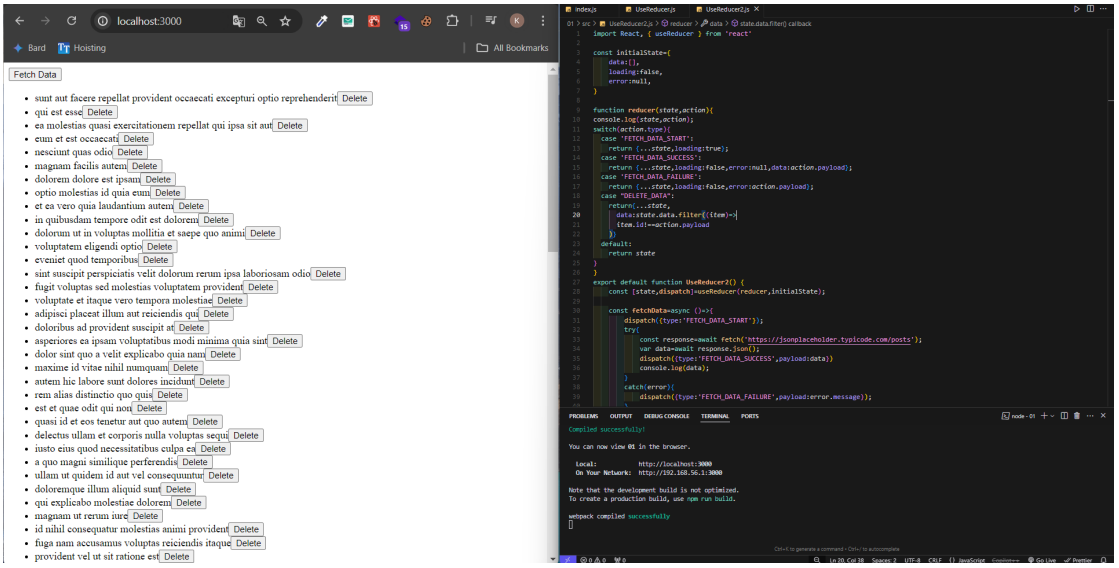
```
import React, { useReducer } from 'react'

const initialState={
    data:[],
    loading:false,
    error:null,
}
```


Khushal Savalakha

```
function reducer(state,action){
console.log(state,action);
switch(action.type){
  case 'FETCH_DATA_START':
    return {...state,loading:true};
  case 'FETCH_DATA_SUCCESS':
    return {...state,loading:false,error:null,data:actio
  case 'FETCH_DATA_FAILURE':
    return {...state,loading:false,error:action.payload}
  case "DELETE_DATA":
    return{...state,
      data:state.data.filter((item)=>
      item.id!==action.payload
    )}
  default:
    return state
}
}
export default function UseReducer2() {
    const [state,dispatch]=useReducer(reducer,initialSta

    const fetchData=async ()=>{
        dispatch({type:'FETCH_DATA_START'});
        try{
            const response=await fetch('https://jsonplac
            var data=await response.json();
            dispatch({type:'FETCH_DATA_SUCCESS',payload:
            console.log(data);
        }
        catch(error){
            dispatch({type:'FETCH_DATA_FAILURE',payload:
        }
    }

    const deleteData=(id)=>{
      dispatch({type:"DELETE_DATA",payload:id})
    }
  return (
```


Khushal Savalakha

```
      <div>
        <button onClick={fetchData}>Fetch Data</button>
        {state.loading && <p>Loading...</p>}
        {state.error && <p>Error:{state.error}</p>}
        <ul>
          { state.data.map((item)=>(
            <li key={item.id}>
              {item.title}
              <button onClick={()=>deleteData(item.id)}>De
            </li>
          ))}
        </ul>
      </div>
    )
  }
```



Sure, let's break down how this code works step by step:

1. Initial Setup

- **Imports**:
  - `useReducer` from React: This hook is used for managing complex state logic in a functional component.
- **Initial State**:

Khushal Savalakha

```
const initialState = {
    data: [],
    loading: false,
    error: null,
};
```

- `data`: An array to store fetched data.
- `loading`: A boolean to indicate the loading state.
- `error`: A variable to store any error messages.

2. Reducer Function

The `reducer` function takes the current state and an action, and returns a new state based on the action type:

```
function reducer(state, action) {
    console.log(state, action);
    switch (action.type) {
        case 'FETCH_DATA_START':
            return { ...state, loading: true };
        case 'FETCH_DATA_SUCCESS':
            return { ...state, loading: false, error:
null, data: action.payload };
        case 'FETCH_DATA_FAILURE':
            return { ...state, loading: false, error:
action.payload };
        case 'DELETE_DATA':
            return {
                ...state,
                data: state.data.filter((item) => ite
m.id !== action.payload),
            };
        default:
            return state;
    }
}
```

- **Action Types**:

Khushal Savalakha

- `FETCH_DATA_START`: Sets `loading` to `true`.
- `FETCH_DATA_SUCCESS`: Sets `loading` to `false`, clears any error, and updates `data` with the payload.
- `FETCH_DATA_FAILURE`: Sets `loading` to `false` and updates `error` with the payload.
- `DELETE_DATA`: Filters out the item with the given `id` from `data`.

3. Component: `UseReducer2`

The component utilizes the `useReducer` hook and renders UI elements:

```
export default function UseReducer2() {
    const [state, dispatch] = useReducer(reducer, init
ialState);

    const fetchData = async () => {
        dispatch({ type: 'FETCH_DATA_START' });
        try {
            const response = await fetch('<https://jso
nplaceholder.typicode.com/posts>');
            const data = await response.json();
            dispatch({ type: 'FETCH_DATA_SUCCESS', pay
load: data });
            console.log(data);
        } catch (error) {
            dispatch({ type: 'FETCH_DATA_FAILURE', pay
load: error.message });
        }
    };

    const deleteData = (id) => {
        dispatch({ type: 'DELETE_DATA', payload: id
});
    };

    return (
        <div>
            <button onClick={fetchData}>Fetch Data</bu
```


Khushal Savalakha

```
tton>
            {state.loading && <p>Loading...</p>}
            {state.error && <p>Error: {state.error}</p
>}
            <ul>
                {state.data.map((item) => (
                    <li key={item.id}>
                        {item.title}
                        <button onClick={() => deleteD
ata(item.id)}>Delete</button>
                    </li>
                ))}
            </ul>
        </div>
    );
}
```

How It Works

1. **useReducer Hook**:
   - Initializes state with `initialState`.
   - Provides `state` and `dispatch` for managing state updates.
2. **fetchData Function**:
   - Dispatches `FETCH_DATA_START` to indicate loading.
   - Fetches data from the API.
   - Dispatches `FETCH_DATA_SUCCESS` with the fetched data on success.
   - Dispatches `FETCH_DATA_FAILURE` with an error message on failure.
3. **deleteData Function**:
   - Dispatches `DELETE_DATA` with the ID of the item to be deleted.
4. **Rendering**:
   - Renders a button to fetch data.
   - Displays a loading message if `state.loading` is `true`.
   - Displays an error message if `state.error` is not `null`.

Khushal Savalakha

- Renders a list of fetched items, each with a delete button.

Key Points

- **Reducer Logic**: Manages complex state updates in a predictable manner.
- **Dispatch Function**: Used to trigger state updates based on actions.
- **Component State**: Managed via the `useReducer` hook to handle asynchronous data fetching and error handling efficiently.

By following these steps, the component effectively fetches data from an API, manages loading and error states, and allows for the deletion of items from the fetched data.

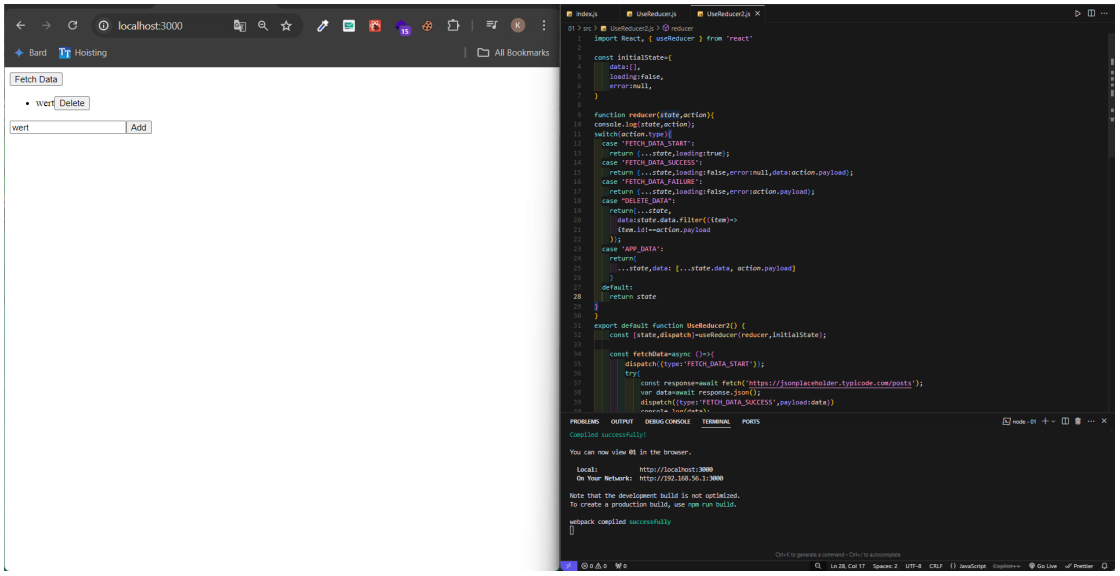
```
import React, { useReducer } from 'react'

const initialState={
    data:[],
    loading:false,
    error:null,
}

function reducer(state,action){
console.log(state,action);
switch(action.type){
  case 'FETCH_DATA_START':
    return {...state,loading:true};
  case 'FETCH_DATA_SUCCESS':
    return {...state,loading:false,error:null,data:actio
  case 'FETCH_DATA_FAILURE':
    return {...state,loading:false,error:action.payload}
  case "DELETE_DATA":
    return{...state,
      data:state.data.filter((item)=>
      item.id!==action.payload
    )};//It will return data and set to state internally
  case 'APP_DATA':
    return{
      ...state,data: [...state.data, action.payload]
    }
```


Khushal Savalakha

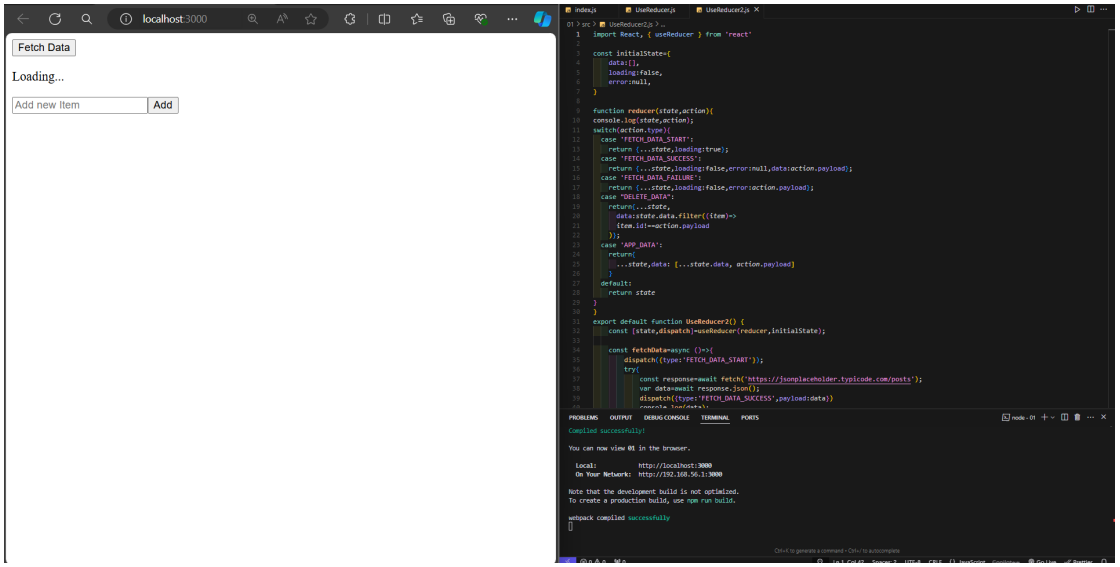
```
  default:
    return state
}
}
export default function UseReducer2() {
    const [state,dispatch]=useReducer(reducer,initialSta

    const fetchData=async ()=>{
        dispatch({type:'FETCH_DATA_START'});
        try{
            const response=await fetch('https://jsonplac
            var data=await response.json();
            dispatch({type:'FETCH_DATA_SUCCESS',payload:
            //here reducer function is executed when dis
            console.log(data);
        }
        catch(error){
            dispatch({type:'FETCH_DATA_FAILURE',payload:
        }
    }

    const deleteData=(id)=>{
      dispatch({type:"DELETE_DATA",payload:id})
    }

    const addData=(newData)=>{
      dispatch({type:'APP_DATA',payload:newData})
    }
  return (
    <div>
      <button onClick={fetchData}>Fetch Data</button>
      {state.loading && <p>Loading...</p>}
      {state.error && <p>Error:{state.error}</p>}
      <ul>
        { state.data.map((item)=>(
          <li key={item.id}>
            {item.title}
            <button onClick={()=>deleteData(item.id)}>De
```

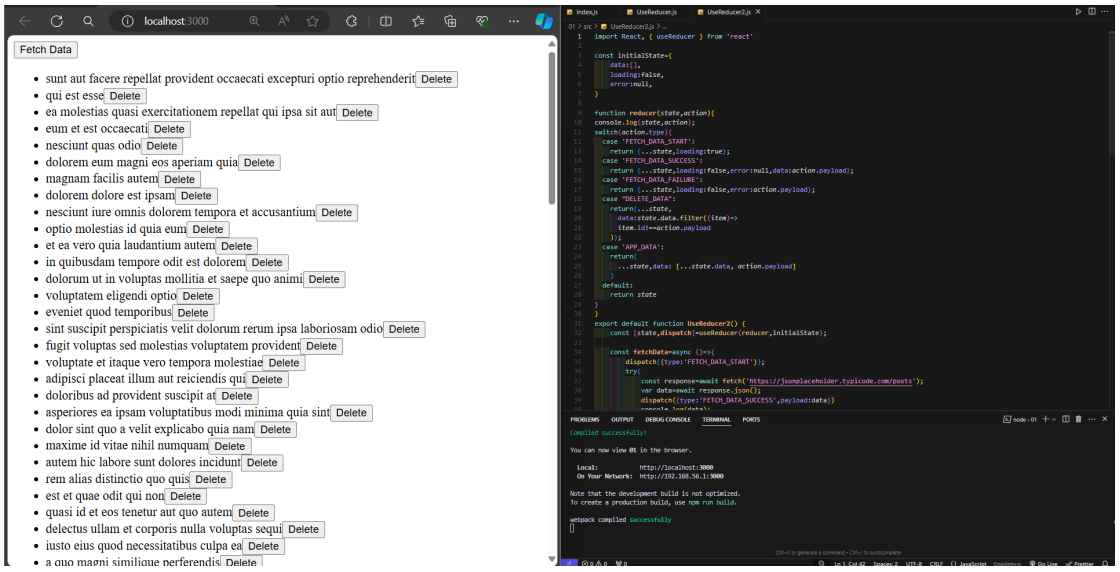

Khushal Savalakha

```
            </li>
          ))}
        </ul>
        <form onSubmit={(e)=>{
          e.preventDefault();
          addData(
            {
              id:Date.now(),
              title:e.target.title.value,
            }
          )
        }}>
          <input type='text' name='title' placeholder='Add
          <button type='submit'>Add</button>
        </form>
      </div>
    )
  }
```



Khushal Savalakha






```
Import {useReducer} from 'react';
const initialState = {count: 0};
function reducer(state, action) {
 switch (action.type) {
 case 'increment':
 return {count: state.count + action.payload};
 case 'decrement':
 return {count: state.count - action.payload};
 default:
 throw new Error();
 }
}
```


Khushal Savalakha

```
function Counter() {
 const [state, dispatch] = useReducer(reducer, initialSt
 return (
 <>
 Count: {state.count}
 <button onClick={() => dispatch({type: 'decrement', pay
button>
 <button onClick={() => dispatch({type: 'increment', pay
button>
 </>
 );
}
export default Counter;
```

> 💡 But here when dispatch called reducer interllay it goes to switch case and that time it will return something but how it will set that return data to state?

In the `useReducer` hook, when you dispatch an action, it triggers the reducer function. The reducer function takes the current state and the dispatched action as its arguments. Based on the action type, it computes the new state and returns it. The `useReducer` hook then takes care of updating the state with this new state.

Here's a step-by-step explanation of how this works:

1. **Initial Setup**:
   - `useReducer` is initialized with a reducer function and an initial state (`initialState`).
   - It returns the current state and a `dispatch` function to send actions to the reducer.
2. **Dispatching an Action**:
   - When you call `dispatch({ type: 'increment', payload: 4 })`, the `dispatch` function sends the action object to the reducer.
3. **Executing the Reducer**:
   - The reducer function is called with the current state and the action object.

Khushal Savalakha

- The reducer function uses a switch statement to determine the action type.
- Depending on the action type, it computes and returns a new state object.

4. **Updating the State**:
   - The `useReducer` hook automatically updates the state with the new state returned by the reducer function.
   - The component re-renders with the updated state.

Code Flow Example

1. **Initial State**:

```
const initialState = { count: 0 };
```

2. **Reducer Function**:

```
function reducer(state, action) {
  switch (action.type) {
    case 'increment':
      return { count: state.count + action.payload
};
    case 'decrement':
      return { count: state.count - action.payload
};
    default:
      throw new Error();
  }
}
```

3. **Using `useReducer`**:

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

4. **Dispatching an Action**:

```
<button onClick={() => dispatch({ type: 'decremen
t', payload: 4 })}>-4</button>
```


Khushal Savalakhia

```
<button onClick={() => dispatch({ type: 'incremen
t', payload: 4 })}>+4</button>
```

5. **Reducer Execution** (when `dispatch` is called):
   - For `dispatch({ type: 'increment', payload: 4 })`:
     - The reducer function is called with `state = { count: 0 }` and `action = { type: 'increment', payload: 4 }`.
     - It matches the 'increment' case and returns a new state: `{ count: 0 + 4 }` which is `{ count: 4 }`.
   - The `useReducer` hook updates the state to `{ count: 4 }`.
6. **Component Re-render**:
   - The component re-renders with the updated state, showing `Count: 4`.

The `useReducer` hook handles the state update process internally, ensuring that the component always has the latest state after the reducer function computes the new state.

> 💡 can you write it in short way by making any table and fill data which should be inside default function and other ?

Sure, here's a concise table summarizing what should be inside the component function and what should be outside:

Outside the Component Function	**Inside the Component Function**
Initial State	**useReducer Hook**
``const initialState = {	`const [state, dispatch] = useReducer(reducer, initialState);`
data: [],	
loading: false,	
error: null,	
};```	
Reducer Function	**Component Logic and Handlers**
``function reducer(state, action) {	```const fetchData = async () => {
switch (action.type) {	dispatch({ type: 'FETCH_DATA_START' });

Khushal Savalakha

case 'FETCH_DATA_START':	try {
return { ...state, loading: true };	const response = await fetch('https://jsonplaceholder.typicode.com/posts');
case 'FETCH_DATA_SUCCESS':	var data = await response.json();
return { ...state, loading: false, error: null, data: action.payload };	dispatch({ type: 'FETCH_DATA_SUCCESS', payload: data });
case 'FETCH_DATA_FAILURE':	console.log(data);
return { ...state, loading: false, error: action.payload };	} catch (error) {
case 'DELETE_DATA':	dispatch({ type: 'FETCH_DATA_FAILURE', payload: error.message });
return { ...state, data: state.data.filter(item => item.id !== action.payload) };	}
case 'APP_DATA':	};```
return { ...state, data: [...state.data, action.payload] };	```const deleteData = (id) => {
default:	dispatch({ type: 'DELETE_DATA', payload: id });
return state;	};```
}	```const addData = (newData) => {
}```	dispatch({ type: 'APP_DATA', payload: newData });
	};```
	JSX/Return Statement
	``return (
	<button onClick={fetchData}>Fetch Data</button>
	{state.loading && Loading... }
	{state.error && Error: {state.error} }
	<ul>
	{state.data.map((item) => (
	<li key={item.id}>
	{item.title}

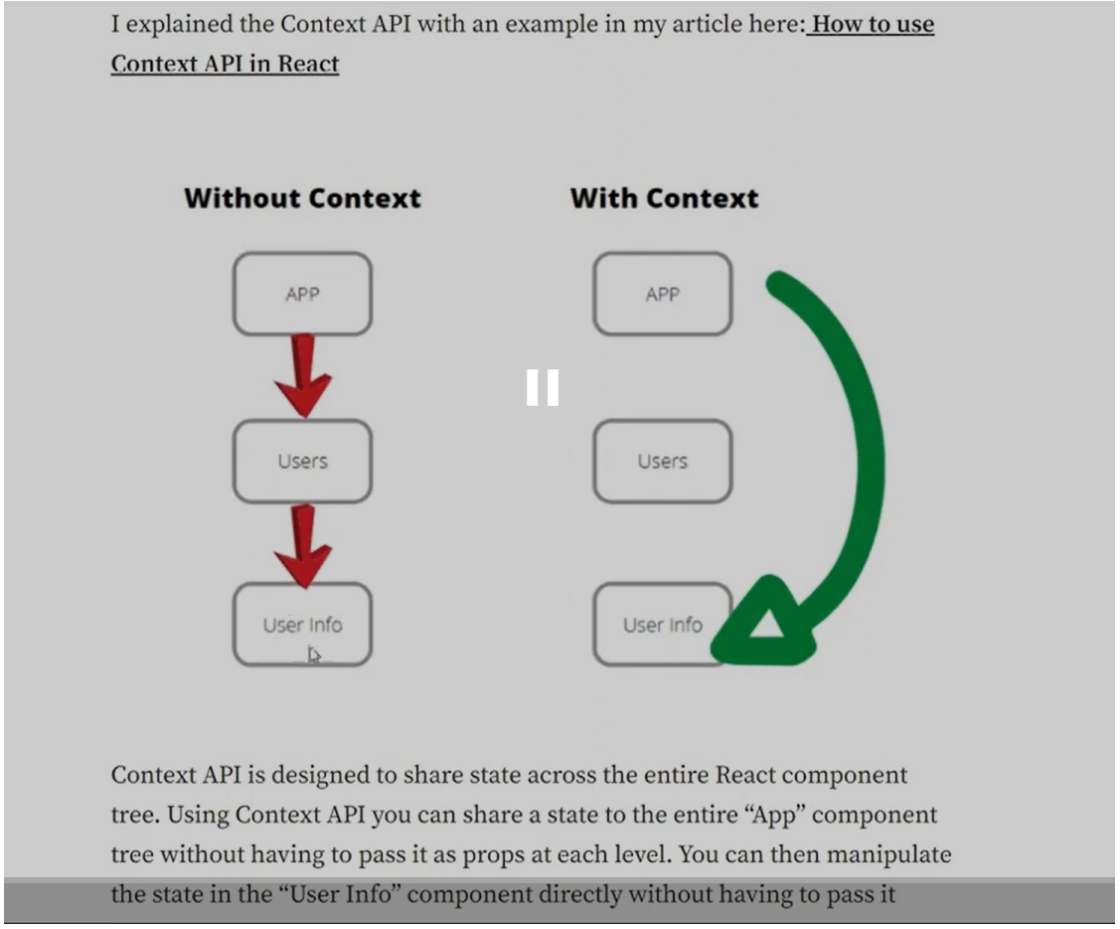
Khushal Savalakha

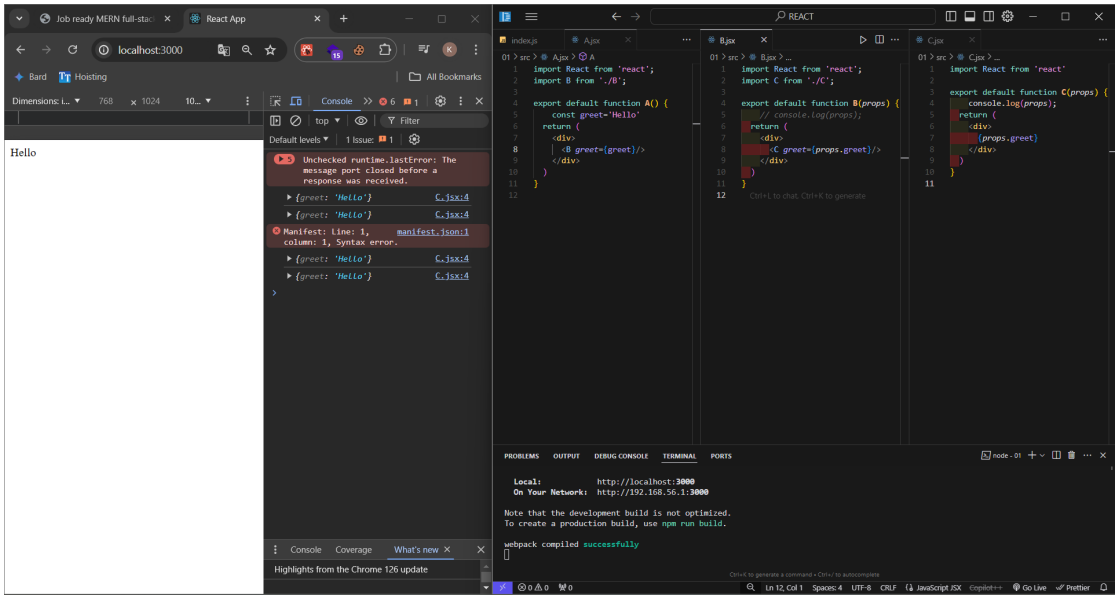
	<button onClick={() => deleteData(item.id)}>Delete</button>
	</li>
	))}
	</ul>
	<form onSubmit={(e) => {
	e.preventDefault();
	addData({
	id: Date.now(),
	title: e.target.title.value,
	});
	}}>
	<input type='text' name='title' placeholder='Add new Item' />
	<button type='submit'>Add</button>
	</form>
	);```

This table provides a clear distinction of what should be defined inside and outside of the component function.

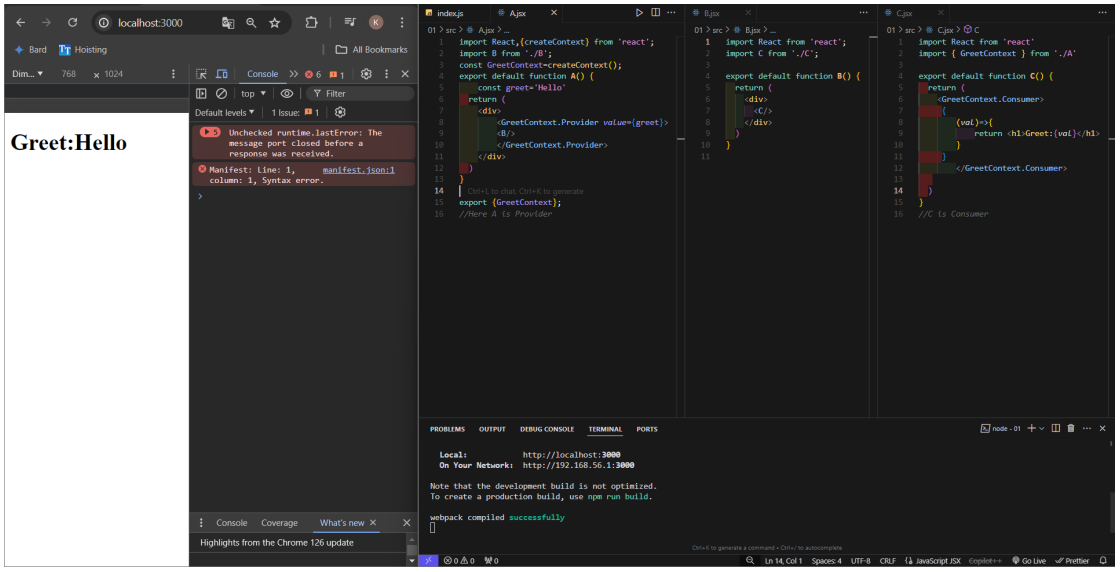
▼ Context API

Khushal Savalakha





Here we want to send the data to C but there is no need of data in B but still we have to send data like in flow A→B→C through props this is not a good way.



Based on the code snippets and the image provided, the code uses React's Context API to manage and pass down data through the component tree without having to pass props manually at every level. Here's an explanation of the code structure:

Index.js

```
import React from 'react';
import ReactDOM from 'react-dom/client';
import A from './A';

const root = ReactDOM.createRoot(document.getElementById

root.render(
  <React.StrictMode>
    <A/>
  </React.StrictMode>
);
```

A.jsx

```
import React, { createContext } from 'react';
import B from './B';
const GreetContext = createContext();
const GreetContext2 = createContext();
```


Khushal Savalakha

```
export default function A() {
  const greet = 'Hello';
  const greet2 = 'Hello2';
  return (
    <div>
      <GreetContext.Provider value={greet}>
        <GreetContext2.Provider value={greet2}>
          <B />
        </GreetContext2.Provider>
      </GreetContext.Provider>
    </div>
  );
}

export { GreetContext, GreetContext2 };
//Here A is Provider
```

B.jsx

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

export default function B() {
  return (
    <div>
      <C />
    </div>
  );
}
```

C.jsx

```
import React from 'react';
import { GreetContext, GreetContext2 } from './A';

export default function C() {
  return (
```


Khushal Savalakha

```
    <GreetContext.Consumer>
      {(val) => (
        <GreetContext2.Consumer>
          {(val2) => {
            return <h1>Greet:{val} ,{val2}</h1>;
          }}
        </GreetContext2.Consumer>
      )}
    </GreetContext.Consumer>
  );
}
//C is Consumer
```

Explanation

1. **index.js (Component A)**:
   - Imports `createContext` from React to create two contexts: `GreetContext` and `GreetContext2`.
   - Creates two context providers, wrapping component `B` inside them, passing `greet` and `greet2` as their values respectively.
   - Exports the contexts and the component `A`.
2. **B.jsx (Component B)**:
   - Imports React.
   - Imports and renders component `C`.
   - Acts as a middle layer, it doesn't directly deal with the context.
3. **C.jsx (Component C)**:
   - Imports the contexts `GreetContext` and `GreetContext2` from `A`.
   - Uses `GreetContext.Consumer` to access the value of `GreetContext` and `GreetContext2.Consumer` to access the value of `GreetContext2`.
   - Renders an `<h1>` element displaying the values from both contexts.

How It Works

- **Component A**: Defines and provides context values (`greet` and `greet2`).

Khushal Savalakha

- **Component B**: Acts as a bridge and renders `C`.
- **Component C**: Consumes the context values provided by `A` and renders them.

Output

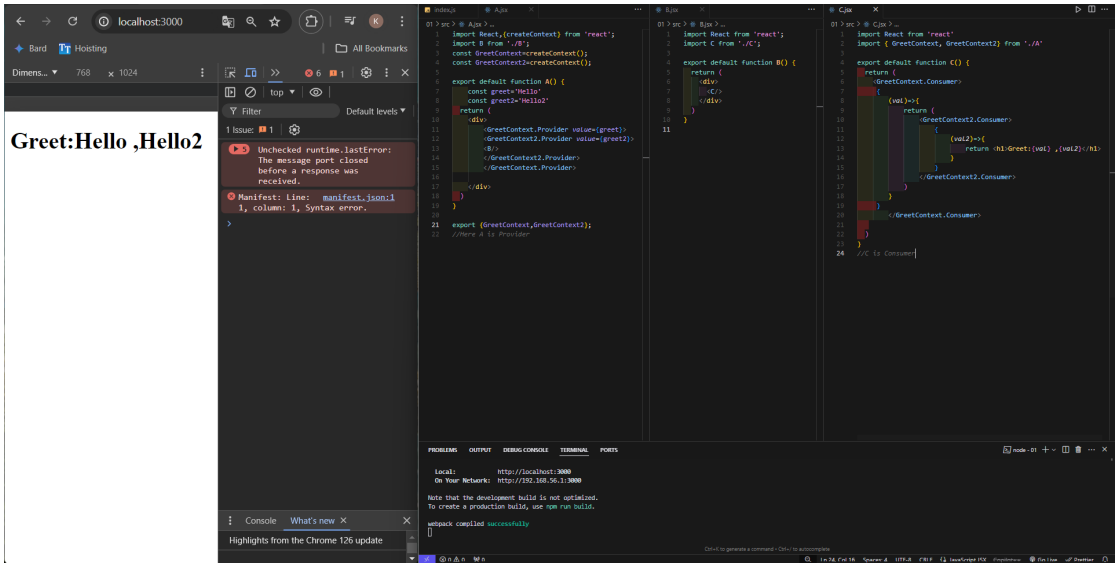
The output of this code is a webpage displaying:

```
Greet:Hello ,Hello2
```

This is because the values `Hello` and `Hello2` are provided by the context providers and consumed in `C`.

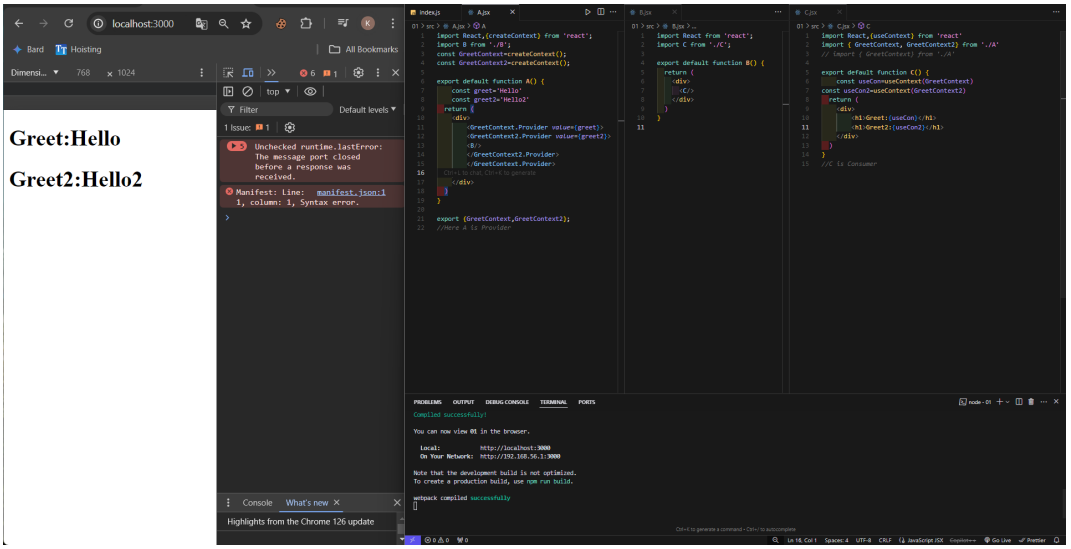
Error in the Console

The error "Unchecked runtime.lastError: The message port closed before a response was received" and "Manifest: Line: 1, column: 1, Syntax error." seems related to a browser extension rather than your code. Ensure your extension's manifest file is correctly formatted. This shouldn't affect the working of your React application.



▼ useContext

Khushal Savalakha



Based on the new image and updated code, let's break down how the updated React code is functioning.

A.jsx

```
import React, { createContext } from 'react';
import B from './B';
const GreetContext = createContext();
const GreetContext2 = createContext();

export default function A() {
  const greet = 'Hello';
  const greet2 = 'Hello2';
  return (
    <div>
      <GreetContext.Provider value={greet}>
        <GreetContext2.Provider value={greet2}>
          <B />
        </GreetContext2.Provider>
      </GreetContext.Provider>
    </div>
  );
}

export { GreetContext, GreetContext2 };
//Here A is Provider
```


Khushal Savalakha

B.jsx

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

export default function B() {
  return (
    <div>
      <C />
    </div>
  );
}
```

C.jsx

```
import React, { useContext } from 'react';
import { GreetContext, GreetContext2 } from './A';

export default function C() {
  const useCon = useContext(GreetContext);
  const useCon2 = useContext(GreetContext2);
  return (
    <div>
      <h1>Greet:{useCon}</h1>
      <h1>Greet2:{useCon2}</h1>
    </div>
  );
}
//C is Consumer
```

Advantages of `useContext`

- **Simplicity**: `useContext` provides a more straightforward way to consume context values compared to the traditional `Context.Consumer`.
- **Readability**: The code is more concise and easier to read, especially in larger components or when using multiple contexts.

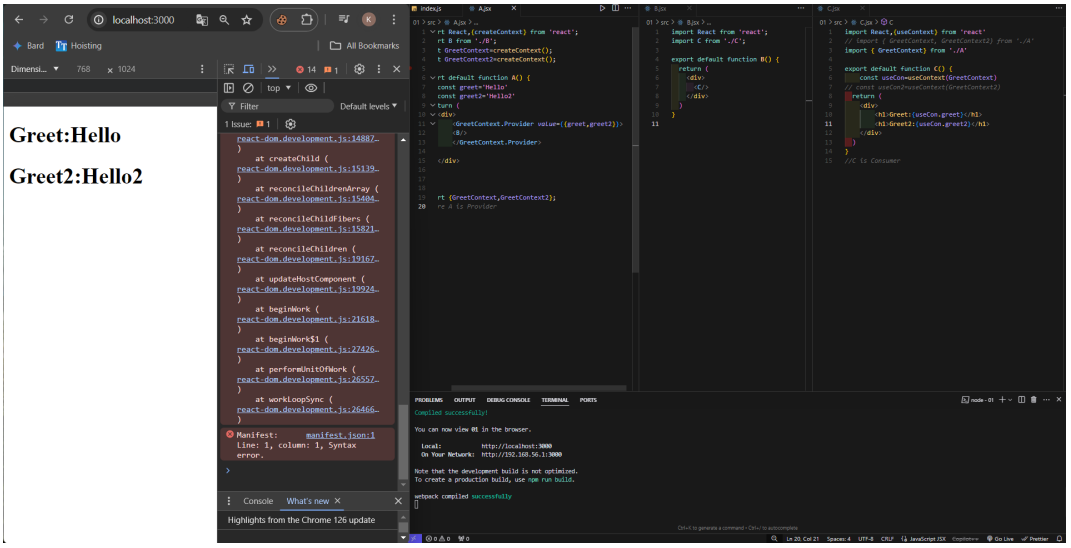
Khushal Savalakhia

- **Functional Components**: Works seamlessly with functional components, embracing the modern React paradigm.

When to Use `useContext`

- **Global State Management**: When you need to manage state or data globally and pass it through the component tree.
- **Theming**: For passing down theme-related information.
- **Localization**: For managing language preferences across the application.
- **Authentication**: For passing user authentication status and data.

In summary, `useContext` is a powerful hook that simplifies consuming context values in functional components, making your React code cleaner and more maintainable.



Your updated code structure nests `greet` and `greet2` within a single context object `GreetContext`. Now, instead of using two separate contexts (`GreetContext` and `GreetContext2`), you can pass both values in a single context. This approach simplifies context management when multiple values need to be passed together.

Updated Code Explanation using Object

A.jsx

```
import React, { createContext } from 'react';
import B from './B';
```


Khushal Savalakha

```
const GreetContext = createContext();

export default function A() {
    const greet = 'Hello';
    const greet2 = 'Hello2';

    return (
        <div>
            <GreetContext.Provider value={{ greet,
greet2 }}>
                <B />
            </GreetContext.Provider>
        </div>
    );
}

export { GreetContext };
//Here A is Provider
```

B.jsx

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

export default function B() {
  return (
    <div>
      <C />
    </div>
  );
}
```

C.jsx

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


Khushal Savalakha

```
export default function C() {
  const { greet, greet2 } = useContext(GreetContex
t);

  return (
    <div>
      <h1>Greet: {greet}</h1>
      <h1>Greet2: {greet2}</h1>
    </div>
  );
}
```

Detailed Breakdown

1. **Creating Context**:
   - `const GreetContext = createContext();`
     This line creates a single context object `GreetContext`.
2. **Providing Context Values**:
   - `GreetContext.Provider value={{ greet, greet2 }}`
     In the `A` component, `GreetContext.Provider` wraps `B` and provides an object containing `greet` and `greet2` as its value. This makes the `greet` and `greet2` values available to any nested component within the provider.
3. **Consuming Context Values with `useContext`**:
   - `const { greet, greet2 } = useContext(GreetContext);`
     Inside the `C` component, `useContext(GreetContext)` is used to access the context value. This returns an object with properties `greet` and `greet2`, which are then destructured and used within the component.

Advantages of Using a Single Context

- **Simplicity**: Managing multiple related values within a single context reduces the complexity of your code.
- **Efficiency**: Only one context provider and consumer are needed, simplifying the component tree.

Khushal Savalakha

- **Consistency**: Grouping related data together ensures they are always consumed together, reducing the risk of mismatch or errors.

Output

The output of this code is a webpage displaying:

```
Greet: Hello
Greet2: Hello2
```

This is because the values `Hello` and `Hello2` are provided by the `GreetContext.Provider` and consumed in `C`.

Summary

Using `useContext` with a single context object that holds multiple related values is a clean and efficient way to manage and consume global data in React. It simplifies the code, reduces redundancy, and ensures that related context values are always accessed together.

▼ useLocation

The `useLocation` hook in React is used to access the current URL's location data in components. It's part of the `react-router-dom` library, which is used for routing in React applications.

How `useLocation` Works:

When you navigate between different routes in a React app (like changing URLs), the `useLocation` hook allows you to access information about the current location, such as:

- **pathname**: The current URL path (like `/home`, `/about`).
- **search**: The query string part of the URL (like `?id=123`).
- **hash**: The hash fragment (like `#section1`).
- **state**: Data passed along with navigation (like form inputs or custom data).

The `useLocation` hook provides access to this information, allowing components to react to changes in the URL.

Syntax:

Khushal Savalakha

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

const location = useLocation();
```

Key Properties Returned by `useLocation`:

1. **pathname**: The path of the current URL.
2. **search**: The query string in the URL.
3. **hash**: The URL fragment.
4. **state**: Custom state passed when navigating between pages (optional).

Example:

Let's look at a simple example of using `useLocation` in a React app.

App Component with Routes:

```
import { BrowserRouter as Router, Routes, Route, Link
} from 'react-router-dom';
import Home from './Home';
import About from './About';

function App() {
  return (
    <Router>
      <nav>
        <Link to="/">Home</Link>
        <Link to="/about">About</Link>
      </nav>

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


Khushal Savalakha

Home Component Using `useLocation`:

```
import React from 'react';
import { useLocation } from 'react-router-dom';

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

  return (
    <div>
      <h1>Home Page</h1>
      <p>Current Path: {location.pathname}</p>
      <p>Search Query: {location.search}</p>
      <p>Hash: {location.hash}</p>
    </div>
  );
}

export default Home;
```

About Component:

```
function About() {
  return <h1>About Page</h1>;
}

export default About;
```

Explanation:

1. **App Component**: It uses `react-router-dom` to define routes. The `<Link>` components are used for navigation between "Home" and "About" pages.
2. **Home Component**: This component uses the `useLocation` hook to access the location object. We display the `pathname`, `search`, and `hash` values.
   - When the user navigates to the Home page, the current URL's path (`pathname`), query parameters (`search`), and any hash (`hash`) are displayed.

Khushal Savalakha

How to Use:

- If you visit `/`, you will see:

```
Current Path: /
Search Query:
Hash:
```

- If you visit `/about`, you will see only the About page.
- If you visit `/about?name=John#info`, and modify the code similarly, you can extract the query and hash parts:

```
Current Path: /about
Search Query: ?name=John
Hash: #info
```

Use Cases:

1. **Reading query parameters**: Extracting data from the URL like user IDs or filters (`?id=123`).
2. **Tracking user navigation**: Knowing which part of the app the user is currently on by observing `location.pathname`.
3. **Custom state handling**: Passing additional state when navigating and accessing it in the new component via `location.state`.

Summary:

- `useLocation` lets you access the URL's current location data.
- It's useful for handling query strings, current path, and custom navigation state.

💡 explain how to pass state and access state ?

Imagine you are building a travel website where users can browse different destinations. On the homepage, they click on a destination card, and you want to pass some details about that destination (like name, image, and description) to the next page without putting it in the URL or using global state.

Khushal Savalakha

This is where passing state comes in handy, like a traveler handing over a ticket at every stop of the journey, ensuring everything moves smoothly from one point to another.

The Setup:

You have two components in your app: the **Home** page where users can select destinations and the **Details** page where the information about the chosen destination is displayed.

Now, you want to pass the destination data from the **Home** page to the **Details** page when a user clicks on a destination.

Step 1: Passing the State

When a user clicks on a destination, we use `react-router-dom`'s `Link` or `navigate` to pass the destination details as state.

Here's how it looks:

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

function Home() {
  const destination = {
    name: "Paris",
    image: "paris.jpg",
    description: "The city of lights!",
  };

  return (
    <div>
      <h1>Choose Your Destination</h1>
      <Link
        to="/details"
        state={destination} // Passing state to the ne
xt page
      >
        <button>Explore Paris</button>
      </Link>
    </div>
  );
```


Khushal Savalakha

```
}

export default Home;
```

In this code, when the "Explore Paris" button is clicked, we use the `state` prop inside the `Link` component to pass the `destination` object (which contains the name, image, and description of Paris) to the **Details** page. The traveler has handed over the "ticket" containing all the necessary details for the journey!

Step 2: Accessing the State

Now that the state has been passed, the **Details** page can receive and use it. On the **Details** page, we can access this state using the `useLocation` hook.

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

function Details() {
  const location = useLocation();
  const { name, image, description } = location.state
|| {}; // Accessing state

  return (
    <div>
      <h1>{name}</h1>
      <img src={image} alt={name} />
      <p>{description}</p>
    </div>
  );
}

export default Details;
```

Here, we use `useLocation` to grab the state. The `location.state` holds the data that was passed from the **Home** page. If Paris was chosen, the details of Paris (name, image, and description) will be displayed.

If the state is undefined (maybe someone landed on the page directly), we safeguard against errors by using a fallback with `|| {}`.

What's Happening:

Khushal Savalakha

1. **Passing State**: On the **Home** page, when the user clicks the button, the destination details are passed through the `state` prop of the `Link` component.
2. **Receiving State**: On the **Details** page, we use `useLocation` to access the `state` object. This contains all the details about the destination that we passed earlier.

The Magic of State Passing:

Think of this process like a traveler moving through an airport. At every checkpoint, they need to show their ticket (the state), which holds all the necessary information for where they are going and what they will do next. It's a smooth handoff from one page to the next without needing to modify the URL or use external storage.

This ability to pass state makes navigation in React more dynamic and context-aware, allowing different pages to "communicate" as the user moves through the app.

Now, every time someone explores a new destination on your travel site, you can smoothly pass information about their selection between pages, creating a seamless experience!

App.jsx:

```
import React from 'react'
import { BrowserRouter,Routes,Route,Link } from 'react-r
import Home from './Home'
import About from './About'

export default function App() {
  return (
    <BrowserRouter>
    <nav>
      <Link to="/">Home</Link>
      <Link to="/about">About</Link>
    </nav>
    <Routes>
        <Route path='/' element={<Home/>}></Route>
        <Route path='/about' element={<About/>}></Route>
    </Routes>
    </BrowserRouter>
  )
}
```


Khushal Savalakha

Home.jsx:

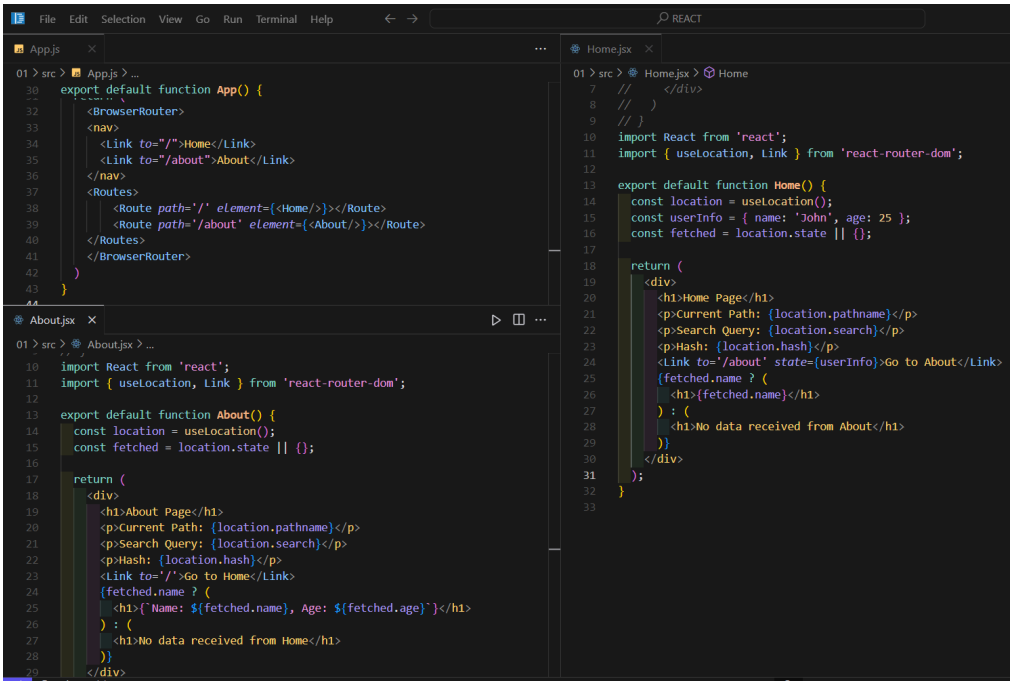
```
import React from 'react';
import { useLocation, Link } from 'react-router-dom';

export default function Home() {
  const location = useLocation();
  const userInfo = { name: 'John', age: 25 };
  const fetched = location.state || {};

  return (
    <div>
      <h1>Home Page</h1>
      <p>Current Path: {location.pathname}</p>
      <p>Search Query: {location.search}</p>
      <p>Hash: {location.hash}</p>
      <Link to='/about' state={userInfo}>Go to About</Li
      {fetched.name ? (
        <h1>{fetched.name}</h1>
      ) : (
        <h1>No data received from About</h1>
      )}
    </div>
  );
}
```

About.jsx:

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

export default function About() {
  const location = useLocation();
  const fetched = location.state || {};

  return (
    <div>
      <h1>About Page</h1>
```


Khushal Savalakha

```
      <p>Current Path: {location.pathname}</p>
      <p>Search Query: {location.search}</p>
      <p>Hash: {location.hash}</p>
      <Link to='/'>Go to Home</Link>
      {fetched.name ? (
        <h1>{`Name: ${fetched.name}, Age: ${fetched.age}
      ) : (
        <h1>No data received from Home</h1>
      )}
    </div>
  );
}
```



HomeAbout

Home Page

Current Path: /

Search Query:

Hash:

Go to About

No data received from About

Khushal Savalakha

HomeAbout

About Page

Current Path: /about

Search Query:

Hash:

Go to Home

Name: John, Age: 25

> 💡 it is necessory to write state ={} while go another page or we can use any word?

You can use any valid key name while passing state in `Link` or navigating between pages in React Router. The key name `state` is not mandatory, but it is the default and most commonly used. You can pass any object with any key names, as long as you're consistent when accessing it on the receiving page.

Example with Custom Key:

1. **Passing Custom State:**

```
<Link to="/about" state={{ name: 'John', age: 25 }}>Go to About</Link>
```

In this example, we're passing an object with `name` and `age` as keys in the `state`.

1. **Using a Custom Key Instead of `state`:**

Khushal Savalakha

You can use any custom word instead of `state`. For example:

```
<Link to="/about" myData={{ name: 'John', age: 25 }}>Go to About</Link>
```

But note that **React Router only supports** `state` as a reserved prop in the `Link` component. So while `state={{}}` is automatically available on the receiving route through `location.state`, if you use something like `myData`, it won't be passed correctly.

Accessing the Custom State on the Next Page:

On the target page (like `About.jsx`), when you use `location.state`, you can access whatever you passed in `state`. For example:

```
const location = useLocation();
const fetched = location.state || {};  // Accessing state directly

return (
  <div>
    <h1>About Page</h1>
    <h1>{fetched.name}</h1>
    <h1>{fetched.age}</h1>
  </div>
);
```

Conclusion:

- **You must use `state` in the `Link` component to pass data between pages.**
- Any key-value pair inside `state={{}}` can be used.
- Avoid replacing `state` with other names like `myData` directly in the `Link` component as it won't work with `React Router`.

▼ useNavigation

Khushal Savalakha

💡 then how to use it with useNavigation ?

The `useNavigation` hook is part of the React Router Data APIs (like the `<Form>` component and loaders). It's typically used in applications where you're handling data submissions and responses, particularly for transitions between different pages (e.g., form submissions) rather than simple page navigations. For basic page navigation between routes using `Link` and `state`, `useNavigation` is not necessary.

That being said, if you want to use `useNavigation` for something similar, it works best with **data-fetching routers** like `<Form>` and `<Navigate>`, but it's not typically used in conjunction with simple navigation like you're doing with `Link` and `useLocation`. In your case, when you're passing data between routes using `Link`, `useNavigation` is overkill and not required.

Scenario: Passing Data with `Link` and Using `useNavigation`

If you really want to use `useNavigation` in your app, you need to switch to handling data using form submissions or other data-fetching methods like `loader` or `action` in React Router v6. Here's an outline of how that can be done:

1. **Use `Form` to Submit Data:**

   You can replace your `<Link>` with a `<Form>` element to simulate navigation that includes data submissions:

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

export default function Home() {
  return (
    <div>
      <h1>Home Page</h1>
      <Form action="/about" method="post">
        <input type="hidden" name="name" value="Joh
n" />
        <input type="hidden" name="age" value="25"
/>
        <button type="submit">Go to About</button>
      </Form>
    </div>
```


Khushal Savalakha

```
  );
}
```

2. **Access Data in About Page via `useNavigation`:**

   Now you can use `useNavigation` on the `About` page to get information about the navigation state:

```
import React from 'react';
import { useNavigation, useLoaderData } from 'react
-router-dom';

export default function About() {
  const navigation = useNavigation();
  const data = useLoaderData();

  return (
    <div>
      <h1>About Page</h1>
      <p>{`Navigation state: ${navigation.state}`}
</p>
      <p>{`Name: ${data.name}, Age: ${data.age}`}</
p>
    </div>
  );
}
```

   Here, you could track the `navigation.state` to show loading states or transitions during the form submission.

3. **Setting Up a Loader (for Fetching Data):**

   For the `useLoaderData` to work, you'd need to define a `loader` function in your route configuration that fetches the data:

```
import { createBrowserRouter, Route } from 'react-r
outer-dom';

const router = createBrowserRouter([
  {
```


Khushal Savalakha

```
    path: "/about",
    element: <About />,
    loader: async () => {
      return { name: "John", age: 25 };
    }
  }
]);
```

Important Notes:

- `useNavigation` is used for form submissions and tracking transitions or pending states during data transitions (like when a page is loading). It's not meant for general navigation between routes with `Link` and `useLocation`.
- If you just want to pass data between routes, `Link` **with** `state` **and** `useLocation` is the right approach.
- For tracking transitions or data-fetching states, use `useNavigation` along with `Form` or `Navigate`.

In your current setup, using `useLocation` and `Link` is sufficient, and you don't need to introduce `useNavigation` unless you're moving into forms and data loaders in React Router.

> 💡 then how it send's data while using navigate('/home',{{data:"helo"}}) ?

When you use `navigate('/home', { state: { data: "hello" } })` in React Router, it sends data through the `state` object (as part of the second argument). This is very similar to how you use the `state` prop with the `Link` component, and this data can be accessed on the target page via the `useLocation` hook.

How `navigate()` Works with `state`

The `navigate()` function allows you to programmatically navigate between routes in your app. You can pass an optional `state` object along with the path, which will be accessible on the destination route.

Here's how to use it:

1. **Sending data using** `navigate`

Khushal Savalakha

Suppose you have a button in the `Home` component that navigates to the `About` component and sends data along:

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

export default function Home() {
  const navigate = useNavigate();

  const handleClick = () => {
    navigate('/about', { state: { data: 'hello' }
});
  };

  return (
    <div>
      <h1>Home Page</h1>
      <button onClick={handleClick}>Go to About wit
h Data</button>
    </div>
  );
}
```

In the example above:

- `navigate('/about', { state: { data: "hello" } })` will send the `state` object to the `/about` route.
- `state` is an object that can hold any kind of data, like strings, numbers, objects, etc.

2. **Accessing the data in the `About` component**

On the `/about` route, you can retrieve the data using the `useLocation` hook.

```
import { useLocation, Link } from 'react-router-do
m';

export default function About() {
  const location = useLocation();
  const data = location.state?.data || 'No data rec
eived';
```


Khushal Savalakha

```
  return (
    <div>
      <h1>About Page</h1>
      <p>Data from Home: {data}</p>
      <Link to="/">Go back to Home</Link>
    </div>
  );
}
```

Here, `useLocation()` retrieves the `state` object sent by `navigate()`. The `location.state?.data` contains the `data` that was passed when navigating.

Example Breakdown:

- `navigate('/about', { state: { data: "hello" } })`:
  - The `navigate` function is called with two arguments:
    1. The **URL** to navigate to (`'/about'`).
    2. An optional **state** object (`{ state: { data: "hello" } }`).
  - This sends `data: "hello"` to the `/about` page.
- **Accessing in the `About` component using `useLocation`**:
  - The `useLocation` hook returns the current location object, which includes the `state` object.
  - You access the `state` by `location.state`, where the `data` can be retrieved.

Full Working Example:

`App.jsx`

```
import React from 'react';
import { BrowserRouter, Routes, Route, Link } from 're
act-router-dom';
import Home from './Home';
import About from './About';

export default function App() {
  return (
```


Khushal Savalakha

```
    <BrowserRouter>
      <nav>
        <Link to="/">Home</Link>
        <Link to="/about">About</Link>
      </nav>
      <Routes>
        <Route path="/" element={<Home />} />
        <Route path="/about" element={<About />} />
      </Routes>
    </BrowserRouter>
  );
}
```

Home.jsx

```
import React from 'react';
import { useNavigate } from 'react-router-dom';

export default function Home() {
  const navigate = useNavigate();

  const handleClick = () => {
    navigate('/about', { state: { data: 'hello from Ho
me' } });
  };

  return (
    <div>
      <h1>Home Page</h1>
      <button onClick={handleClick}>Go to About with D
ata</button>
    </div>
  );
}
```

About.jsx

Khushal Savalakhia

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

export default function About() {
  const location = useLocation();
  const data = location.state?.data || 'No data receiv
ed';

  return (
    <div>
      <h1>About Page</h1>
      <p>Data from Home: {data}</p>
      <Link to="/">Go back to Home</Link>
    </div>
  );
}
```

Key Points:

- `navigate(path, { state })`: This allows you to navigate programmatically and pass data in the `state` object.
- `useLocation`: Retrieves the location object, which includes the `state` passed during navigation. This is where you access the data sent via `navigate`.

Let me know if you need further clarification or modifications!

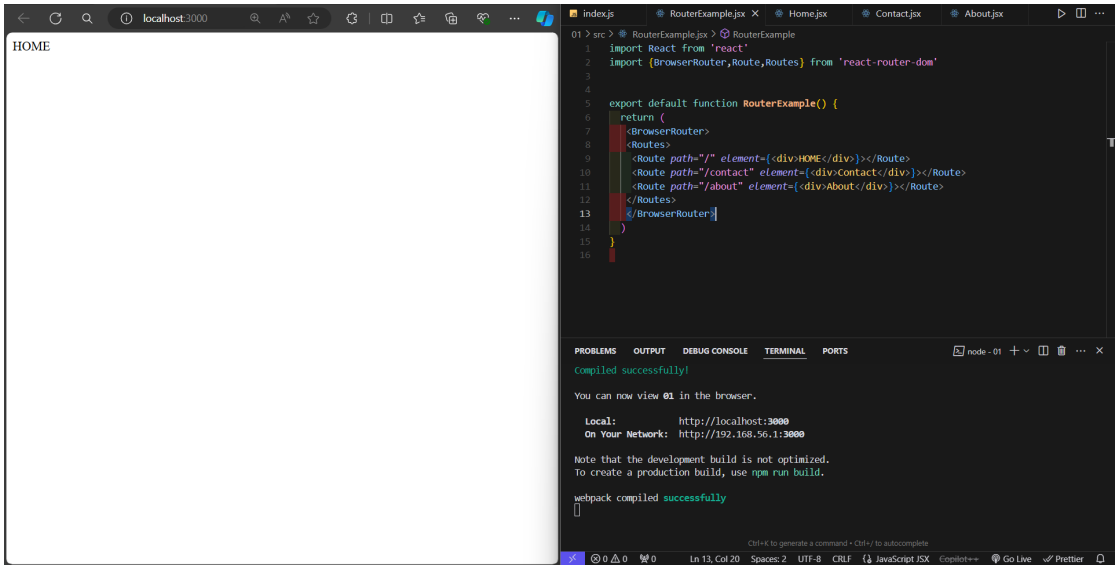
▼ React Router

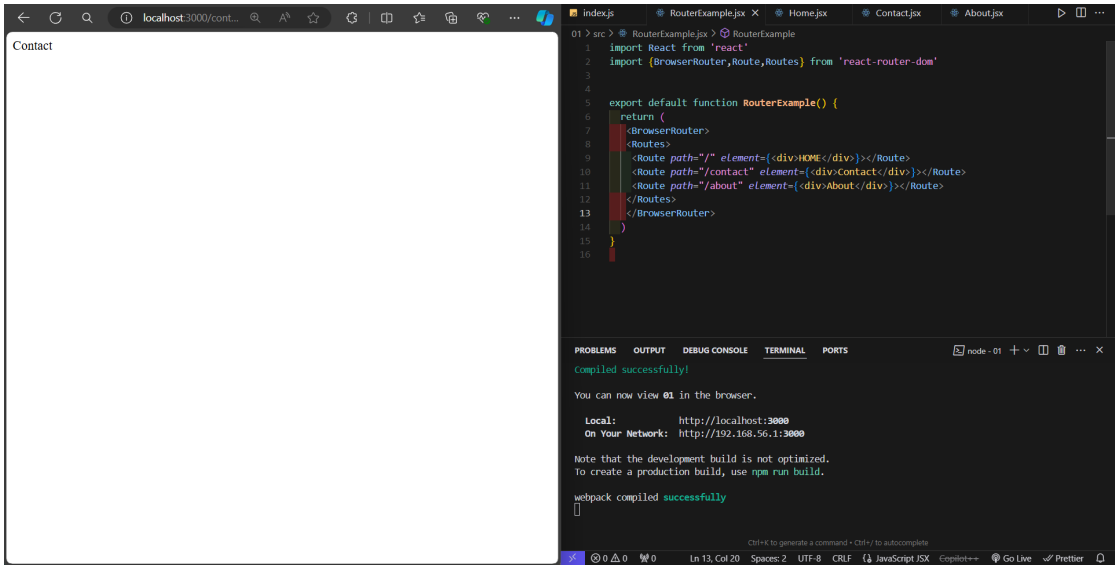
▼ ***REACT ROUTER NAVBAR***

```
import React from 'react'
import {BrowserRouter,Route,Routes} from 'react-router-d


export default function RouterExample() {
  return (
   <BrowserRouter>
```

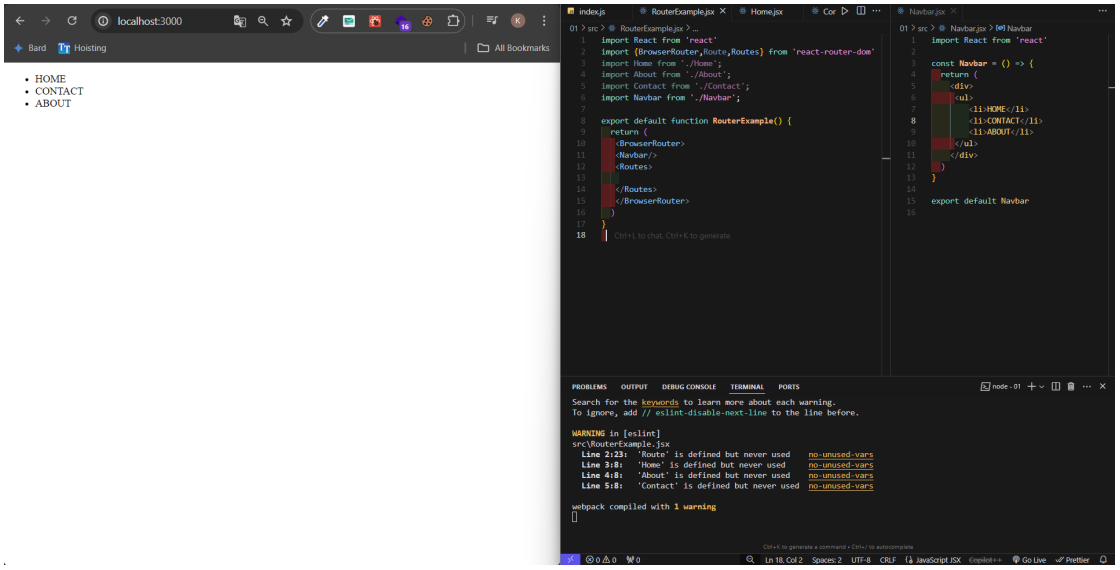

Khushal Savalakha

```
    <Routes>
     <Route path="/" element={<div>HOME</div>}></Route>//
     <Route path="/contact" element={<div>Contact</div>}>
     <Route path="/about" element={<div>About</div>}></Ro
    </Routes>
    </BrowserRouter>
   )
}
```





Khushal Savalakha



RouterExample.js

```
import React from 'react'
import {BrowserRouter,Route,Routes} from 'react-router-d
import Home from './Home';
import About from './About';
import Contact from './Contact';
import Navbar from './Navbar';

export default function RouterExample() {
  return (
   <BrowserRouter>
   <Navbar/>
   <Routes>
    <Route path='/' element={<Home/>}></Route>
    <Route path='/about' element={<About/>}></Route>
    <Route path='/Contact' element={<Contact/>}></Route>
    <Route path='/login' element={<div style={{textAlign
   </Routes>
   </BrowserRouter>
  )
}
```

Navbar.js

Khushal Savalakha

```
import React from 'react';
import { Link } from 'react-router-dom';
import './App.css';

const Navbar = () => {
  return (
    <nav className="navbar">
      <ul>
        <li><Link to="/">LOGO</Link></li>
        <li><Link to="/about">ABOUT</Link></li>
        <li><Link to="/contact">CONTACT</Link></li>
        <li><Link to="/login"><button>LOGIN</button></Li
      </ul>
    </nav>
  );
}

export default Navbar;
```

App.css

```
body {
    margin: 0;
    font-family: Arial, sans-serif;
  }

  .navbar {
    background-color: #333;
    padding: 0 20px;
  }

  .navbar ul {
    display: flex;
    align-items: center;
    justify-content: space-between;
    list-style: none;
    margin: 0;
    padding: 0;
```


Khushal Savalakha

```
  height: 60px;
}

.navbar li {
  margin: 0 10px;
}

.navbar a, .navbar button {
  color: white;
  text-decoration: none;
  padding: 10px 15px;
  font-size: 16px;
  transition: background-color 0.3s, color 0.3s;
}

.navbar a:hover, .navbar button:hover {
  background-color: #575757;
  border-radius: 4px;
}

.navbar button {
  background-color: #007BFF;
  border: none;
  border-radius: 4px;
  cursor: pointer;
}

.navbar button:hover {
  background-color: #0056b3;
}
```

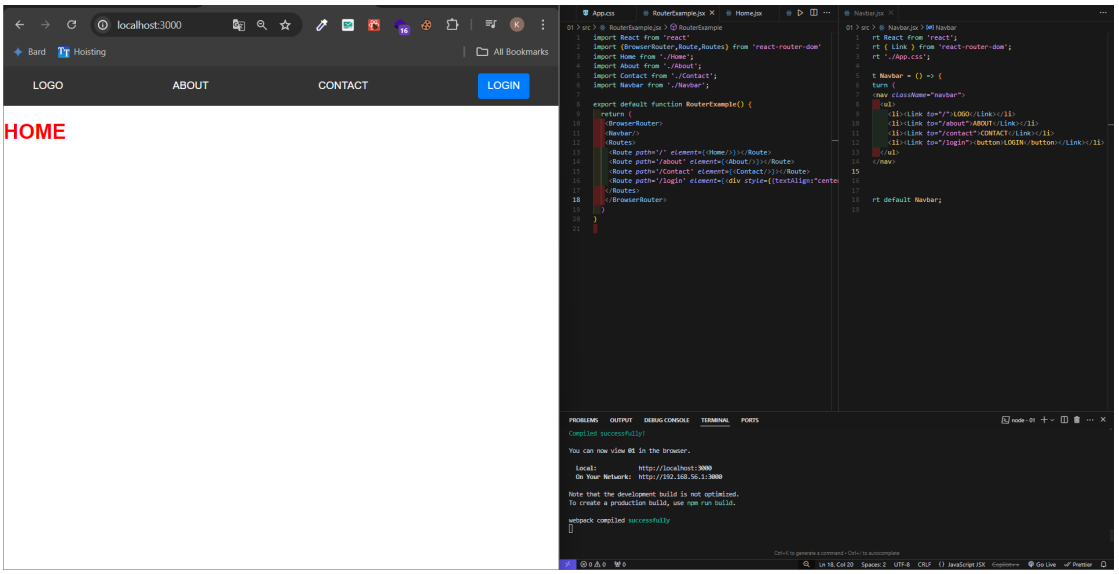

Khushal Savalakhia

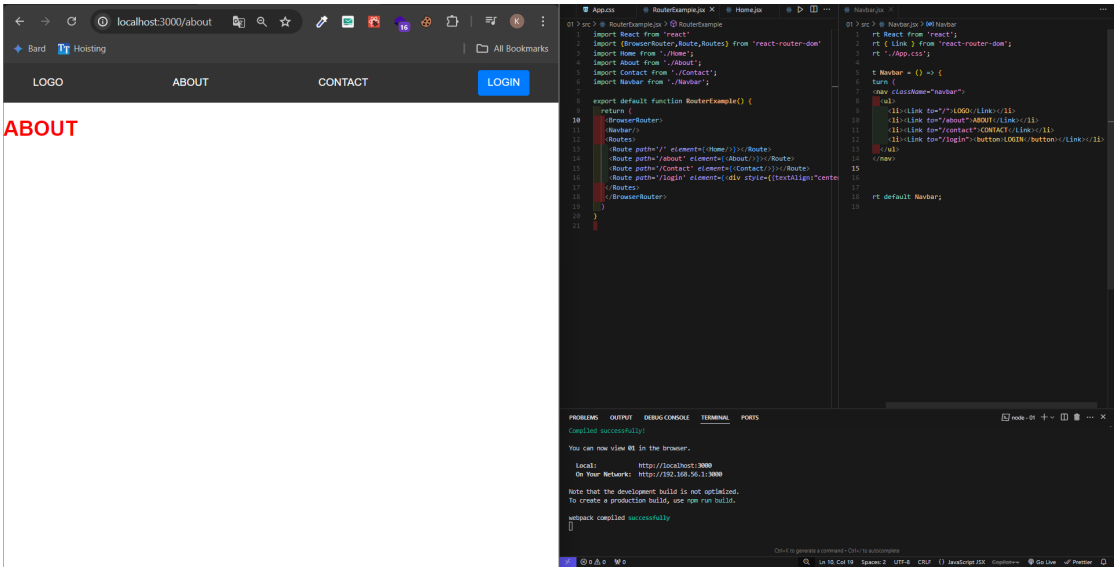
```jsx
import React from 'react'

export default function Home() {
  return (
    <div>
    <h1 style={{color:'red'}}>HOME</h1>
    </div>
  )
}
```

```jsx
import React from 'react'

export default function Contact() {
  return (
    <div>
    <h1 style={{color:'red'}}>CONTACT</h1>
    </div>
  )
}
```

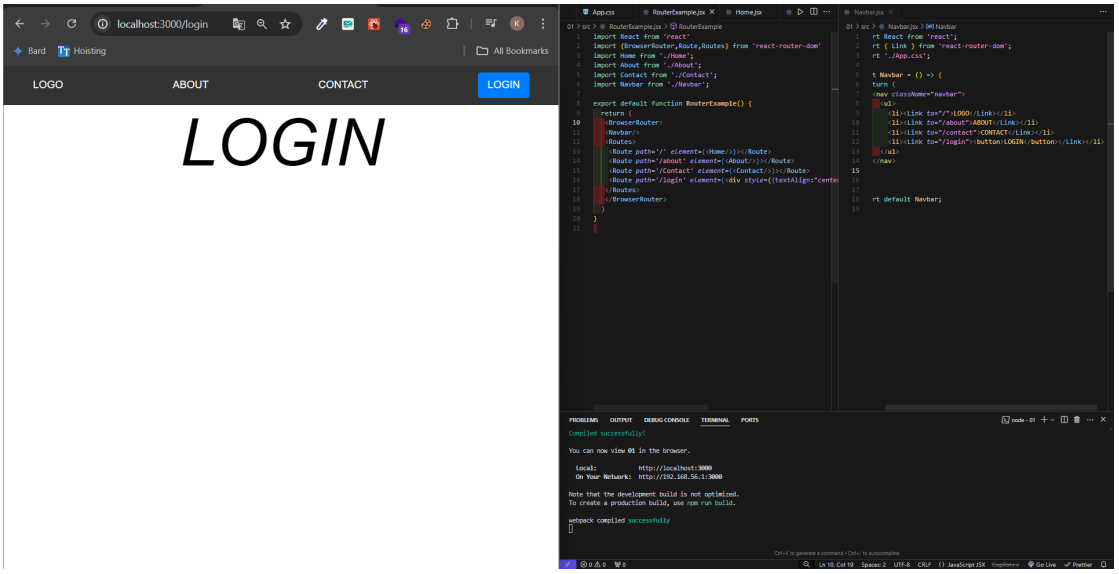
```jsx
import React from 'react'

export default function About() {
  return (
    <div>
    <h1 style={{color:'red'}}>ABOUT</h1>
    </div>
  )
}
```

```
Compiled successfully!

You can now view 01 in the browser.

  Local:            http://localhost:3000
  On Your Network:  http://192.168.56.1:3000

Note that the development build is not optimized.
To create a production build, use npm run build.

webpack compiled successfully
```



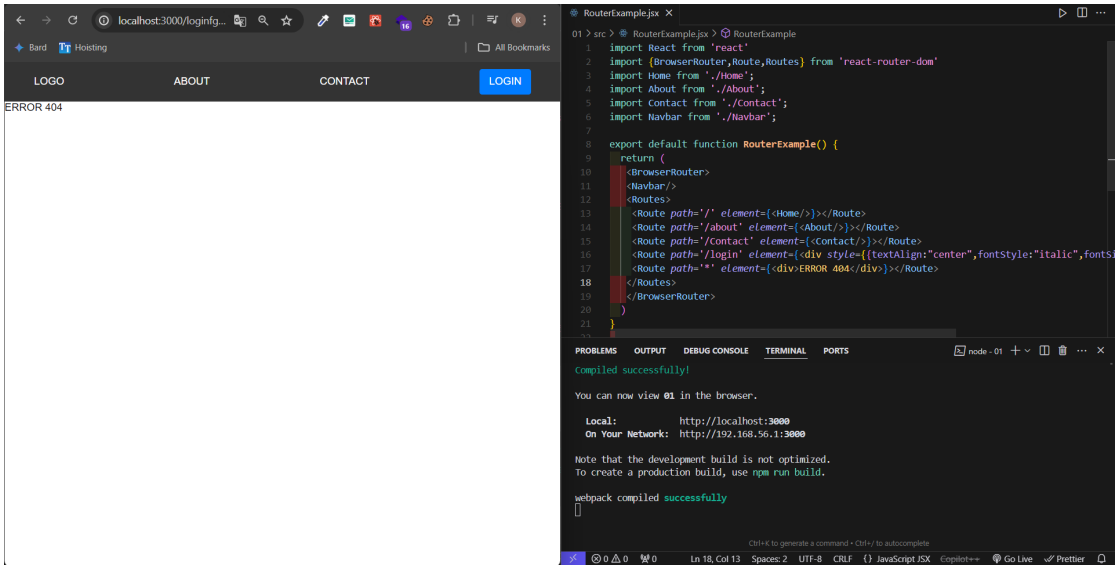


Khushal Savalakha




```
import React from 'react'
import {BrowserRouter,Route,Routes} from 'react-router-d
import Home from './Home';
import About from './About';
import Contact from './Contact';
import Navbar from './Navbar';

export default function RouterExample() {
  return (
   <BrowserRouter>
   <Navbar/>
   <Routes>
    <Route path='/' element={<Home/>}></Route>
    <Route path='/about' element={<About/>}></Route>
    <Route path='/Contact' element={<Contact/>}></Route>
    <Route path='/login' element={<div style={{textAlign
    <Route path='*' element={<div>ERROR 404</div>}></Rou
   </Routes>
   </BrowserRouter>
  )
}
```


Khushal Savalakha



The provided code is a React component that sets up client-side routing using `react-router-dom`. This allows different components to be rendered based on the URL path. Here's a breakdown of how the code works:

Explanation

1. **Imports**:
   - `React`: The core library for building user interfaces with React.
   - `{BrowserRouter, Route, Routes}`: Components from `react-router-dom` used for setting up routing in a React application.
   - `Home`, `About`, `Contact`, `Navbar`: Components imported from their respective files.
2. **Router Example Component**:
   - This component sets up the routing for the application. It is exported as the default export of the module.
3. **BrowserRouter**:
   - The `BrowserRouter` component wraps the entire application, enabling the use of the HTML5 history API for navigation. This is required for routing to work.
4. **Navbar**:
   - The `Navbar` component is included inside the `BrowserRouter`, which means it will be displayed on every page since it is outside of the `Routes` component.

Khushal Savalakha

5. **Routes**:
   - The `Routes` component acts as a container for all the `Route` components. It ensures that only one route is rendered at a time.
6. **Route**:
   - `<Route path='/' element={<Home/>}>`: This route renders the `Home` component when the URL path is `/`.
   - `<Route path='/about' element={<About/>}>`: This route renders the `About` component when the URL path is `/about`.
   - `<Route path='/contact' element={<Contact/>}>`: This route renders the `Contact` component when the URL path is `/contact`.
   - `<Route path='/login' element={<div style={{textAlign:"center",fontStyle:"italic",fontSize:"100px"}}>LOGIN</div>}>`: This route renders an inline `div` with styled text "LOGIN" when the URL path is `/login`.
   - `<Route path='*' element={<div>ERROR 404</div>}>`: This route catches any unspecified paths and renders an inline `div` with the text "ERROR 404". This acts as a catch-all for undefined routes.

How It Works

1. **Application Start**:
   - When the application starts, the `RouterExample` component is rendered.
   - `BrowserRouter` is the top-level component, enabling routing capabilities.
2. **Rendering the Navbar**:
   - The `Navbar` component is rendered, providing navigation links to the different routes (e.g., Home, About, Contact, Login).
3. **Matching Routes**:
   - When the URL path changes, the `Routes` component looks for a matching `Route`.
   - If the path is `/`, the `Home` component is rendered.
   - If the path is `/about`, the `About` component is rendered.
   - If the path is `/contact`, the `Contact` component is rendered.

Khushal Savalakha

- If the path is `/login`, an inline `div` with styled "LOGIN" text is rendered.
- If the path does not match any of the specified routes, the catch-all route renders an inline `div` with "ERROR 404".

Navigation Example

- **Home**: Visiting `http://localhost:3000/` will render the `Home` component.
- **About**: Visiting `http://localhost:3000/about` will render the `About` component.
- **Contact**: Visiting `http://localhost:3000/contact` will render the `Contact` component.
- **Login**: Visiting `http://localhost:3000/login` will render the styled "LOGIN" text.
- **404**: Visiting `http://localhost:3000/some-invalid-path` will render "ERROR 404".

This setup ensures that your application has a navigable structure with defined routes, providing a smooth and user-friendly navigation experience.

▼ ***FORM VALIDATION APP***

FormValidationApp.js

```
import React, { useState } from "react";
import "./form.css";

function FormValidationApp() {
  const [name, setName] = useState("");
  const [email, setEmail] = useState("");
  const [password, setPassword] = useState("");
  const [confirmPassword, setConfirmPassword] = useState
  const [age, setAge] = useState("");
  const [gender, setGender] = useState("");
  const [error, setError] = useState("");

  const handleSubmit = (e) => {
    e.preventDefault();
    const validationErrors = validateForm();
    if (Object.keys(validationErrors).length > 0) {
```


Khushal Savalakhia

```
      setError(validationErrors);
    } else {
      console.log("Form submitted.");
    }
    // if (password !== confirmPassword) {
    //   setError("Passwords do not match");
    //   return;
    // }
    // Add form submission logic here
    // console.log("Form submitted", { name, email, pass
    // setError("");
  };

  function validateForm() {
    const errors = {};
    if (!name) {
      errors.name = "Name is Required";
    }
    if (!email) {
      errors.email = "Email is Required";
    } else if (!/\S+@\S+\.\S+/.test(email)) {
      errors.email = "Email is invalid";
    }
    if (!password) {
      errors.password = "Password is Required";
    } else if (password.length) {
      errors.password = "Password must be 6 characters."
    }
    if (!confirmPassword) {
      errors.confirmPassword = "Confirm Password is Requ
    } else if (password !== confirmPassword) {
      errors.confirmPassword = "Confirm Password must be
    }
    if (!age) {
      errors.age = "Age is required";
    } else if (isNaN(age) || age < 18) {
      errors.age = "Age must be number and atleast 18.";
    }
```


Khushal Savalakha

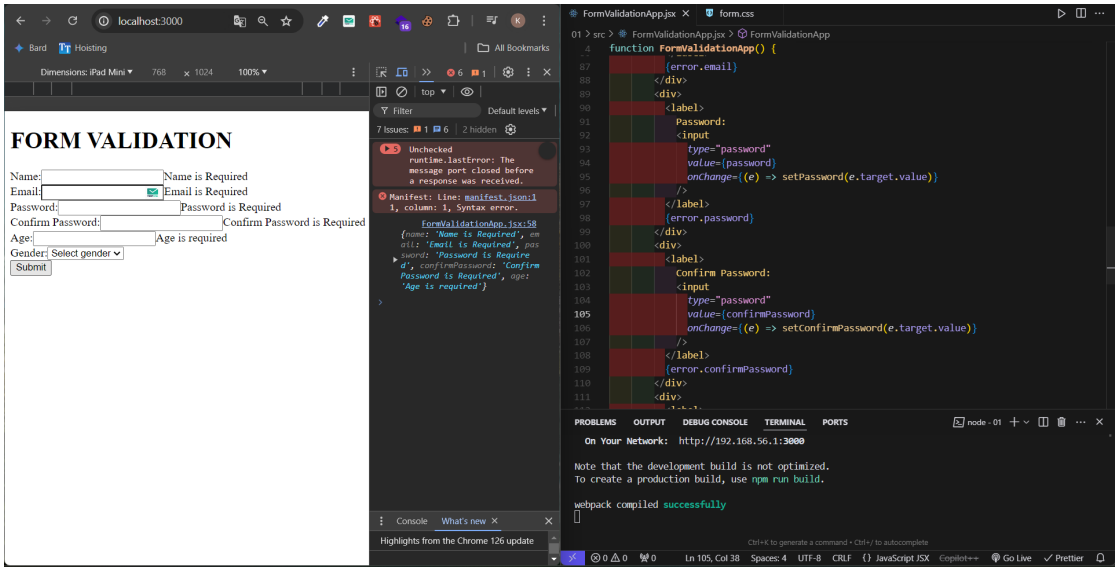
```
    if (gender) {
      errors.gender = "Gender is required..";
    }
    console.log(errors);
    return errors;
  }
  return (
    <div className="container">
      <h1>FORM VALIDATION</h1>
      <form onSubmit={handleSubmit}>
        <div>
          <label>
            Name:
            <input
              type="text"
              value={name}
              onChange={(e) => {
                setName(e.target.value);
              }}
            />
          </label>
          {error.name}
        </div>
        <div>
          <label>
            Email:
            <input
              type="email"
              value={email}
              onChange={(e) => setEmail(e.target.value)}
            />
          </label>
          {error.email}
        </div>
        <div>
          <label>
            Password:
            <input
```

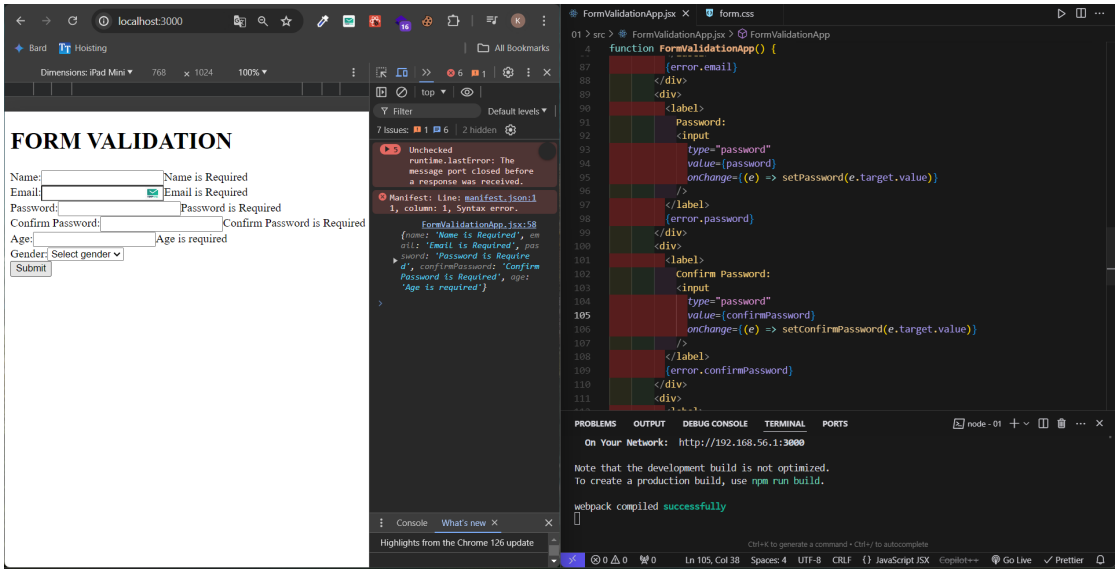

Khushal Savalakha

```
                type="password"
                value={password}
                onChange={(e) => setPassword(e.target.valu
              />
            </label>
            {error.password}
          </div>
          <div>
            <label>
              Confirm Password:
              <input
                type="password"
                value={confirmPassword}
                onChange={(e) => setConfirmPassword(e.targ
              />
            </label>
            {error.confirmPassword}
          </div>
          <div>
            <label>
              Age:
              <input
                type="text"
                value={age}
                onChange={(e) => setAge(e.target.value)}
              />
            </label>
            {error.age}
          </div>
          <div>
            <label>
              Gender:
              <select value={gender} onChange={(e) => setG
                <option value="" disabled>
                  Select gender
                </option>
                <option value="Male">Male</option>
                <option value="Female">Female</option>
```


Khushal Savalakha

```
                <option value="Other">Other</option>
              </select>
            </label>
            {error.gender}
          </div>
          <button type="submit">Submit</button>
        </form>
      </div>
    );
  }

export default FormValidationApp;
```





Khushal Savalakha

The provided code is a React component that renders a form with validation. Here's a simple and short explanation of the code:

Key Parts:

1. **State Management**:
   - The `useState` hook is used to manage the state of form fields (`name`, `email`, `password`, `confirmPassword`, `age`, `gender`) and errors (`error`).
2. **Form Submission**:
   - The `handleSubmit` function is called when the form is submitted.
   - It prevents the default form submission behavior using `e.preventDefault()`.
   - It calls `validateForm` to check for validation errors.
   - If there are errors, they are set in the `error` state. Otherwise, it logs "Form submitted."
3. **Validation Logic**:
   - The `validateForm` function checks each field for required conditions and proper formats:
     - Name and email are required.
     - Email must be in a valid format.
     - Password must be at least 6 characters long.
     - Confirm Password must match Password.
     - Age must be a number and at least 18.
     - Gender is required.
   - Errors are stored in an `errors` object and returned.
4. **Rendering**:
   - The form fields are rendered with their current values.
   - Each field has an `onChange` handler to update its state.
   - If there are errors, they are displayed below the respective fields.

Summary:

Khushal Savalakha

- The form collects user input for name, email, password, confirm password, age, and gender.
- The form validates the input before submission and displays appropriate error messages if there are issues.
- On successful validation, the form data is logged to the console.

▼ ***FORM VALIDATION APP2 using react library***

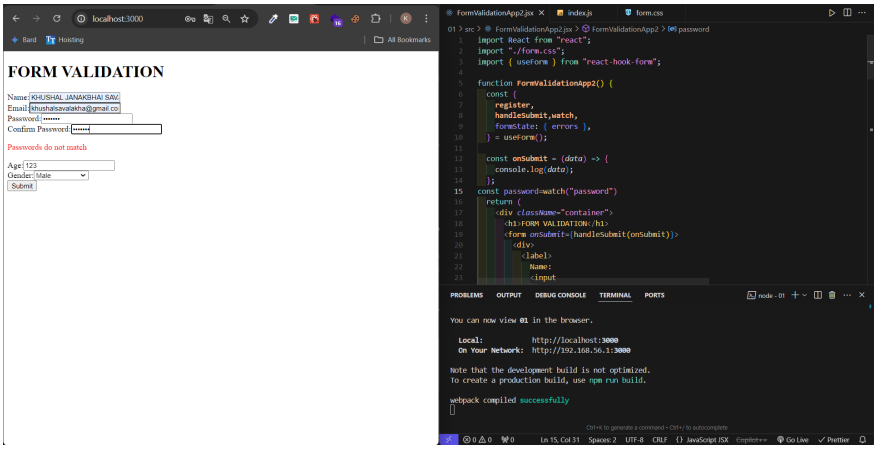
```
import React from "react";
import "./form.css";
import { useForm } from "react-hook-form";

function FormValidationApp2() {
  const {
    register,
    handleSubmit,
    formState: { errors },
  } = useForm();

  const onSubmit = (data) => {
    console.log(data);
  };

  return (
    <div className="container">
      <h1>FORM VALIDATION</h1>
      <form onSubmit={handleSubmit(onSubmit)}>
        <div>
          <label>
            Name:
            <input type="text" {...register("name", { re
          </label>
          {errors.name && <p style={{ color: "red" }}>{e
        </div>
        <div>
          <label>
            Email:
            <input type="email" {...register("email", {
```


Khushal Savalakha

```
            </label>
            {errors.email && <p style={{ color: "red" }}>{
          </div>
          <div>
            <label>
              Password:
              <input type="password" {...register("passwor
            </label>
            {errors.password && <p style={{ color: "red" }
          </div>
          <div>
            <label>
              Confirm Password:
              <input type="password" {...register("confirm
            </label>
            {errors.confirmPassword && <p style={{ color:
          </div>
          <div>
            <label>
              Age:
              <input type="text" {...register("age", { req
            </label>
            {errors.age && <p style={{ color: "red" }}>{er
          </div>
          <div>
            <label>
              Gender:
              <select {...register("gender", { required: "
                <option value="" disabled>Select gender</o
                <option value="Male">Male</option>
                <option value="Female">Female</option>
                <option value="Other">Other</option>
              </select>
            </label>
            {errors.gender && <p style={{ color: "red" }}>
          </div>
          <button type="submit">Submit</button>
        </form>
```


Khushal Savalakha

```
    </div>
  );
}

export default FormValidationApp2;
```



💡 but here why we have to use const password = watch("password");?

In `react-hook-form`, the `watch` function is used to keep track of the current value of form fields. When implementing password confirmation functionality, you need to ensure that the value entered in the "Confirm Password" field matches the value in the "Password" field. To do this, you need access to the current value of the "Password" field within the validation logic of the "Confirm Password" field.

The `watch` function allows you to achieve this by observing the value of the "Password" field as it changes. This value is then used in the `validate` function to compare it with the value entered in the "Confirm Password" field.

Here's a step-by-step breakdown of why and how it's used:

1. **Need for Current Value**:
    - You need the current value of the "Password" field to compare it with the "Confirm Password" field.
    - Without `watch`, you wouldn't be able to dynamically access the "Password" field's value during validation.
2. **Using `watch`**:

Khushal Savalakha

- The `watch` function is called with the name of the field you want to observe (`"password"` in this case).
- `watch("password")` returns the current value of the "Password" field.

3. **Validation Logic**:
   - Within the `validate` function of the `confirmPassword` field, you use the value obtained from `watch("password")` to check if it matches the value of `confirmPassword`.
   - This ensures that the validation logic dynamically checks the current value of the "Password" field.

Here's a simple example to illustrate this:

```
import React from "react";
import "./form.css";
import { useForm } from "react-hook-form";

function FormValidationApp2() {
  const {
    register,
    handleSubmit,
    formState: { errors },
    watch,
  } = useForm();

  const onSubmit = (data) => {
    console.log(data);
  };

  const password = watch("password");

  return (
    <div className="container">
      <h1>FORM VALIDATION</h1>
      <form onSubmit={handleSubmit(onSubmit)}>
        <div>
          <label>
            Password:
```


Khushal Savalakha

```
            <input
              type="password"
              {...register("password", {
                required: "Password is Required",
                minLength: {
                  value: 6,
                  message: "Password must be at least
6 characters",
                },
              })}
            />
          </label>
          {errors.password && (
            <p style={{ color: "red" }}>{errors.passwo
rd.message}</p>
          )}
        </div>
        <div>
          <label>
            Confirm Password:
            <input
              type="password"
              {...register("confirmPassword", {
                required: "Confirm Password is Require
d",
                validate: (value) =>
                  value === password || "Passwords do
not match",
              })}
            />
          </label>
          {errors.confirmPassword && (
            <p style={{ color: "red" }}>{errors.confir
mPassword.message}</p>
          )}
        </div>
        <button type="submit">Submit</button>
      </form>
```


Khushal Savalakha

```
    </div>
  );
}

export default FormValidationApp2;
```

In summary, `watch("password")` is used to dynamically get the current value of the "Password" field so that the "Confirm Password" field can validate against it, ensuring both fields contain the same value before allowing form submission.

> 💡 explain here how we validate password and confirm password and how to print error message using { && };

In the provided React component `FormValidationApp2`, password and confirm password validation is implemented using the `react-hook-form` library. Here's how the validation and error message display work:

1. **Validation Rules Setup:**
   - For the `password` field:

     ```
     <input
       type="password"
       {...register("password", { required: "Password is Required" })}
     />
     ```

     This setup uses the `required` rule to ensure that the password field cannot be empty.

   - For the `confirmPassword` field:

     ```
     <input
       type="password"
       {...register("confirmPassword", {
         required: "Confirm Password is Required",
         validate: (value) => value === password || "Passwords do not match",
     ```


Khushal Savalakha

```
  })}
/>
```

Here, `validate` is used to define a custom validation rule. It checks if the value of the `confirmPassword` field matches the value entered in the `password` field (`password === value`). If they do not match, the validation fails, and the error message `"Passwords do not match"` is displayed.

2. **Error Message Display:**

   - Each input field has an associated error message display condition:

```
{errors.password && (
  <p style={{ color: "red" }}>{errors.password.m
essage}</p>
)}
```

This condition checks if there is an error (`errors.password`) for the `password` field. If an error exists, it displays the error message (`errors.password.message`) in red.

Similarly, for the `confirmPassword` field:

```
{errors.confirmPassword && (
  <p style={{ color: "red" }}>{errors.confirmPas
sword.message}</p>
)}
```

This condition checks for errors in the `confirmPassword` field and displays the associated error message if there is one.

3. **Submission and Validation Trigger:**

   - Validation is triggered automatically when the form is submitted (`handleSubmit(onSubmit)`). Errors are collected and displayed based on the rules defined for each field.

In summary, the `react-hook-form` library allows us to define validation rules directly in the `register` method's configuration object. Error messages are displayed conditionally using `{}` in JSX, ensuring that users receive feedback when validation criteria are not met.

Khushal Savalakha

> Depth

Certainly! Let's break down how the `FormValidationApp2` React component works, step by step:

Setup and Imports

```
import React from "react";
import "./form.css";
import { useForm } from "react-hook-form";
```

- **Imports**: The component imports necessary modules:
  - `React` for creating React components.
  - `"./form.css"` for styling (assuming it contains custom styles).
  - `{ useForm }` from `react-hook-form` for managing form state and validation.

FormValidationApp2 Component

```
function FormValidationApp2() {
  const {
    register,
    handleSubmit,
    watch,
    formState: { errors },
  } = useForm();
```

- **useForm Hook**:
  - `useForm()` initializes the form state and provides methods (`register`, `handleSubmit`, `watch`) and properties (`errors`) to manage the form.
  - `register` is used to register input fields with validation rules.
  - `handleSubmit` handles form submission with validation.
  - `watch` allows watching input values, useful for password confirmation.
  - `errors` holds validation error messages.

Form Submission Handling

Khushal Savalakha

```
  const onSubmit = (data) => {
    console.log(data);
  };
```

- **onSubmit Function**:
  - When the form is submitted (`handleSubmit(onSubmit)`), `onSubmit` function is called.
  - `data` parameter contains the form data submitted by the user.
  - Here, it logs the form data to the console; in a real application, you would typically send this data to a server or perform further actions.

JSX Structure

```
  return (
    <div className="container">
      <h1>FORM VALIDATION</h1>
      <form onSubmit={handleSubmit(onSubmit)}>
        {/* Input fields for Name, Email, Password, et
c. */}
      </form>
    </div>
  );
}
```

- **JSX Structure**:
  - A `<div>` with `className="container"` wraps the entire form.
  - `<h1>` displays the title "FORM VALIDATION".
  - `<form>` element wraps input fields and handles form submission via `handleSubmit(onSubmit)`.

Input Fields with Validation

```
        <div>
          <label>
            Name:
```


Khushal Savalakha

```
            <input
              type="text"
              {...register("name", { required: "Name i
s Required" })}
            />
          </label>
          {errors.name && <p style={{ color: "red" }}>
{errors.name.message}</p>}
        </div>
```

- **Input Field Example**:
  - Each input field (`<input>`) is associated with `register` to link it to the form state and validation rules.
  - Example with `"name"`:
    - `type="text"` specifies input type.
    - `{...register("name", { required: "Name is Required" })}` registers "name" field with a `required` rule and error message.
    - `{errors.name && <p style={{ color: "red" }}>{errors.name.message}</p>}` displays error message in red if validation fails.

Validation Rules

- **Validation Rules**:
  - Various rules (`required`, `pattern`, `validate`) are defined using object literals within `register`.
  - Error messages (`errors.name.message`) are shown conditionally based on validation results (`errors.name`).

Select Field

```
        <div>
          <label>
            Gender:
            <select {...register("gender", { required:
"Gender is Required" })}>
              <option value="" disabled>
```


Khushal Savalakha

```
                  Select gender
                </option>
                <option value="Male">Male</option>
                <option value="Female">Female</option>
                <option value="Other">Other</option>
              </select>
            </label>
            {errors.gender && (
              <p style={{ color: "red" }}>{errors.gende
r.message}</p>
            )}
          </div>
```

- **Select Field**:
  - `<select>` element with options for selecting gender.
  - `{...register("gender", { required: "Gender is Required" })}` registers "gender" field with `required` rule.
  - Error message displayed if validation fails (`{errors.gender && ...}`).

Submit Button

```
          <button type="submit">Submit</button>
```

- **Submit Button**:
  - `<button>` triggers form submission when clicked.

Overall Functionality

- **Validation**: Validates fields on form submission (`handleSubmit`), displaying error messages (`errors`).
- **Error Display**: Errors are shown inline with fields (`{errors.fieldName && ...}`).
- **Form Submission**: Data is logged to console in `onSubmit` function.

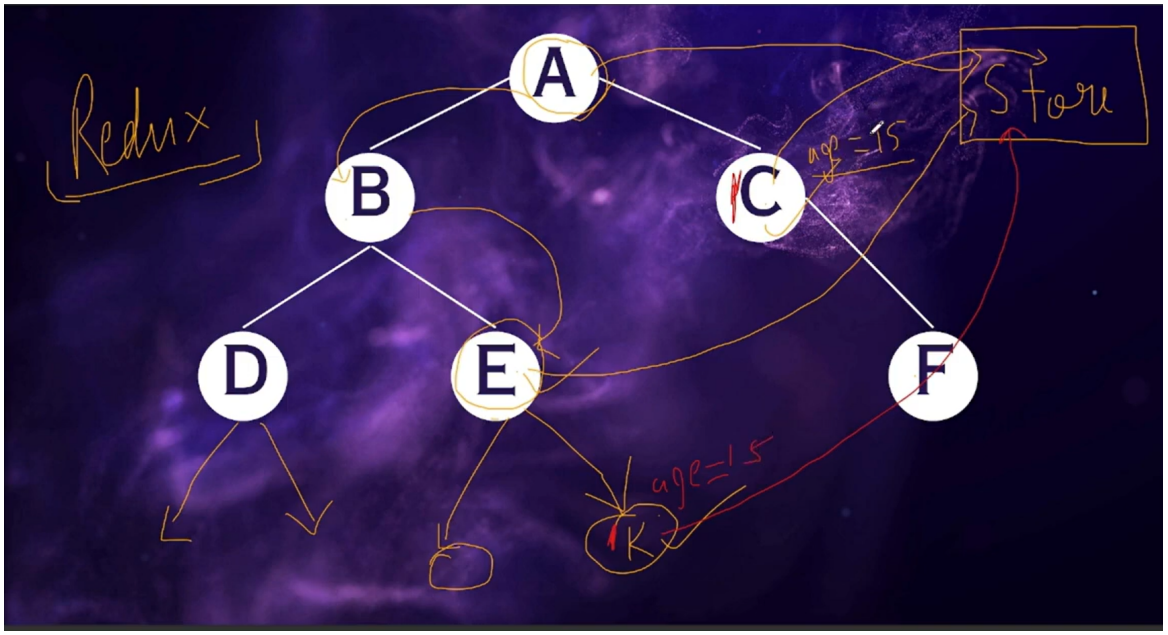
Conclusion

This component leverages `react-hook-form` to efficiently manage form state and validation in React. It provides a structured approach to validating user input and

Khushal Savalakha

displaying meaningful error messages, ensuring a robust user experience when filling out and submitting forms.

▼ REDUX



- Create a Redux Store
- Provide the Redux Store to React Main App.js
- Create a Redux State Slice
- Add Slice Reducers to the Store
- Use Redux State and Actions in React Component
- To use redux state in other components use the useSelector hook from react-redux.
- First Step is to configure our Store, now this is where you'll see the magic of redux toolkit! Now your redux store src/app/store.js will contains.

▼ Theory with example of (Incr,Decr)

https://redux-toolkit.js.org/tutorials/quick-start

Not quite. The term "Higher-Order Component" (HOC) in React has a specific meaning. Let's clarify it:

Higher-Order Component (HOC)

Khushal Savalakha

A Higher-Order Component is a function that takes a component and returns a new component. It's a pattern used in React to reuse component logic. An HOC does not refer to the component hierarchy (i.e., parent-child relationships) but to a function that wraps a component to add additional behavior or data.

Example of a Higher-Order Component:

```
function withExtraProps(WrappedComponent) {
  return function EnhancedComponent(props) {
    return <WrappedComponent extraProp="I am extra"
{...props} />;
  };
}

// Usage
const EnhancedApp = withExtraProps(App);
```

`Provider` in Redux

In your code, `Provider` is a component provided by `react-redux` that makes the Redux store available to any nested components that need to access the Redux store. While it wraps your `App` component, it is not considered a Higher-Order Component (HOC) in the traditional React sense. It is a regular component that uses the React Context API to pass the Redux store down the component tree.

Your Code Example

Here is your code again for reference:

```
import React from 'react';
import ReactDOM from 'react-dom/client';
import App from './App';
import { store } from './app/store';
import { Provider } from 'react-redux';

const root = ReactDOM.createRoot(document.getElementBy
Id('root'));
root.render(
  <Provider store={store}>
    <App />
```


Khushal Savalakha

```
  </Provider>
);

// If you want to start measuring performance in your
app, pass a function
// to log results (for example: reportWebVitals(consol
e.log))
// or send to an analytics endpoint. Learn more: <http
s://bit.ly/CRA-vitals>
reportWebVitals();
```

In this code, `Provider` is not an HOC. It is a component that wraps your `App` to provide it with the Redux store.

Summary

- **HOC (Higher-Order Component)**: A function that takes a component and returns a new component.
- **Provider**: A component from `react-redux` that uses the Context API to provide the Redux store to your app.

So, while `Provider` does wrap your `App` component, it is not a Higher-Order Component by definition.

> 💡 it means provider is connecter of store ?

Yes, you can think of the `Provider` component from `react-redux` as a connector that makes the Redux store available to the rest of your React application. It uses the Context API internally to pass the Redux store down through the component tree, so any component within the `Provider` can access the store via hooks like `useSelector` and `useDispatch` or the `connect` function.

Summary of Key Concepts

1. **Provider**:
    - Connects the Redux store to the React application.
    - Wraps the root component (or any component) to pass the store down the component tree.

Khushal Savalakha

- Allows child components to access the store using `useSelector`, `useDispatch`, or `connect`.

2. **useSelector and useDispatch**:
   - `useSelector`: A hook to extract data from the Redux store state.
   - `useDispatch`: A hook to dispatch actions to the Redux store.
3. **Higher-Order Component (HOC)**:
   - A function that takes a component and returns a new component with added props or behavior.

💡 Difference between useReducer and createSlice

Let's break down the differences between using `useReducer` in React and using `createSlice` in Redux Toolkit by comparing the two approaches side by side.

Comparison Table

Aspect	`useReducer` in React	`createSlice` in Redux Toolkit
State Initialization	State is initialized within the component using `useReducer` hook	State is initialized as part of the slice definition
Reducer Function	Defined inside the component or imported separately	Defined as part of the slice object
State Management	Managed locally within the component	Managed globally through the Redux store

Khushal Savalakhia

Actions	Defined manually within the reducer function	Automatically generated by `createSlice`
Usage	Typically used for local component state management	Used for global state management across the application
Boilerplate	Requires more boilerplate to set up actions and reducer	Less boilerplate as `createSlice` combines actions and reducer into one
Middleware Support	No built-in support for middleware like thunks	Full support for middleware through Redux
Debugging	Limited to component scope	Powerful debugging tools like Redux DevTools

Example Code Breakdown

`useReducer` in React

1. **State Initialization and Reducer Function**:

```
import React, { useReducer } from "react";

const reducer = (state, action) => {
    if (action.type === "incr") {
        return { count: state.count + 1 };
    } else if (action.type === "decr") {
        return { count: state.count - 1 };
    } else {
        throw new Error("Unsupported action typ
e");
    }
};
```

2. **Component with Local State Management**:

```
const App1 = () => {
    const [state, dispatch] = useReducer(reduce
r, { count: 0 });

    const handleInc = () => {
```


Khushal Savalakha

```
        dispatch({ type: "incr" });
    };

    const handleDec = () => {
        dispatch({ type: "decr" });
    };

    return (
        <div>
            <h1>{state.count}</h1>
            <button onClick={handleInc}>+</butto
n>
            <button onClick={handleDec}>-</butto
n>
        </div>
    );
};

export default App1;
```

`createSlice` in Redux Toolkit

1. **State Initialization and Slice Definition**:

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

const initialState = {
    value: 0
};

const counterSlice = createSlice({
    name: 'counter',
    initialState,
    reducers: {
        increment: (state) => {
            state.value += 1;
        },
        decrement: (state) => {
```


Khushal Savalakha

```
            state.value -= 1;
        }
    }
});

export const { increment, decrement } = counterS
lice.actions;
export default counterSlice.reducer;
```

2. **Component with Global State Management**:

```
import React from "react";
import { useSelector, useDispatch } from "react-
redux";
import { increment, decrement } from "./counterS
lice";

const Counter = () => {
    const count = useSelector((state) => state.c
ounter.value);
    const dispatch = useDispatch();

    return (
        <div>
            <h1>{count}</h1>
            <button onClick={() => dispatch(incr
ement())}>+</button>
            <button onClick={() => dispatch(decr
ement())}>-</button>
        </div>
    );
};

export default Counter;
```

Key Differences

- **State Initialization**:

Khushal Savalakha

- `useReducer`: State is initialized within the component (`useReducer(reducer, { count: 0 })`).
- `createSlice`: State is initialized within the slice definition (`initialState = { value: 0 }`).

- **Reducer Function**:
  - `useReducer`: The reducer function is defined separately and can be quite verbose with manual action handling.
  - `createSlice`: The reducer and actions are defined together within the slice, reducing boilerplate.
- **State Management**:
  - `useReducer`: Manages state locally within the component, useful for isolated state.
  - `createSlice`: Manages state globally through Redux, allowing for shared state across components.
- **Actions**:
  - `useReducer`: Actions are defined manually and dispatched directly.
  - `createSlice`: Actions are automatically generated and exported for use.
- **Boilerplate**:
  - `useReducer`: More boilerplate due to separate action type definitions and reducer function.
  - `createSlice`: Less boilerplate as actions and reducer logic are combined.
- **Middleware and Debugging**:
  - `useReducer`: No built-in support for middleware or advanced debugging tools.
  - `createSlice`: Full support for middleware and powerful debugging tools like Redux DevTools.

In summary, `useReducer` is suitable for managing complex local state within a single component, while `createSlice` in Redux Toolkit provides a more structured and efficient way to manage global state across your application with less boilerplate and more powerful features.

▼ Increment Decrement code

Khushal Savalakha

index.js

```
import React from 'react';
import ReactDOM from 'react-dom/client'; // Import
import { store } from './app/store'; // Import the
import { Provider } from 'react-redux'; // Import F
import Counter from './features/counter/Counter'; /

// Create a root element to render the React compor
const root = ReactDOM.createRoot(document.getElemer

// Render the app with Redux store provider
root.render(
  <Provider store={store}>
    <Counter />  {/* Render the Counter component */
  </Provider>
);
```

counterSlice.js

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

// Define the initial state for the counter
const initialState = {
    value: 0
};

// Create a slice for the counter with reducers
export const counterSlice = createSlice({
    name: 'counter', // Name of the slice
    initialState,
    reducers: {
        // Reducer to increment the counter
        increment: (state) => {
            state.value += 1;
        },
        // Reducer to decrement the counter
        decrement: (state) => {
```


Khushal Savalakhia

```
            state.value -= 1;
        },incrementByAmount: (state, action) => {
            state.value += action.payload
          },decrementByAmount: (state, action) => {
            state.value -= action.payload
          },
    }
});

// Export the actions generated by createSlice
export const { increment, decrement,incrementByAmou
// Export the reducer to be used in the store
export default counterSlice.reducer;
```

store.js

```
import { configureStore } from '@reduxjs/toolkit';
import counterReducer from '../features/counter/cou

// Create and configure the Redux store
export const store = configureStore({
    reducer: {
        counter: counterReducer, // Add the counter
    },
});
```

counter.js

```
import React from 'react';
import { useSelector, useDispatch } from 'react-red
import { decrement, increment, incrementByAmount,de

export default function Counter() {
    // Select the count value from the Redux store
    const count = useSelector((state) =>
        /* Here all states are in counter
        reducer: {
            counter: counterReducer,
```


Khushal Savalakha

```
        }, */
        state.counter.value // Access the 'value' p
    );

    // Get the dispatch function to dispatch actior
    const dispatch = useDispatch();
    console.log(useDispatch());
    return (
        <div>
            <div>
                <button
                    aria-label="Increment value" //
                    onClick={() => dispatch(increme
                >
                    Increment
                </button>
                <span>{count}</span> {/* Display th
                <button
                    aria-label="Decrement value" //
                    onClick={() => dispatch(decreme
                >
                    Decrement
                </button><br />
                <button
                    aria-label="Increment value by
                    onClick={() => dispatch(increme
                >
                    Increment value by 100
                </button><br />
                <button
                    aria-label="Decrement value by
                    onClick={() => dispatch(decreme
                >
                    Decrement value by 100
                </button>
            </div>
        </div>
```


Khushal Savalakha

```
    );
}
```

↪ **Redux Store**: The central place where the application state is managed. It holds the state tree and allows state updates through dispatching actions.

what is the work of state tree and what store do ?

State Tree and Redux Store

State Tree

The **state tree** is a single JavaScript object that holds the entire state of your application in Redux. It's called a tree because it can have nested objects, arrays, and other structures, making it a hierarchical structure similar to a tree.

Here's an example of what a state tree might look like:

```
{
  counter: {
    value: 0
  },
  user: {
    name: 'John Doe',
    age: 30
  },
  todos: [
    { id: 1, text: 'Learn Redux', completed: fal
se },
    { id: 2, text: 'Build a project', completed:
true }
  ]
}
```

In this state tree:

- `counter` holds the state related to a counter feature.
- `user` holds user information.
- `todos` holds a list of to-do items.

Khushal Savalakha

Redux Store

The **Redux store** is a central object that holds the state tree and provides methods to interact with it. The store's responsibilities include:

1. **Holds the State**: It holds the entire state of the application as a single object.
2. **Dispatches Actions**: It provides a `dispatch` function to send actions to the reducers.
3. **Registers Reducers**: It uses reducers to specify how the state tree is transformed by actions.
4. **Subscribes to Changes**: It allows components to subscribe to changes in the state.

Key Methods of the Redux Store

1. `getState`:
   - Retrieves the current state of the application.

   ```
   const currentState = store.getState();
   ```

2. `dispatch`:
   - Dispatches an action to change the state.

   ```
   store.dispatch({ type: 'INCREMENT' });
   ```

3. `subscribe`:
   - Registers a callback that gets called whenever the state changes.

   ```
   const unsubscribe = store.subscribe(() => {
     console.log(store.getState());
   });
   ```

4. `replaceReducer`:
   - Replaces the current reducer with a new one. Useful for dynamic code splitting or hot reloading.

Khushal Savalakha

```
store.replaceReducer(newReducer);
```

Example: Incrementing the Counter

Let's break down what happens when you increment the counter step-by-step:

1. **Initial Setup**:
   - The state tree is initialized with the counter value.
   - The store is created with the `counterReducer`.

```
import { configureStore } from '@reduxjs/toolki
t';
import counterReducer from '../features/counter/
counterSlice';

export const store = configureStore({
  reducer: {
    counter: counterReducer,  // Registering cou
nterReducer
  },
});
```

1. **User Interaction**:
   - The user clicks the "Increment" button in the `Counter` component.
   - The button's `onClick` handler dispatches the `increment` action to the store.

```
<button onClick={() => dispatch(increment())}>In
crement</button>
```

1. **Dispatching Action**:
   - The `dispatch` function sends the `increment` action to the Redux store.

```
store.dispatch(increment());
```


Khushal Savalakhia

1. **Reducer Handles Action**:
   - The store forwards the current state and the action to the `counterReducer`.

```
export const counterSlice = createSlice
```

💡 import { decrement, increment, incrementByAmount,decrementByAmount } from './counterSlice'; // Import actions from the counter slice
but here we are import this from counter slice though when dispatch works then how request directly goes to store and it will send to reducer ?

This is a great question that dives into the flow of data and actions in Redux. Let's break it down step-by-step:

The Flow of Actions in Redux

1. **Action Creation**:
   - You import action creators (`increment`, `decrement`, etc.) from the `counterSlice`. These action creators return action objects.

```
import { decrement, increment, incrementByAmo
unt, decrementByAmount } from './counterSlic
e';
```

2. **Dispatching an Action**:
   - When you dispatch an action, it is sent to the Redux store.

```
const dispatch = useDispatch();
dispatch(increment()); // Dispatching the inc
rement action
```

3. **Store Receives the Action**:
   - The Redux store receives the dispatched action.

Khushal Savalakha

```
store.dispatch(increment());
```

4. **Store Forwards the Action to Reducers**:
   - The store forwards the action to the relevant reducer(s). In this case, it forwards it to the `counterReducer` defined in `counterSlice.js`.
5. **Reducer Handles the Action**:
   - The `counterReducer` processes the action and updates the state accordingly.

```
export const counterSlice = createSlice({
    name: 'counter',
    initialState,
    reducers: {
        increment: (state) => {
            state.value += 1;
        },
        decrement: (state) => {
            state.value -= 1;
        },
        incrementByAmount: (state, action) =>
{
            state.value += action.payload;
        },
        decrementByAmount: (state, action) =>
{
            state.value -= action.payload;
        },
    },
});

export const { increment, decrement, incremen
tByAmount, decrementByAmount } = counterSlic
e.actions;
export default counterSlice.reducer;
```

Detailed Explanation with Example

Khushal Savalakha

Let's walk through the process using an example where the `increment` action is dispatched:

1. **Action Creator**:
   - The `increment` action creator is imported from `counterSlice.js`. When called, it returns an action object.

```
// counterSlice.js
export const { increment } = counterSlice.actions;
```

2. **Dispatching the Action**:
   - In the `Counter` component, when the user clicks the "Increment" button, the `increment` action is dispatched.

```
const dispatch = useDispatch();
dispatch(increment());
```

3. **Action Object**:
   - The `increment` action creator returns an action object like this:

```
{ type: 'counter/increment' }
```

4. **Redux Store**:
   - The Redux store receives this action object and forwards it to the `counterReducer`.

```
// store.js
export const store = configureStore({
    reducer: {
        counter: counterReducer,
    },
});
```

5. **Reducer Handles the Action**:

Khushal Savalakha

- The `counterReducer` in `counterSlice.js` is responsible for updating the state based on the action type.

```
export const counterSlice = createSlice({
    name: 'counter',
    initialState,
    reducers: {
        increment: (state) => {
            state.value += 1;
        },
        decrement: (state) => {
            state.value -= 1;
        },
        incrementByAmount: (state, action) =>
{
            state.value += action.payload;
        },
        decrementByAmount: (state, action) =>
{
            state.value -= action.payload;
        },
    },
});
```

- Since the action type is `'counter/increment'`, the `increment` reducer function is executed, updating the state.

Visualization

Here's a visual representation of the flow:

> 💡 means dispatch returns the action to store because provider renders the counter component am i true ?

Yes, you're on the right track! Let's clarify and elaborate on the relationship between `dispatch`, `Provider`, and how components interact with the Redux store.

Khushal Savalakha

Understanding the Flow

1. **Provider Component**:
   - The `Provider` component from `react-redux` is used to make the Redux store available to any nested components that need to access the Redux store.
   - It wraps the entire app or the part of the app that needs access to the Redux store.

```
import React from 'react';
import ReactDOM from 'react-dom/client';
import App from './App';
import { store } from './app/store';
import { Provider } from 'react-redux';

const root = ReactDOM.createRoot(document.get
ElementById('root'));
root.render(
  <Provider store={store}>
    <App />
  </Provider>
);
```

   - By using `Provider`, the `store` is passed down through the React component tree, making it available to any nested components that are connected to the Redux store.

2. **Dispatching an Action**:
   - In a component (e.g., `Counter`), you use the `useDispatch` hook to get the `dispatch` function.
   - When an action (e.g., `increment`) is dispatched, it is sent to the Redux store.

```
import React from 'react';
import { useSelector, useDispatch } from 'rea
ct-redux';
import { increment, decrement } from './count
erSlice';
```


Khushal Savalakria

```
export default function Counter() {
  const count = useSelector((state) => state.
counter.value);
  const dispatch = useDispatch();

  return (
    <div>
      <button onClick={() => dispatch(increme
nt())}>Increment</button>
      <span>{count}</span>
      <button onClick={() => dispatch(decreme
nt())}>Decrement</button>
    </div>
  );
}
```

Detailed Flow Explanation

1. **Provider Renders Counter Component**:
   - The `Provider` component wraps your app and makes the Redux store available to all nested components.
   - In this setup, the `Counter` component is a child of the `Provider`.

```
<Provider store={store}>
  <App />
  <Counter />
</Provider>
```

2. **Counter Component Dispatches an Action**:
   - When a user interacts with the `Counter` component (e.g., clicking the "Increment" button), the `dispatch` function is called.
   - The `dispatch` function is provided by the `useDispatch` hook, which is connected to the Redux store via the `Provider`.

```
dispatch(increment());
```


Khushal Savalakha

3. **Store Receives the Action**:
   - The dispatched action (e.g., `{ type: 'counter/increment' }`) is sent to the Redux store.
   - The Redux store forwards the action to the `counterReducer`.

```
// counterSlice.js
const counterSlice = createSlice({
  name: 'counter',
  initialState: { value: 0 },
  reducers: {
    increment: (state) => {
      state.value += 1;
    },
    decrement: (state) => {
      state.value -= 1;
    },
  },
});

export const { increment, decrement } = count
erSlice.actions;
export default counterSlice.reducer;
```

4. **Reducer Updates the State**:
   - The `counterReducer` processes the action and updates the state accordingly.
   - The updated state is stored in the Redux store.

```
const counterSlice = createSlice({
  name: 'counter',
  initialState,
  reducers: {
    increment: (state) => {
      state.value += 1;
    },
    decrement: (state) => {
      state.value -= 1;
```


Khushal Savalakha

```
    },
  },
});
```

5. **Component Re-Renders with Updated State**:
   - The `useSelector` hook in the `Counter` component subscribes to the Redux store and selects the necessary state (e.g., `state.counter.value`).
   - When the state changes (e.g., the counter value is incremented), the `Counter` component re-renders with the updated state.

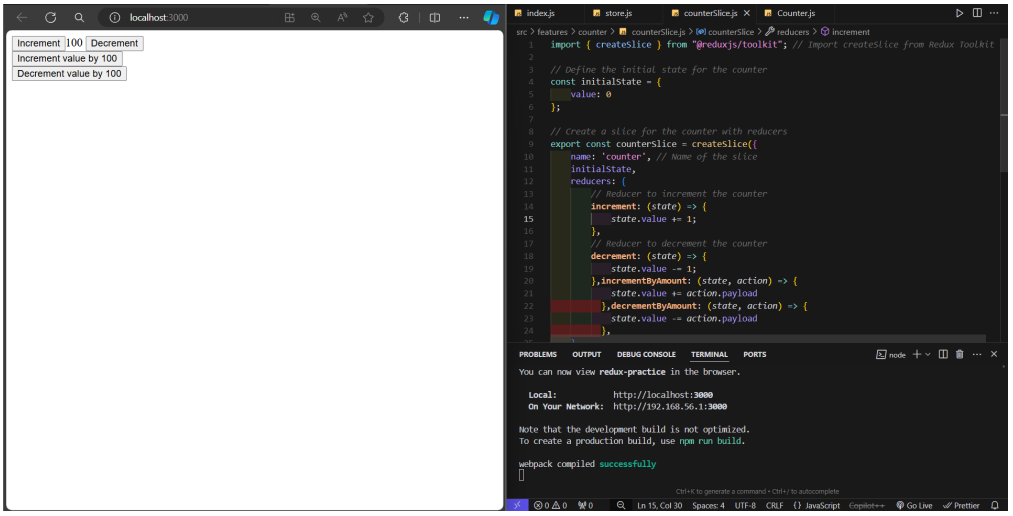
```
const count = useSelector((state) => state.counter.value);
```

Summary

- **Provider**: The `Provider` component makes the Redux store available to any nested components that need to access it.
- **Dispatch**: The `dispatch` function sends actions to the Redux store.
- **Store**: The store receives actions and forwards them to the appropriate reducer(s).
- **Reducer**: Reducers update the state based on the received actions.
- **Component Re-Render**: Components connected to the Redux store re-render when the state changes.

So, the `Provider` component ensures that the `dispatch` function in the `Counter` component is connected to the Redux store, allowing actions to be dispatched and processed correctly.

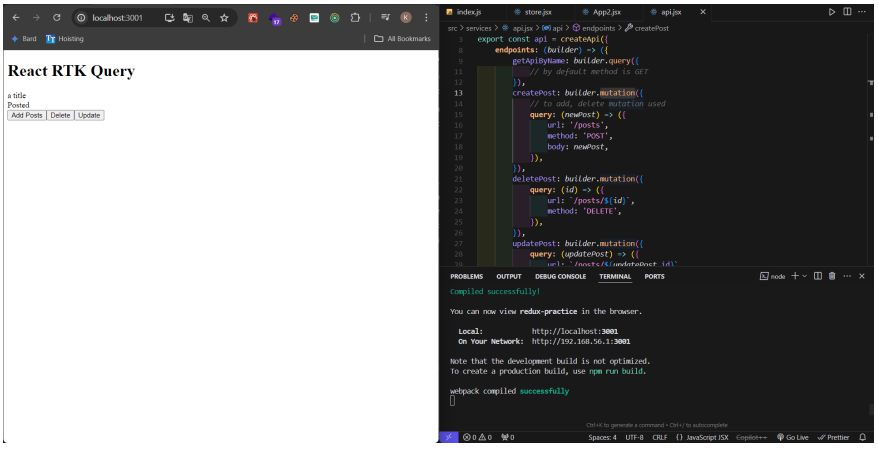
Khushal Savalakha

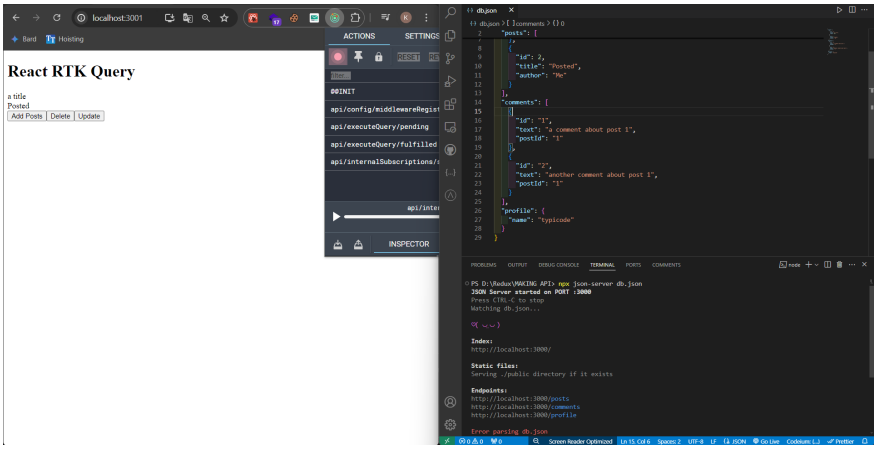


▼ RTK QUERY & CRUD

npx json-server db.json

https://github.com/typicode/json-server





index.js

Khushal Savalakha

```
import React from 'react';
import ReactDOM from 'react-dom/client'; // Import React
import App2 from './App2'
import { store } from './store'; // Import the Redux sto
import { Provider } from 'react-redux'; // Import Provid

// Create a root element to render the React component t
const root = ReactDOM.createRoot(document.getElementById

// Render the app with Redux store provider
root.render(
  <Provider store={store}>
    <App2/>
  </Provider>
);
```

store.js

```
import { configureStore } from "@reduxjs/toolkit";
import { api } from "./services/api";

export const store = configureStore({
  reducer: {
    [api.reducerPath]: api.reducer,
  },
  middleware: (getDefaultMiddleware) => {
    console.log(getDefaultMiddleware().concat(api.middle
    return getDefaultMiddleware().concat(api.middleware)
    //concat function combine default middleware to api
  },
});
```

App2.jsx

```
import React from 'react';
import { useGetApiByNameQuery, useCreatePostMutation, us

export default function App2() {
```


Khushal Savalakhia

```
    const { data, error, isLoading, isSuccess, isFetchin

    return (
        <div>
            <h1>React RTK Query</h1>
            {isLoading && <h2>Loading...</h2>}
            {isFetching && <h2>Fetching...</h2>}
            {error && <h2>Error</h2>}
            {isSuccess && (
                <div>
                    {data?.map(post => (
                        <div key={post.id}>
                            <span>{post.title}</span>
                        </div>
                    ))}
                    <AddPost />
                </div>
            )}
        </div>
    );
}

export const AddPost = () => {
    const [addPost] = useCreatePostMutation();
    const [updatePost] = useUpdatePostMutation();
    const [deletePost] = useDeletePostMutation();
    console.log(addPost);
    console.log('------------');
    console.log(useCreatePostMutation());
    const post = {
        id: 2,
        title: "Posted",
        author: "Me"
    };

    const postUpdate = {
        id: 2,
        title: "UpdatedPosted",
```


Khushal Savalakha

```
        author: "Me Updated"
    };

    const handleAdd = async () => {
        await addPost(post);
    };

    const handleDelete = async () => {
        await deletePost(post.id);
    };

    const handleUpdate = async () => {
        await updatePost(postUpdate);
    };

    return (
        <>
            <button onClick={handleAdd}>Add Posts</butto
            <button onClick={handleDelete}>Delete</butto
            <button onClick={handleUpdate}>Update</butto
        </>
    );
}
```

api.jsx

```
import { createApi, fetchBaseQuery } from '@reduxjs/tool

export const api = createApi({
    reducerPath: 'api',
    baseQuery: fetchBaseQuery({
        baseUrl: 'http://localhost:3000/',
    }),
    endpoints: (builder) => ({
        getApiByName: builder.query({
            query: () => '/posts', // query only written
            // by default method is GET
        }),
```


Khushal Savalakha

```
        createPost: builder.mutation({
            // to add, delete mutation used
            query: (newPost) => ({
                url: '/posts',
                method: 'POST',
                body: newPost,
            }),
        }),
        deletePost: builder.mutation({
            query: (id) => ({
                url: `/posts/${id}`,
                method: 'DELETE',
            }),
        }),
        updatePost: builder.mutation({
            query: (updatePost) => ({
                url: `/posts/${updatePost.id}`,
                method: 'PUT',
                body: updatePost,
            }),
        }),
    }),
});

export const { useGetApiByNameQuery, useCreatePostMutati
```

💡 it means it is a reducer but with unique name because if other counter slice are availabel then they also have they reducer so reducerpath it's like key by which internally store access states a i true ?

Yes, you're correct! Let's break it down clearly:

Understanding `reducerPath`

- **Reducer with a Unique Name**: The `reducerPath` is essentially a unique name or key under which the reducer for the RTK Query API slice is mounted in the

Khushal Savalakha

Redux state tree. This helps in distinguishing it from other slices or reducers that might exist in your application.

- **Avoiding Collisions**: By specifying a unique `reducerPath`, you avoid state collisions and ensure that the API-related state is organized under a specific namespace within the Redux state.

Summary

- `reducerPath` determines where in the Redux state tree the API slice's state will be stored.
- `state.api` accesses the state managed by the API slice, allowing you to read data, loading states, errors, etc.
- This setup helps keep the state organized and makes it easier to manage API-related state in your application.

Sure, let's go through the code step-by-step to understand how it works in detail.

index.js

```
import React from 'react';
import ReactDOM from 'react-dom/client'; // Import Rea
ctDOM to render the React component tree
import App2 from './App2';
import { store } from './store'; // Import the Redux s
tore
import { Provider } from 'react-redux'; // Import Prov
ider to connect the Redux store with the React app

// Create a root element to render the React component
tree
const root = ReactDOM.createRoot(document.getElementBy
Id('root'));

// Render the app with Redux store provider
root.render(
  <Provider store={store}>
    <App2 />
```


Khushal Savalakha

```
  </Provider>
);
```

Explanation:

1. **Imports**:
   - React and ReactDOM are imported to create and render the component tree.
   - `App2` is imported as the main component of the application.
   - `store` is imported from the Redux setup to manage the state.
   - `Provider` from `react-redux` is used to connect the Redux store with the React app.
2. **Create a Root Element**:
   - `ReactDOM.createRoot` is used to create a root element to render the component tree.
3. **Render the App**:
   - The `Provider` component wraps the `App2` component, passing the Redux `store` to it. This allows any component within `App2` to access the Redux store.

`store.jsx`

```
import { configureStore } from "@reduxjs/toolkit";
import { api } from "./services/api";

export const store = configureStore({
  reducer: {
    [api.reducerPath]: api.reducer,
  },
  middleware: (getDefaultMiddleware) => {
    console.log(getDefaultMiddleware().concat(api.midd
leware));
    return getDefaultMiddleware().concat(api.middlewar
e);
  },
```


Khushal Savalakha

```
});
// setupListeners(store.dispatch)
```

Explanation:

1. **Imports**:
   - `configureStore` from `@reduxjs/toolkit` is used to configure the Redux store.
   - `api` is imported from `./services/api`.
2. **Configure the Store**:
   - The store is configured with a reducer. `api.reducerPath` is used as the key, and `api.reducer` is used as the value. This connects the API reducer to the Redux store.
   - The `middleware` option is used to customize the middleware. It concatenates the default middleware with the `api.middleware`, ensuring that API-related middleware is included.
3. **Note**:
   - The `setupListeners` function call is commented out. It is generally used to set up automatic cache invalidation for the RTK Query.

`App2.jsx`

```
import React from 'react';
import { useGetApiByNameQuery, useCreatePostMutation,
useDeletePostMutation, useUpdatePostMutation } from
'./services/api'; // Correct import

export default function App2() {
    const { data, error, isLoading, isSuccess, isFetch
ing } = useGetApiByNameQuery();

    return (
        <div>
            <h1>React RTK Query</h1>
            {isLoading && <h2>Loading...</h2>}
            {isFetching && <h2>Fetching...</h2>}
            {error && <h2>Error</h2>}
```


Khushal Savalakha

```
            {isSuccess && (
                <div>
                    {data?.map(post => (
                        <div key={post.id}>
                            <span>{post.title}</span>
                        </div>
                    ))}
                    <AddPost />
                </div>
            )}
        </div>
    );
}

export const AddPost = () => {
    const [addPost] = useCreatePostMutation();
    const [updatePost] = useUpdatePostMutation();
    const [deletePost] = useDeletePostMutation();

    const post = {
        id: 2,
        title: "Posted",
        author: "Me"
    };

    const postUpdate = {
        id: 2,
        title: "UpdatedPosted",
        author: "Me Updated"
    };

    const handleAdd = async () => {
        await addPost(post);
    };

    const handleDelete = async () => {
        await deletePost(post.id);
    };
```


Khushal Savalakha

```
    const handleUpdate = async () => {
        await updatePost(postUpdate);
    };

    return (
        <>
            <button onClick={handleAdd}>Add Posts</but
ton>
            <button onClick={handleDelete}>Delete</but
ton>
            <button onClick={handleUpdate}>Update</but
ton>
        </>
    );
}
```

Explanation:

1. **Imports**:
   - Necessary hooks are imported from `./services/api` to interact with the API.
2. `App2` **Component**:
   - Uses `useGetApiByNameQuery` hook to fetch data.
   - Destructures the result into `data`, `error`, `isLoading`, `isSuccess`, and `isFetching`.
   - Conditionally renders different UI elements based on the query state (`isLoading`, `isFetching`, `error`, `isSuccess`).
   - Maps over the fetched data to display posts and includes the `AddPost` component.
3. `AddPost` **Component**:
   - Uses hooks for mutations: `useCreatePostMutation`, `useUpdatePostMutation`, `useDeletePostMutation`.
   - Defines post data and update data.
   - `handleAdd`, `handleDelete`, and `handleUpdate` are asynchronous functions that call the respective mutation hooks with the post data.

Khushal Savalakha

- Returns buttons to trigger these actions.

api.jsx

```
import { createApi, fetchBaseQuery } from '@reduxjs/to
olkit/query/react';

export const api = createApi({
    reducerPath: 'api',
    baseQuery: fetchBaseQuery({
        baseUrl: '<http://localhost:3000/>',
    }),
    endpoints: (builder) => ({
        getApiByName: builder.query({
            query: () => '/posts',
        }),
        createPost: builder.mutation({
            query: (newPost) => ({
                url: '/posts',
                method: 'POST',
                body: newPost,
            }),
        }),
        deletePost: builder.mutation({
            query: (id) => ({
                url: `/posts/${id}`,
                method: 'DELETE',
            }),
        }),
        updatePost: builder.mutation({
            query: (updatePost) => ({
                url: `/posts/${updatePost.id}`,
                method: 'PUT',
                body: updatePost,
            }),
        }),
    }),
});
```


Khushal Savalakha

```
export const { useGetApiByNameQuery, useCreatePostMutation, useDeletePostMutation, useUpdatePostMutation } = api;
```

Explanation:

1. **Imports**:
   - `createApi` and `fetchBaseQuery` from `@reduxjs/toolkit/query/react`.
2. **API Configuration**:
   - `createApi` is used to define the API endpoints and base query configuration.
   - `reducerPath` sets the key for the API slice in the Redux store.
   - `baseQuery` defines the base URL for the API.
   - `endpoints` function takes a `builder` to define different endpoints.
     - `getApiByName`: Fetches a list of posts with a `GET` request to `/posts`.
     - `createPost`: Adds a new post with a `POST` request to `/posts`.
     - `deletePost`: Deletes a post with a `DELETE` request to `/posts/{id}`.
     - `updatePost`: Updates a post with a `PUT` request to `/posts/{id}`.
3. **Export Hooks**:
   - Hooks are generated for each endpoint and exported for use in components.

Summary

- **index.js** sets up the main React app with Redux store integration.
- **store.jsx** configures the Redux store with RTK Query API reducer and middleware.
- **App2.jsx** demonstrates usage of query and mutation hooks to interact with the API, including fetching, adding, deleting, and updating posts.
- **api.jsx** defines the API endpoints and generates hooks for querying and mutating data.

This setup allows for a scalable and maintainable way to manage API calls and state in a React-Redux application using RTK Query.

Khushal Savalakria

> 💡 The middleware option is used to customize the middleware. It concatenates the default middleware with the api.middleware, ensuring that API-related middleware is included.
> here i don't understand here what is the default middleware? and how it concat with api middle ware explain simply with example.

Certainly! Let's break down what default middleware is and how it is concatenated with the `api` middleware in a simple example.

What is Middleware?

Middleware in Redux is a way to extend the functionality of Redux. It sits between the action dispatch and the reducer, allowing you to perform side effects or process actions before they reach the reducer.

Default Middleware

When you use `configureStore` from Redux Toolkit, it automatically includes some default middleware. These typically include:

- `redux-thunk`: Allows you to write action creators that return a function instead of an action, useful for asynchronous logic.
- `redux-immutable-state-invariant`: Warns you if you accidentally mutate the state in Redux (only in development).
- `redux-logger`: Logs actions and state changes to the console (optional and not included by default).

Adding Custom Middleware

In your case, you are adding the `api.middleware` to the default middleware. The `api.middleware` is required by RTK Query to handle caching, invalidation, polling, and other functionalities for API calls.

Example

Let's see a simple example to illustrate how middleware is added:

```
import { configureStore, getDefaultMiddleware } from
'@reduxjs/toolkit';
import { api } from './services/api';
```


Khushal Savalakha

```
// Create a Redux store
export const store = configureStore({
  reducer: {
    [api.reducerPath]: api.reducer,
  },
  middleware: (getDefaultMiddleware) => {
    // Concatenate default middleware with api middlew
are
    return getDefaultMiddleware().concat(api.middlewar
e);
  },
});
```

Breakdown

1. `getDefaultMiddleware`:
   - This function returns an array of the default middleware.
2. **Concatenation**:
   - `getDefaultMiddleware().concat(api.middleware)`:
     - `getDefaultMiddleware()` returns the default middleware as an array.
     - `.concat(api.middleware)` adds the `api.middleware` to the end of this array.

Visual Example

Let's visualize it step by step:

1. **Default Middleware**:

```
const defaultMiddleware = getDefaultMiddleware();
// defaultMiddleware is an array like [thunk, immut
ableStateInvariant]
```

2. **Concatenating api.middleware**:

```
const combinedMiddleware = defaultMiddleware.concat
(api.middleware);
```


Khushal Savalakha

```
// combinedMiddleware is now [thunk, immutableState
Invariant, api.middleware]
```

3. **Using in configureStore**:

```
export const store = configureStore({
  reducer: {
    [api.reducerPath]: api.reducer,
  },
  middleware: (getDefaultMiddleware) => {
    return getDefaultMiddleware().concat(api.middle
ware);
  },
});
```

Simplified Analogy

Imagine you have a default list of tasks you perform every day:

1. **defaultTasks**: ["Wake up", "Brush teeth", "Have breakfast"]
2. Now, you have a special task you need to do: "Exercise"
3. **Concatenating the special task**:
   - `combinedTasks = defaultTasks.concat("Exercise")`
   - Now, `combinedTasks` is: ["Wake up", "Brush teeth", "Have breakfast", "Exercise"]

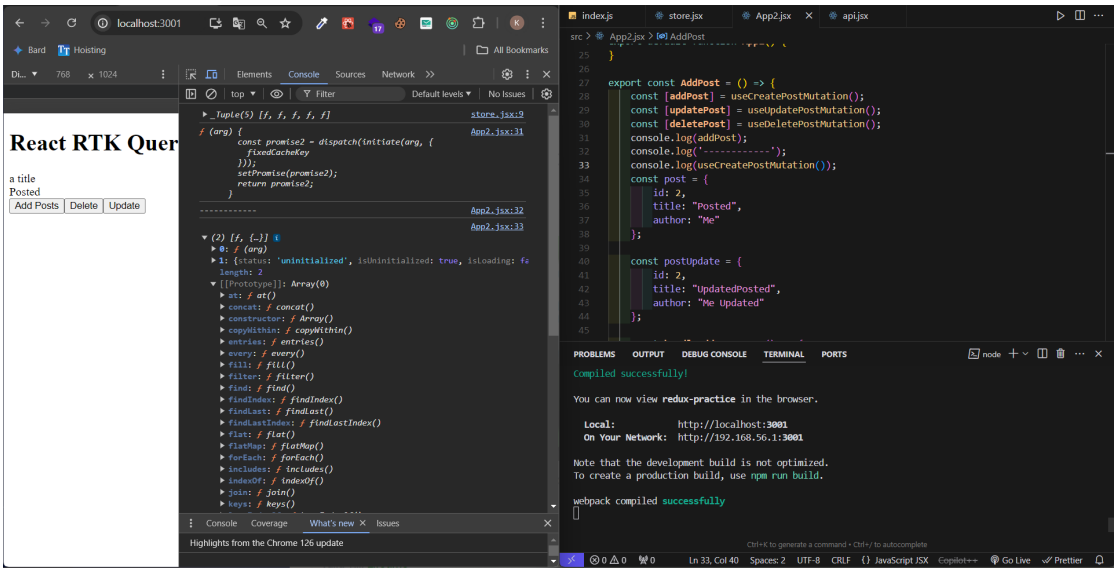
In the same way, `getDefaultMiddleware()` provides the default middleware tasks, and `concat(api.middleware)` adds your special `api.middleware` task to this list.

Conclusion

The `middleware` option in `configureStore` allows you to customize which middleware your Redux store uses. By default, Redux Toolkit includes some common middleware. You can add your custom middleware, such as `api.middleware` from RTK Query, to this default set using the `.concat()` method. This ensures that all necessary middleware, both default and custom, are included in your Redux store.

Khushal Savalakha

why we can't write like this ? const addPost = useCreatePostMutation(); it gives error.



In the image, we can see the console log output of calling `useCreatePostMutation()`, which indeed shows that it returns an array. The first element in this array is the function to trigger the mutation, and the second element is an object containing the state of the mutation. This is why the correct way to use it involves array destructuring.

When you do:

```
const [addPost] = useCreatePostMutation();
```

You are extracting the first element of the array returned by `useCreatePostMutation()`, which is the function to trigger the mutation. The rest of the array, which contains metadata about the mutation, is ignored unless you explicitly destructure it.

Explanation

The array returned by `useCreatePostMutation()` contains:

1. **Mutation Function**: This is the function you call to trigger the mutation.
2. **Mutation State Object**: This object contains the state of the mutation, including properties like `isLoading`, `error`, `isSuccess`, etc.

Array Destructuring

Khushal Savalakha

Array destructuring allows you to extract elements from an array and assign them to variables. In this case:

```
const [addPost, { isLoading, error }] = useCreatePostMutation();
```

- `addPost` is the mutation function.
- `{ isLoading, error }` is the mutation state object.

Example

Here's how you use it in a component:

```
import React from 'react';
import { useCreatePostMutation } from './services/api';

const AddPostComponent = () => {
  const [addPost, { isLoading, error }] = useCreatePostMutation();

  const posts = {
    id: 2,
    title: "Posted",
    author: "Me"
  };

  const handler = async () => {
    await addPost(posts);
  };

  return (
    <>
      <button onClick={handler}>Add Post</button>
      {isLoading && <p>Loading...</p>}
      {error && <p>Error: {error.message}</p>}
    </>
  );
};
```


Khushal Savalakha

```
export default AddPostComponent;
```

Conclusion

When `useCreatePostMutation()` is called, it returns an array where the first element is the mutation function and the second element is an object with mutation state information. By destructuring this array, you can directly access and use the mutation function (`addPost` in this case) and handle its state (like `isLoading` and `error`).

> 💡 Custom State Names: You can use any names for your endpoints, but the hooks will be generated based on these names using a specific naming convention (use<OperationName>Mutation for mutations and use<OperationName>Query for queries).

> 💡 it means like useState we are creating use<OperationName>Mutation thet will create a new state and when we assign get,Put,Post state by setting method name and it will genearate function am i true ?

Yes, you're correct in understanding the naming convention used by RTK Query for generating hooks based on endpoint names. Let's clarify this with an example and breakdown:

Naming Convention for Hooks

RTK Query follows a specific naming convention to generate hooks (`use<OperationName>Mutation` for mutations and `use<OperationName>Query` for queries) based on the names you provide when defining endpoints.

Example

Let's take an example API configuration where we define endpoints with specific names:

```
export const api = createApi({
    reducerPath: 'api',
    baseQuery: fetchBaseQuery({
        baseUrl: '<http://localhost:3000/>',
```


Khushal Savalakha

```
    }),
    endpoints: (builder) => ({
        getApiByName: builder.query({
            query: () => '/posts',
        }),
        createPost: builder.mutation({
            query: (newPost) => ({
                url: '/posts',
                method: 'POST',
                body: newPost,
            }),
        }),
        updatePost: builder.mutation({
            query: (updatedPost) => ({
                url: `/posts/${updatedPost.id}`,
                method: 'PUT',
                body: updatedPost,
            }),
        }),
        deletePost: builder.mutation({
            query: (id) => ({
                url: `/posts/${id}`,
                method: 'DELETE',
            }),
        }),
    }),
});
```

Generated Hooks

Based on the above configuration, RTK Query will generate the following hooks:

- **Queries**:
  - `useGetApiByNameQuery`: This hook will fetch data using the `getApiByName` endpoint.
- **Mutations**:
  - `useCreatePostMutation`: This hook will perform a POST request using the `createPost` endpoint.

Khushal Savalakha

- `useUpdatePostMutation`: This hook will perform a PUT request using the `updatePost` endpoint.
- `useDeletePostMutation`: This hook will perform a DELETE request using the `deletePost` endpoint.

How Hooks Work

- **Mutation Hooks** (`use<OperationName>Mutation`):
  - When you call `useCreatePostMutation()`, for example, it returns a function (`mutate`) that you can call to initiate the POST request defined in `createPost` endpoint. This function handles the HTTP request and updates the Redux state accordingly (loading state, success, or error).
- **Query Hooks** (`use<OperationName>Query`):
  - When you call `useGetApiByNameQuery()`, it returns an object (`data`, `isLoading`, `error`, etc.) that represents the state of the GET request defined in `getApiByName` endpoint. It handles fetching data asynchronously and manages the state changes based on the API response.

Summary

- RTK Query uses a naming convention (`use<OperationName>Mutation` and `use<OperationName>Query`) to generate hooks based on the endpoint names you provide.
- These hooks encapsulate the logic for making API requests (`mutation` for modifying data and `query` for fetching data) and managing their respective states within Redux.
- By following this convention, RTK Query makes it easy to integrate API interactions into your React components with minimal boilerplate code.

it means it is like state useState of react or not with different work ?=yes it is.

In the image, we can see the console log output of calling `useCreatePostMutation()`, which indeed shows that it returns an array. The first element in this array is the function to trigger the mutation, and the second element is an object containing the state of the mutation. This is why the correct way to use it involves array destructuring.

When you do:

Khushal Savalakha

```
const [addPost] = useCreatePostMutation();
```

You are extracting the first element of the array returned by `useCreatePostMutation()`, which is the function to trigger the mutation. The rest of the array, which contains metadata about the mutation, is ignored unless you explicitly destructure it.

Explanation

The array returned by `useCreatePostMutation()` contains:

1. **Mutation Function**: This is the function you call to trigger the mutation.
2. **Mutation State Object**: This object contains the state of the mutation, including properties like `isLoading`, `error`, `isSuccess`, etc.

Array Destructuring

Array destructuring allows you to extract elements from an array and assign them to variables. In this case:

```
const [addPost, { isLoading, error }] = useCreatePostM
utation();
```

- `addPost` is the mutation function.
- `{ isLoading, error }` is the mutation state object.

Example

Here's how you use it in a component:

```
import React from 'react';
import { useCreatePostMutation } from './services/ap
i';

const AddPostComponent = () => {
  const [addPost, { isLoading, error }] = useCreatePos
tMutation();

  const posts = {
    id: 2,
    title: "Posted",
```


Khushal Savalakha

```
    author: "Me"
  };

  const handler = async () => {
    await addPost(posts);
  };

  return (
    <>
      <button onClick={handler}>Add Post</button>
      {isLoading && <p>Loading...</p>}
      {error && <p>Error: {error.message}</p>}
    </>
  );
};

export default AddPostComponent;
```

Conclusion

When `useCreatePostMutation()` is called, it returns an array where the first element is the mutation function and the second element is an object with mutation state information. By destructuring this array, you can directly access and use the mutation function (`addPost` in this case) and handle its state (like `isLoading` and `error`).

> 💡 what is the work of Builder and mutation?

In the context of RTK Query, `Builder` and `Mutation` are concepts used to define API endpoints within the `endpoints` configuration function. Let's break down their roles:

Builder

- **Purpose**: The `builder` object is provided as a parameter to the `endpoints` function when configuring your API with RTK Query.
- **Usage**: It offers methods (`query` and `mutation`) to define different types of API operations:

Khushal Savalakria

- `builder.query`: Used to define operations that fetch data from the server (typically corresponds to HTTP GET requests).
- `builder.mutation`: Used to define operations that modify data on the server (typically corresponds to HTTP POST, PUT, DELETE requests).

Mutation

- **Purpose**: `Mutation` is a specific method provided by the `builder` object.
- **Usage**: When you call `builder.mutation`, you define an operation that will perform an asynchronous HTTP request to mutate (create, update, delete) data on the server.
- **Parameters**: It takes an object where you define:
  - `query` **function**: Specifies the details of the HTTP request (URL, method, body) based on the operation you want to perform.

Example

Here's an example that demonstrates the use of `builder.mutation` to define a mutation operation:

```
import { createApi, fetchBaseQuery } from '@reduxjs/to
olkit/query/react';

export const api = createApi({
    reducerPath: 'api',
    baseQuery: fetchBaseQuery({
        baseUrl: '<http://localhost:3000/>',
    }),
    endpoints: (builder) => ({
        createPost: builder.mutation({
            // Define the mutation operation to create
a new post
            query: (newPost) => ({
                url: '/posts',
                method: 'POST',
                body: newPost,
            }),
        }),
```


Khushal Savalakha

```
        updatePost: builder.mutation({
            // Define the mutation operation to update
an existing post
            query: (updatedPost) => ({
                url: `/posts/${updatedPost.id}`,
                method: 'PUT',
                body: updatedPost,
            }),
        }),
        deletePost: builder.mutation({
            // Define the mutation operation to delete
a post
            query: (id) => ({
                url: `/posts/${id}`,
                method: 'DELETE',
            }),
        }),
    }),
});
```

In this example:

- `builder.mutation` is used three times to define `createPost`, `updatePost`, and `deletePost` mutations.
- Each mutation specifies a `query` function that returns an object detailing the HTTP request (`url`, `method`, `body`) necessary to perform the desired operation on the server.

Summary

- **Builder**: Provides methods (`query` and `mutation`) to define API operations within the `endpoints` configuration function.
- **Mutation**: A method of `builder` used to define operations that modify data on the server (POST, PUT, DELETE requests).
- **Usage**: Use `builder.mutation` to define the specifics of each mutation operation, including the HTTP request details needed to interact with your API effectively.

▼ Quick explain

Khushal Savalakhia

Sure, let's create a quick flow to visualize how the components and configurations work together in the provided code example.

File/Component	Purpose
index.js	Sets up the React application, renders the main `App2` component wrapped with Redux store provider.
store.jsx	Configures the Redux store using `configureStore` from RTK, integrates the API reducer and middleware.
App2.jsx	React component that fetches and displays posts using `useGetApiByNameQuery` hook. Includes `AddPost` component for creating, updating, and deleting posts.
api.jsx	Defines the API configuration with endpoints (`getApiByName` , `createPost` , `deletePost` , `updatePost`). Generates hooks (`useGetApiByNameQuery` , `useCreatePostMutation` , etc.) for querying and mutating data.

Flow Explanation

1. **index.js**:
   - Initializes the React application and renders the main component `App2`.
   - Provides Redux store (`store`) to the entire application using `Provider`.
2. **store.jsx**:
   - Configures the Redux store using `configureStore` from `@reduxjs/toolkit`.
   - Incorporates the API reducer (`api.reducer`) into the store's reducer setup.
   - Configures middleware to handle asynchronous actions, including API requests.
3. **App2.jsx**:
   - Uses `useGetApiByNameQuery` hook to fetch a list of posts (`data`).
   - Manages loading states (`isLoading` , `isFetching`), error handling (`error`), and success (`isSuccess`) states.
   - Maps through `data` to display posts and includes UI elements based on query states.
   - Contains `AddPost` component, which allows creating, updating, and deleting posts using mutation hooks (`useCreatePostMutation` ,

Khushal Savalakha

`useUpdatePostMutation`, `useDeletePostMutation`).

4. **api.jsx**:
   - Defines the API configuration using `createApi` from `@reduxjs/toolkit/query/react`.
   - Specifies `reducerPath` as `'api'` and `baseUrl` as `'<http://localhost:3000/'`>.
   - Configures endpoints (`getApiByName`, `createPost`, `deletePost`, `updatePost`) using `builder.query` and `builder.mutation`.
   - Generates hooks (`useGetApiByNameQuery`, `useCreatePostMutation`, etc.) based on endpoint names for querying and mutating data.

Visual Flow Summary

- **Data Flow**: React components (`App2`, `AddPost`) fetch and manipulate data using hooks generated by RTK Query (`use<OperationName>`).
- **State Management**: Redux manages application state including API data and UI states.
- **Component Hierarchy**: `index.js` sets up the application, `App2` manages main UI and data fetching, `AddPost` handles post operations.
- **API Integration**: Configured in `api.jsx`, RTK Query simplifies API interaction with predefined endpoints and hooks.

This flow illustrates how the React components, Redux store, and RTK Query work together to manage state and API interactions in a structured and efficient manner.

```
updatePost: builder.mutation({
            query: (updatePost) => ({
                url: `/posts/${updatePost.id}`,
                method: 'PUT',
                body: updatePost,
            }),
        }),
```

Yes, you're correct. The `updatePost` mutation in the given code snippet is designed to update an existing post. Here’s a breakdown of how it works:

Khushal Savalakha

1. **Function Definition**: `updatePost: builder.mutation({ ... })` defines a mutation named `updatePost`.
2. **Query Function**: Inside the mutation, the `query` function takes an argument (`updatePost`), which represents the new post data.
3. **URL Construction**: The URL for the API request is constructed using the ID from `updatePost` (`/posts/${updatePost.id}`). This means the endpoint will target the specific post that needs to be updated.
4. **HTTP Method**: The method is set to `'PUT'`, indicating that this is an update operation.
5. **Request Body**: The body of the request is the `updatePost` object, which contains the new data for the post.

So, in summary, the `updatePost` mutation will send a PUT request to the endpoint for the specific post ID, replacing the old post data with the new data provided in `updatePost`.

▼ REDUX EXPLANATION 2

> 💡 here i don't understand how login and logout change the user state value how create slice works internally and why we have to export and import many things ?

Let's delve into how **Redux's `createSlice` works internally, how it handles state changes**, and why certain exports and imports are necessary.

How `createSlice` Works Internally

1. **Creating a Slice**

   When you use `createSlice` from Redux Toolkit, you define a slice of the Redux store with an initial state and reducers. Here's a basic example:

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

export const userSlice = createSlice({
  name: 'user',
```


Khushal Savalakria

```
  initialState: {
    user: null,
  },
  reducers: {
    login: (state, action) => {
      state.user = action.payload; // Sets the user s
tate with action payload
    },
    logout: (state) => {
      state.user = null; // Clears the user state
    },
  },
});

export const { login, logout } = userSlice.actions;
export const selectUser = (state) => state.user.user;
export default userSlice.reducer;
```

Explanation:

- `name`: Defines the slice's name, which is used in the Redux store to identify this part of the state.
- `initialState`: Sets the initial value of the state for this slice.
- `reducers`: Defines the functions that will handle actions to update the state. Each reducer function receives the current state and an action. The state is updated directly inside these reducer functions (using the Immer library under the hood to handle immutability).

2. **Reducers and Actions**

   - **Reducers** are functions that specify how the state changes in response to actions. In `createSlice`, each reducer automatically becomes an action creator. For example, the `login` reducer changes the state based on `action.payload`, and `logout` clears the user state.
   - **Actions** are dispatched to the Redux store to trigger state changes. When you call `dispatch(login(userData))`, it executes the `login` reducer with `userData` as the payload.

3. **Exporting and Importing**

Khushal Savalakha

- **Exporting Actions**: You export actions (`login`, `logout`) so other parts of your application can dispatch these actions. For example, you use `dispatch(login(userData))` in your components to update the state.

```
export const { login, logout } = userSlice.actions;
```

- **Exporting Selectors**: You export selector functions (`selectUser`) to allow components to access specific parts of the state. This function is used with the `useSelector` hook to get the user data from the state.

```
export const selectUser = (state) => state.user.user;
```

- **Exporting Reducer**: The slice reducer is exported as the default export so it can be used in the store configuration. The reducer function is responsible for handling the state updates defined in the reducers.

```
export default userSlice.reducer;
```

4. **How State Changes**

- **Login Action**: When `dispatch(login(userData))` is called, Redux Toolkit uses the `login` reducer function to update the state. The `user` field in the state is set to `userData`.

```
login: (state, action) => {
  state.user = action.payload; // Action payload is used to set user data
},
```

- **Logout Action**: When `dispatch(logout())` is called, the `logout` reducer sets the `user` field in the state to `null`, effectively clearing the user data.

```
logout: (state) => {
  state.user = null; // Action sets user data to null
},
```


Khushal Savalakha

Why Export and Import?

1. **Separation of Concerns**: By exporting and importing actions, selectors, and reducers, you separate different concerns and make your code modular. This modularity helps in maintaining and scaling your codebase.
2. **Reusability**: Actions and selectors can be reused in different components or middleware. For instance, the `login` action is used in the login component to update the state, and the `selectUser` selector is used in various components to access user data.
3. **Store Configuration**: The store configuration needs to know which reducers to use, which is why the reducer is exported and imported into the store setup.

```
import userReducer from './features/userSlice';

export default configureStore({
  reducer: {
    user: userReducer,
  },
});
```

Summary

- `createSlice` simplifies Redux setup by combining action creators and reducers in one place.
- **Reducers** update the state based on dispatched actions.
- **Actions** are dispatched to trigger state changes.
- **Selectors** are used to access specific parts of the state.
- **Exports and Imports** help manage modular and maintainable code.

By understanding this flow, you can handle more complex state logic in Redux effectively.

Khushal Savalakha

> 💡 here i don't understand why have to send login function in dispatch(login(userData)) and what is the work of Store.jsx ,if we export reducer then why we have to export reducer and what is the need of Store.jsx by making flow chart explain how each and everything is connected here .

Let's clarify these concepts with a detailed explanation and a flowchart to show how everything connects.

Why Dispatch `login(userData)`

When you call `dispatch(login(userData))`, you're performing a few key steps:

1. **Action Creation**: The `login` function is an action creator generated by `createSlice`. It creates an action object with a type (like `'user/login'`) and payload (i.e., the `userData`).
2. **Dispatching Action**: When you call `dispatch`, Redux Toolkit sends this action object to the Redux store.
3. **Reducer Handling**: The Redux store then uses the relevant reducer function (`login` in this case) to process the action and update the state based on the action payload.

Flow:

- `dispatch(login(userData))` → Creates an action object with `userData`.
- **Redux Store** → Receives the action and uses the `login` reducer to update the state with `userData`.

Role of `Store.jsx`

The `Store.jsx` file is where you configure the Redux store. It sets up how the Redux state is structured and which reducers are used.

Key Points:

1. **Reducer Integration**: You import and combine reducers into the store configuration. This setup tells Redux how to update its state using the reducers.
2. **Store Creation**: The `configureStore` function from Redux Toolkit creates the store, integrating the reducers and middleware.

Example:

Khushal Savalakha

```
import { configureStore } from '@reduxjs/toolkit';
import userReducer from '../features/UserSlice'; // Impo
rting the user reducer

export default configureStore({
  reducer: {
    user: userReducer, // Adding the user reducer to the
store
  },
});
```

Flow:

- `configureStore` → Creates a Redux store with the specified reducers.
- `userReducer` → Handles state updates for the `user` slice.

Why Export Reducer

When you export the reducer from `UserSlice.jsx`, you're making it available for the store configuration in `Store.jsx`. This is necessary because:

1. **Store Configuration**: The store needs to know which reducers to use. By exporting the reducer, you provide the necessary information to the store configuration.
2. **Modularity**: Exporting the reducer allows you to keep the reducer logic separate from the store configuration, promoting a modular codebase.

Flowchart

Here's a flowchart to illustrate how everything is connected:

```
[Login Component] ---> [dispatch(login(userData))]
       |
       v
   [Redux Store]
   |      |
   |      v
[Reducers] (login reducer updates state with userData)
   |
   v
[Updated State]
```


Khushal Savalakria

```
[Logout Component] ---> [dispatch(logout())]
        |
        v
    [Redux Store]
    |       |
    |       v
[Reducers] (logout reducer clears user data)
    |
    v
[Updated State]
```

Detailed Explanation

1. **Login Process**:
   - **Login Component**: User enters email, username, and password. On form submission, `dispatch(login(userData))` is called.
   - **Action Creation**: `login(userData)` creates an action with type `user/login` and payload `userData`.
   - **Redux Store**: The store receives the action and passes it to the `login` reducer.
   - **Reducer**: The `login` reducer updates the state with `userData`.
   - **Updated State**: The new state, which includes the user data, is now available to other components.
2. **Logout Process**:
   - **Logout Component**: User clicks logout. `dispatch(logout())` is called.
   - **Action Creation**: `logout()` creates an action with type `user/logout` and no payload.
   - **Redux Store**: The store receives the action and passes it to the `logout` reducer.
   - **Reducer**: The `logout` reducer clears the user data from the state.
   - **Updated State**: The state is updated to remove the user data.

By understanding this flow and how each part connects, you can better grasp how Redux manages state changes and actions in your application.

Khushal Savalakha

💡 ALL DATA EXPLAINED

Absolutely! To make Redux concepts clear for beginners and help them understand how to handle complex logic, let's break it down with simplified explanations, visualizations, and flowcharts.

Understanding Redux with an Example: User Authentication

Imagine you have a web application where users can log in and log out. We'll use Redux to manage user authentication state. Here's a simplified guide to how this works:

1. What is Redux?

Redux is a state management library for JavaScript applications. It helps you manage the state of your app in a predictable way.

2. Key Concepts

- **Store**: The central place where the state of your application is stored.
- **Actions**: Plain JavaScript objects that describe what happened in the application.
- **Reducers**: Functions that specify how the state changes in response to an action.
- **Dispatch**: A function used to send actions to the Redux store.

Example Components

We'll use two components: **Login** and **Logout**.

Login Component

- **Purpose**: To log in a user by sending their data to the Redux store.
- **What it does**: It updates the state with user data (email, username, and password).

Code:

```
// Login.jsx
import React, { useState } from 'react';
import { useDispatch } from 'react-redux';
import { login } from '../features/UserSlice';

export default function Login() {
```


Khushal Savalakha

```
  const [email, setEmail] = useState('');
  const [username, setUsername] = useState('');
  const [password, setPassword] = useState('');
  const dispatch = useDispatch();

  const handleLogin = (e) => {
    e.preventDefault();
    dispatch(login({ email, username, password }));
  };

  return (
    <form onSubmit={handleLogin}>
      <input value={email} onChange={(e) => setEmail(e.t
arget.value)} placeholder="Email" />
      <input value={username} onChange={(e) => setUserna
me(e.target.value)} placeholder="Username" />
      <input type="password" value={password} onChange=
{(e) => setPassword(e.target.value)} placeholder="Passwo
rd" />
      <button type="submit">Login</button>
    </form>
  );
}
```

Logout Component

- **Purpose**: To log out a user by clearing their data from the Redux store.
- **What it does**: It sets the state to null, effectively logging out the user.

Code:

```
// Logout.jsx
import React from 'react';
import { useDispatch, useSelector } from 'react-redux';
import { logout, selectUser } from '../features/UserSlic
e';

export default function Logout() {
  const dispatch = useDispatch();
```


Khushal Savalakha

```
  const user = useSelector(selectUser);

  const handleLogout = () => {
    dispatch(logout());
  };

  return (
    <div>
      <h1>Welcome, {user?.username}</h1>
      <button onClick={handleLogout}>Logout</button>
    </div>
  );
}
```

Redux Setup

1. Create a Slice

This file defines how the state changes based on actions. It includes reducers for login and logout actions.

Code:

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

export const userSlice = createSlice({
  name: 'user',
  initialState: {
    user: null,
  },
  reducers: {
    login: (state, action) => {
      state.user = action.payload;
    },
    logout: (state) => {
      state.user = null;
    },
  },
```


Khushal Savalakha

```
});

export const { login, logout } = userSlice.actions;
export const selectUser = (state) => state.user.user;
export default userSlice.reducer;
```

2. Configure the Store

The store holds the state of your application. It uses the reducers to update the state based on actions.

Code:

```
// Store.jsx
import { configureStore } from '@reduxjs/toolkit';
import userReducer from '../features/UserSlice';

export default configureStore({
  reducer: {
    user: userReducer,
  },
});
```

3. Provide the Store

Wrap your app with the `Provider` component to make the Redux store available to all components.

Code:

```
// index.jsx
import React from 'react';
import ReactDOM from 'react-dom';
import { Provider } from 'react-redux';
import store from './app/Store';
import App from './App';

ReactDOM.render(
  <Provider store={store}>
    <App />
```


Khushal Savalakhia

```
  </Provider>,
  document.getElementById('root')
);
```

Flowchart of Redux Integration

Here's a flowchart showing how everything connects:

```
[Login Component]
    |
    v
[dispatch(login(userData))]
    |
    v
[Redux Store]
    |
    v
[UserSlice Reducer]
    |
    v
[Updated State]
    |
    v
[Logout Component] <----> [useSelector(selectUser)]
    |
    v
[dispatch(logout())]
    |
    v
[Redux Store]
    |
    v
[UserSlice Reducer]
    |
    v
[Cleared State]
```

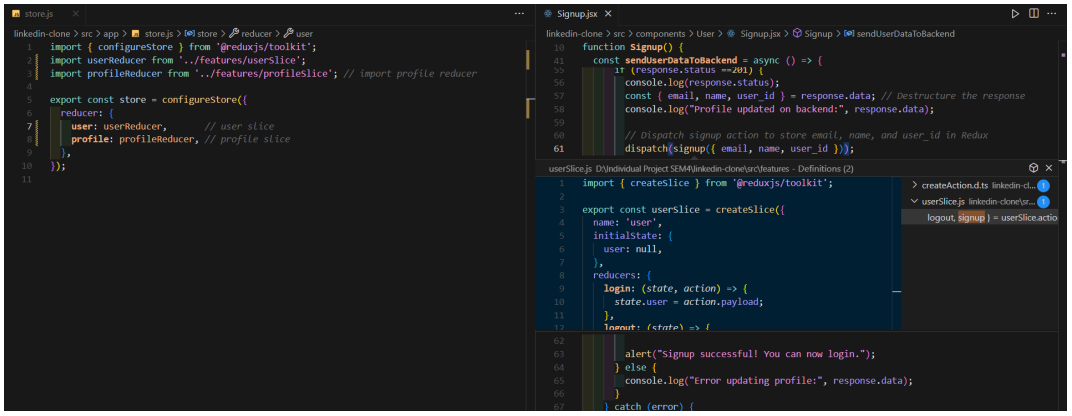
Explanation of Flow

Khushal Savalakha

1. **Login Process**:
    - The **Login Component** collects user data and dispatches the `login` action with that data.
    - The **Redux Store** receives this action and updates the state through the **UserSlice Reducer**.
    - The state is updated to include user data.
2. **Logout Process**:
    - The **Logout Component** dispatches the `logout` action when the user clicks the logout button.
    - The **Redux Store** processes this action and clears the state through the **UserSlice Reducer**.
    - The state is updated to remove user data.

Conclusion

By understanding these concepts and seeing how they interact, you can apply Redux to manage complex state logic in your applications. Redux's predictable state management helps maintain and debug the state of applications as they grow in complexity.



Khushal Savalakha

1]here in UserSlice we are importing createSlice in which we are naming state name which is user after that we are defining initial value of state user which is null and then we are defining reducers this is like functions of REACT reducers but some different here log in and log out automatically got state and action through dispatch(
login({
email,
name,
password,
})
); and2] here you have to add simple explanation what is the need of store why we just use it in index.js 3]here instead of sending type to reducer in UserSlice.jsx we are just sending dispatch(
login({
email,
name,
password,
})
); by which they find login or logout function present inside the reducer object from createSlice 4] here you have to explain simply and depthly how export default userSlice.reducer; can be used in the store configuration. The reducer function is responsible for handling the state updates defined in the reducers.

am i true for every points represents the flow except questions(which you have to answer) ?

Yes, you have a good grasp of the overall flow in Redux with your points. Let me clarify each of your questions in detail:

1. UserSlice Explanation

What is happening:

- `createSlice`: This function from Redux Toolkit simplifies the process of creating a slice of the Redux store. It combines reducers and actions in one place.
- **State Name**: You name your slice (`user` in this case). This name is used to identify this part of the state.

Khushal Savalakha

- **Initial State**: The `initialState` defines the starting state of this slice. Here, it's set to `{ user: null }`.
- **Reducers**: Reducers are functions that handle state updates. In Redux, these functions specify how the state changes in response to actions.

How it works with `dispatch`:

- When you call `dispatch(login({ email, name, password }))`, it triggers the `login` reducer. The action dispatched contains the payload `{ email, name, password }`.
- `createSlice` automatically generates action creators and action types based on the reducer functions you define (`login` and `logout` here). So, you don't need to manually create action types or action creators.

2. Need for the Store

What is the Store?

- The **store** holds the entire state of your application. It's a central place where all the state updates happen.

Why use it in `index.js`?

- The **store** is configured and created in `Store.jsx` and then passed to the `Provider` component in `index.js`. This makes the Redux store available throughout your application.
- `index.js` is the entry point of your React application. Wrapping the `App` component with `Provider` ensures that any component within `App` can access the Redux store via React-Redux hooks like `useDispatch` and `useSelector`.

3. Sending Actions with `dispatch`

What happens with `dispatch`?

- When you use `dispatch(login({ email, name, password }))`, you're dispatching an action to the store.
- `dispatch` looks up the action type (`login` in this case) and passes it to the appropriate reducer function defined in `createSlice`.
- The `login` reducer function updates the state based on the action's payload.

Why not use action types directly?

- `createSlice` simplifies action creation. Instead of manually defining action types and action creators, you directly use the action creators (`login` and `logout`) generated by

Khushal Savalakha

`createSlice`.

4. Exporting Reducer

Why `export default userSlice.reducer;`?

- The `userSlice.reducer` function is the reducer that handles state changes for the `user` slice of your Redux state.
- When you export `userSlice.reducer` from `UserSlice.jsx`, you're exporting the function that will process actions related to user state.

How it's used in store configuration?

- In `Store.jsx`, you import this reducer and pass it to the `configureStore` function as part of the store configuration.
- `configureStore` requires a `reducer` parameter where you define how different parts of the state should be handled. Here's an example:

Store Configuration Example:

```
// Store.jsx
import { configureStore } from '@reduxjs/toolkit';
import userReducer from '../features/UserSlice';

export default configureStore({
  reducer: {
    user: userReducer,  // This sets up the user slice o
f the state with the userReducer
  },
});
```

In summary:

- The **store** manages the overall application state.
- `dispatch` sends actions to the store, where reducers handle state changes.
- `createSlice` automates action and reducer creation.
- `export default userSlice.reducer;` provides the reducer function for configuring the store.

With this setup, your application can easily manage and update its state using Redux in a scalable and maintainable way.

Khushal Savalakria

💡 here how cinfigureStore and and UserSlice.reducer connected what is the work of configure store is it like here it means first of all in Login.jsx we are making path to Store.jsx which is provided in index.js when dispatch(login({})) executes it will send that actions to Store.jsx and it will make changes directly to state am i true if not then explain?

You've got the right idea about how `configureStore` and `UserSlice.reducer` are connected and how the Redux flow works. Here's a detailed breakdown of how these components interact:

1. Understanding `configureStore`

Purpose of `configureStore`:

- `configureStore` is a function provided by Redux Toolkit that simplifies the setup of the Redux store. It helps in configuring the store with the necessary reducers and middleware.

How it works:

- When you call `configureStore`, you pass an object where the `reducer` key specifies how to handle different parts of your state.
- **Reducers** define how actions transform the state. You can think of them as functions that accept the current state and an action and return a new state.

Example of `configureStore` in `Store.jsx`:

```
// Store.jsx
import { configureStore } from '@reduxjs/toolkit';
import userReducer from '../features/UserSlice';  // Imp
ort the user slice reducer

export default configureStore({
  reducer: {
    user: userReducer,  // Connect the user slice reduce
r to the Redux store
  },
});
```


Khushal Savalakha

In this example:

- `userReducer` is the reducer function from `UserSlice`.
- By passing `{ user: userReducer }` to `configureStore`, you are telling Redux how to handle the `user` slice of the state using `userReducer`.

2. Connection to `UserSlice.reducer`

Role of `UserSlice.reducer`:

- `UserSlice.reducer` is the reducer function generated by `createSlice` for managing the `user` slice of the state.
- It processes actions related to the `user` state (like `login` and `logout`) and updates the state accordingly.

How it's used in the store:

- In `Store.jsx`, `userReducer` (which is `UserSlice.reducer`) is passed to `configureStore`.
- This tells Redux that the `user` part of the state should be managed by the `userReducer` function.

3. Flow of Actions and State Updates

Flow of data and actions:

1. **User Action Triggered:**
   - In `Login.jsx`, when a user logs in, you dispatch an action with `dispatch(login({ email, name, password }))`.
2. **Dispatch to Store:**
   - This action is sent to the Redux store. The `dispatch` function is used to send actions to the store.
3. **Store Receives Action:**
   - The Redux store, configured with `userReducer`, receives this action.
4. **Reducer Handles Action:**
   - `userReducer` (from `UserSlice.reducer`) processes the action. It updates the state based on the action's type and payload.
   - For example, the `login` reducer might update the `user` state with the new user information.

Khushal Savalakha

5. **State Update:**
    - After the reducer processes the action, the store's state is updated.
6. **React Components Re-render:**
    - Components that are connected to the Redux store (e.g., using `useSelector` in `Logout.jsx`) will automatically re-render with the new state.

Summary of Your Points:

- `configureStore`: Sets up the Redux store with reducers and middleware.
- `UserSlice.reducer`: Defines how the `user` slice of the state is updated.
- `dispatch`: Sends actions to the store to update the state.
- **Store Configuration**: Ensures that when actions are dispatched, the appropriate reducer handles them and updates the state accordingly.

Visualization of the Flow:

Here's a simplified flowchart to visualize how everything connects:

1. `Login.jsx`
    - User triggers login action
    - `dispatch(login({ email, name, password }))`
2. `index.js`
    - Sets up the Redux `Provider` with the configured store
    - Passes the store to the React app
3. `Store.jsx`
    - Configures the store
    - Connects `userReducer` to the store
4. `UserSlice.jsx`
    - Defines the `userReducer` with actions and state management logic
5. **Redux Store**
    - Receives the action from `dispatch`
    - `userReducer` processes the action and updates the state
6. `Logout.jsx`

Khushal Savalakha

- Uses `useSelector` to get the updated user state
- Renders based on the updated state

This flow ensures that any action dispatched (like login) updates the store's state, and components connected to the store (like `Logout`) reflect these updates automatically.

> 💡 then here in above code what is the role of Higher Order Components in above codes like Provider?

In the Redux setup, `Provider` is an example of a Higher-Order Component (HOC). Let's break down the role and importance of `Provider` in the context of Redux:

What is a Higher-Order Component (HOC)?

A Higher-Order Component is a function that takes a component and returns a new component with additional props or functionality. HOCs are used to enhance or extend the behavior of components without modifying their code directly.

Role of `Provider` in Redux

1. Providing the Redux Store:

- **Purpose:** The `Provider` component makes the Redux store available to any nested components that need access to it.
- **How it Works:** It takes the Redux store as a prop and uses React's context API to pass this store down the component tree. This allows any component in the tree to connect to the store and access or modify the state.

2. Ensuring Global State Access:

- **Purpose:** It ensures that all components within the `Provider` can connect to the Redux store and interact with the global state.
- **How it Works:** By wrapping the root component of your application (e.g., `<App />`) with the `Provider`, any child component (e.g., `<Login />`, `<Logout />`) can access the store and use Redux functionalities.

How `Provider` Works in the Code

Here's a step-by-step explanation using your code example:

1. Setting Up the Store:

Khushal Savalakha

- **Code Example:** `Store.jsx`

```
import { configureStore } from '@reduxjs/toolkit';
import userReducer from '../features/UserSlice';  //
Import the user slice reducer

export default configureStore({
  reducer: {
    user: userReducer,  // Connect the user slice red
ucer to the Redux store
  },
});
```

- **Explanation:** The Redux store is created and configured with the `userReducer` that handles the `user` slice of the state.

2. Wrapping the Application with `Provider`:

- **Code Example:** `index.jsx`

```
import React from 'react';
import ReactDOM from 'react-dom/client';
import './index.css';
import App from './App';
import { Provider } from 'react-redux';  // Import Pr
ovider
import store from './app/Store';  // Import the confi
gured store

const root = ReactDOM.createRoot(document.getElementB
yId('root'));
root.render(
  <Provider store={store}>  {/* Wrap the App componen
t with Provider */}
    <App />
  </Provider>
);
```


Khushal Savalakha

- **Explanation:** The `Provider` component wraps the `<App />` component, passing the `store` as a prop. This setup ensures that all components within `<App />` (including `<Login />`, `<Logout />`, etc.) can access the Redux store.

3. Using Redux in Components:

- **Code Example:** `Login.jsx`

```
import React, { useState } from 'react';
import { useDispatch } from 'react-redux';
import { login } from '../features/UserSlice';

export default function Login() {
  const [email, setEmail] = useState('');
  const [name, setName] = useState('');
  const [password, setPassword] = useState('');
  const dispatch = useDispatch();

  const handleLogin = (e) => {
    e.preventDefault();
    dispatch(login({ email, name, password }));
  };

  return (
    <form onSubmit={handleLogin}>
      <input
        type="email"
        value={email}
        onChange={(e) => setEmail(e.target.value)}
        placeholder="Email"
      />
      <input
        type="text"
        value={name}
        onChange={(e) => setName(e.target.value)}
        placeholder="Name"
      />
      <input
        type="password"
        value={password}
```


Khushal Savalakha

```
          onChange={(e) => setPassword(e.target.value)}
          placeholder="Password"
        />
        <button type="submit">Login</button>
      </form>
    );
  }
```

- **Explanation:** The `Login` component uses `useDispatch` to dispatch actions to the Redux store. Since the component is a child of `<Provider>`, it can access the Redux store and dispatch actions like `login`.

Summary

- `Provider` is a Higher-Order Component that connects the Redux store to your React application.
- **Purpose:** It ensures that any component within the `Provider` can access the Redux store using `useSelector`, `useDispatch`, or `connect`.
- **Usage:** Wrap the root component of your application with `Provider` and pass the store to it.

Visual Flow:

1. **Create Store:** Define the Redux store and reducers.
2. **Wrap with Provider:** Use `Provider` to wrap your root component, passing the store.
3. **Connect Components:** Components can now access and interact with the store through `useSelector`, `useDispatch`, or `connect`.

This setup ensures a clean and maintainable way to manage global state and interact with it across your React application.

↪ **How it Works:** It takes the Redux store as a prop and uses React's context API to pass this store down the component tree. This allows any component in the tree to connect to the store and access or modify the state.

if it is just like props then why we have to import too many things like useSelector and etc why we can not directly use it ?

The Redux `Provider` uses React's context API to pass the Redux store down the component tree. While it might seem similar to props, there are specific reasons why we

Khushal Savalakha

need to use Redux hooks like `useSelector` and `useDispatch` rather than just accessing the store directly.

Why Not Use the Store Directly?

1. **Encapsulation and Separation of Concerns:**
   - **Encapsulation:** The `Provider` ensures that the Redux store is encapsulated within the React context. Components do not need to know the details of how the store is created or how it works internally. They simply use hooks to interact with the store.
   - **Separation of Concerns:** By using hooks like `useSelector` and `useDispatch`, we separate the concerns of component logic (e.g., data fetching, user interactions) from the Redux store's internal management.
2. **React-Redux Hooks:**
   - `useSelector`**:** This hook allows components to access specific parts of the Redux store’s state. It efficiently subscribes to only the parts of the state that the component needs, causing re-renders only when those parts change.
   - `useDispatch`**:** This hook provides a way to dispatch actions to the Redux store. It simplifies the process of sending actions without having to manually handle action types or dispatch functions.
3. **Optimized Performance:**
   - **Selective Updates:** `useSelector` helps in optimizing performance by subscribing only to the necessary pieces of state. This minimizes unnecessary re-renders and ensures that components only re-render when the specific state they depend on changes.
   - **Batched Updates:** Redux hooks and React’s context API are optimized to handle batched updates efficiently, reducing the overhead of prop drilling.
4. **Convenience and Readability:**
   - **Simpler Syntax:** Redux hooks provide a simpler and more intuitive API for interacting with the store compared to the older `connect` method. It reduces boilerplate code and makes the component logic more readable.
   - **Direct Access:** Hooks like `useSelector` and `useDispatch` provide direct access to the state and dispatch functions, making it easier to understand and use them within functional components.

Khushal Savalakha

Example Explanation

Here's a more detailed look at why we use these hooks:

1. `useSelector`:

- **Purpose:** To read data from the Redux store.
- **Usage:**

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

const user = useSelector((state) => state.user);
```

- **Explanation:** `useSelector` extracts the relevant part of the state from the Redux store and subscribes to changes. It updates the component when the specific state it subscribes to changes.

2. `useDispatch`:

- **Purpose:** To dispatch actions to the Redux store.
- **Usage:**

```
import { useDispatch } from 'react-redux';
import { login } from './features/UserSlice';

const dispatch = useDispatch();

const handleLogin = () => {
  dispatch(login({ email, name, password }));
};
```

- **Explanation:** `useDispatch` provides a reference to the `dispatch` function, allowing components to send actions to the Redux store. It simplifies dispatching actions and does not require manual handling of action types or dispatch functions.

Visual Flow:

1. **Provider Setup:**
   - The `Provider` component wraps your root component, passing the Redux store to the React context.
2. **Component Access:**

Khushal Savalakha

- Components use `useSelector` to access specific parts of the state and `useDispatch` to dispatch actions.
- **Example:**
  - `Login` component uses `useDispatch` to send login data.
  - `Logout` component uses `useSelector` to access the current user state.

3. **State Updates:**
   - When actions are dispatched (e.g., `login`), the Redux store updates the state.
   - Components that use `useSelector` will automatically re-render if the relevant state changes.

By using Redux hooks and the `Provider`, you maintain a clear and efficient way to manage state across your application, enhancing both performance and readability.

> 💡 it means true that Provider is high order component?

Yes, you are correct. The `Provider` component from React-Redux is a Higher-Order Component (HOC). Here's a more detailed explanation:

What is a Higher-Order Component (HOC)?

A Higher-Order Component (HOC) is a pattern in React that involves a function taking a component and returning a new component with enhanced functionality. HOCs are used to share common functionality between components without repeating code.

Provider as a Higher-Order Component

1. **Purpose:**
   - **Context Provider:** The `Provider` component makes the Redux store available to any nested components that need access to it via the React context API.
2. **How it Works:**
   - **Wraps Application:** When you wrap your application (or part of it) with `Provider`, it provides the Redux store to all components within the context.
   - **Context API:** It uses React's context API to pass the Redux store down the component tree, allowing components to connect to the Redux store without manually passing props.

Khushal Savalakria

3. **Example Usage:**

```
import { Provider } from 'react-redux';
import store from './store';  // Redux store configur
ation

function App() {
  return (
    <Provider store={store}>
      <YourComponent />
    </Provider>
  );
}

export default App;
```

Why is `Provider` an HOC?

- **Enhancement:** `Provider` enhances the components within its tree by giving them access to the Redux store.
- **Composition:** It takes the `store` as a prop and returns a new component that injects this store into the React context, making it available for `useSelector` and `useDispatch` hooks.

Why Use `Provider`?

- **Centralized State Management:** It centralizes the management of state, making it accessible to any component in the tree without prop drilling.
- **Declarative Data Flow:** It simplifies the process of connecting components to the store and makes the data flow in the application more declarative.

Visual Flow:

1. **Setup:**
   - You configure the Redux store.
   - You wrap your root component (or parts of your application) with the `Provider` component.
2. **Context Creation:**

Khushal Savalakha

- The `Provider` component creates a context that holds the Redux store.

3. **Access Store:**
   - Any nested component can access the Redux store using the `useSelector` and `useDispatch` hooks.

Example Diagram:

```
App
 └── Provider (store={redux store})
      └── ComponentTree
           ├── ComponentA (uses useSelector)
           └── ComponentB (uses useDispatch)
```

In summary, the `Provider` is a Higher-Order Component that facilitates the integration of Redux with React by making the store available throughout your component tree.

↪ A Higher-Order Component (HOC) is a pattern in React that involves a function taking a component and returning a new component with enhanced functionality. HOCs are used to share common functionality between components without repeating code.

here Provider component is higher order component which takeing App Component then what is returning ?

In React, the `Provider` component is indeed a Higher-Order Component (HOC). Here's how it works:

What the `Provider` Component Returns

1. **Takes:**
   - **Component:** The `Provider` component takes a component (e.g., `App`) and the `store` as props.
2. **Returns:**
   - **Enhanced Component:** The `Provider` returns a new component that is wrapped with the ability to pass down the Redux store through React's context API.

Detailed Explanation

- **Function of `Provider`:**

Khushal Savalakha

- **Wraps Components:** The `Provider` component wraps the `App` (or any other component) and provides the Redux store to all the components nested within it.
- **Context Creation:** Internally, `Provider` creates a React context and stores the Redux store in it. This context is then used by the React-Redux library to make the store available to any component in the tree that needs it.

- **Result:**
  - The wrapped component tree, which is effectively the `App` component and all its descendants, is now connected to the Redux store. Components within this tree can access the store using hooks like `useSelector` and `useDispatch`.

Code Example

Here's how it works in practice:

```
import { Provider } from 'react-redux';
import store from './store';  // Redux store configurati
on
import App from './App';

function Root() {
  return (
    <Provider store={store}>
      <App />  // App is wrapped and will have access to
the Redux store
    </Provider>
  );
}

export default Root;
```

In this example:

1. `Provider` **Takes:** It takes the `App` component and the `store` as props.
2. `Provider` **Returns:** It returns a new component that wraps the `App` with the Redux store context. The `App` component and all its children now have access to the Redux store.

Visualization:

Khushal Savalakha

```
Provider
 └── App
      └── ComponentTree
           ├── ComponentA (can access Redux store via us
eSelector)
           └── ComponentB (can dispatch actions via useD
ispatch)
```

Summary:

- **Input to `Provider`:** The `Provider` takes the `App` component and `store`.
- **Output of `Provider`:** It returns a new component that wraps the `App` component with Redux store context, making the store available to the entire component tree.

The `Provider` component does not modify the `App` directly but rather provides it with a context in which it can access the Redux store.

▼ Question Mark

💡 {user ? <Logout /> : <Login />}
how to use question mark in react very effectively.

In React, the question mark (`?`) is commonly used as part of the **ternary operator**, which is a concise way to handle conditional rendering. The basic syntax is:

```
condition ? expressionIfTrue : expressionIfFalse
```

This operator can be useful when you want to render different components or content based on a condition.

Example 1: Conditional Rendering (Login/Logout)

In your example:

```
{user ? <Logout /> : <Login />}
```


Khushal Savalakha

This checks if the `user` variable is truthy (i.e., the user is logged in). If it is, it renders the `<Logout />` component. If `user` is falsy (i.e., the user is not logged in), it renders the `<Login />` component.

When to Use the Ternary Operator Effectively:

1. **Simple Conditional Rendering:** Use the ternary operator when you want to switch between two elements based on a condition.

```
{isLoggedIn ? <WelcomeUser /> : <LoginPrompt />}
```

2. **Short Conditional Statements:** The ternary operator is great for simple conditions that are short and readable.

```
{cartItems.length > 0 ? <Cart /> : <EmptyCartMessage
/>}
```

3. **Avoid Long Ternary Statements:** For more complex logic, it's better to use an `if` statement or break down the condition to improve readability.

Example 2: Nested Ternary (Use Sparingly)

You can nest ternary operators, but it's recommended to keep it simple to avoid readability issues.

```
{user ? (isAdmin ? <AdminPanel /> : <UserDashboard />) :
<Login />}
```

Here, you first check if the user is logged in, and if true, you further check if the user is an admin to decide which dashboard to render.

Example 3: Conditional Class Names

The ternary operator can also be used to conditionally apply class names.

```
<button className={isActive ? 'active-btn' : 'inactive-b
tn'}>
  Click Me
</button>
```


Khushal Savalakhia

Alternative to Ternary: Logical AND (`&&`) for Simple Conditions

If you only need to render something if a condition is true and don't need an "else" condition, you can use the logical AND (`&&`).

```
{user && <Logout />}
```

This will render the `<Logout />` component only if `user` is truthy.

Summary:

- Use the ternary operator (`? :`) for simple, two-branch conditions.
- For more complex logic, stick to traditional `if-else` statements.
- You can also use the logical AND (`&&`) operator for rendering elements based on a single condition.

Khushal Savalakha

dfghnm

```
// src/Components/Logout.jsx
import React from 'react';
import { useDispatch, useSelector } from 'react-redux';
import { logout, selectUser } from '../features/UserSlice';

export default function Logout() {
const dispatch = useDispatch();
const user = useSelector(selectUser);

const handleLogout = () => {
dispatch(logout());
};

return (
<div>
<h1>Welcome, {user?.name || 'User'}!</h1>
<p>Email: {user?.email}</p>
<p>Password: {user?.password}</p>
<button onClick={handleLogout}>Logout</button>
</div>
);
}
```

tell me here how it is checking and if null it will get then it will gives false ?

In the code snippet you provided, you're using the **optional chaining operator (`?.`)** to safely access properties of the `user` object. Let me break it down:

Key Part:

```
<h1>Welcome, {user?.name || 'User'}!</h1>
```

Explanation:

1. **Optional Chaining (`?.`)**:
   - `user?.name`: The **optional chaining** operator checks if `user` is `null` or `undefined`. If it is, the expression will short-circuit and return `undefined` instead

Khushal Savalakha

of trying to access `user.name` (which would throw an error if `user` was `null` or `undefined`).

- If `user` is **not null or undefined**, it will access `user.name`.

2. **Logical OR (`||`)**:
   - `user?.name || 'User'`: This part means if `user?.name` evaluates to `falsy` (e.g., `null`, `undefined`, `''`, `false`, etc.), the string `'User'` will be used as a fallback value.
   - If `user?.name` is a valid string, it will be displayed. Otherwise, `'User'` will be shown as a default.

How the Fallback Works:

- **If `user` is null or undefined**:
  - `user?.name` will evaluate to `undefined`.
  - The logical OR (`||`) will then fall back to `'User'`, because `undefined` is falsy.
- **If `user.name` exists**:
  - The `user?.name` will return the actual `name`, and `'User'` will not be used.

Example Scenarios:

1. **When `user` is `null` or `undefined`**:

```
Welcome, User!
```

   - Since `user?.name` is `undefined`, `'User'` is used.

2. **When `user` is an object, but `name` is missing or empty**:

```
Welcome, User!
```

   - For example, if `user = { email: 'user@example.com', password: 'password123' }`, `user?.name` is `undefined`, so `'User'` will be displayed.

3. **When `user` is an object and `name` exists**:

```
Welcome, John!
```


Khushal Savalakha

- If `user = { name: 'John', email: 'john@example.com', password: 'secret123' }`, `user?.name` will return `'John'`.

Conclusion:

The optional chaining operator (`?.`) ensures that if `user` is `null` or `undefined`, the code will not break with an error and will safely return `undefined`. The logical OR (`||`) then provides a fallback value if the value of `user?.name` is `falsy`.

▼ Qusetion Bank(QB)

```
import React from 'react';
function App() {
const handleclick = () => {
alert ('Welcome to LJU');
}
return (
<div>
<center>
<button onClick={handleclick()}>Click me</button>
</center>
</div>
);
}
export default App;
```

💡 why it does not show any alert while clicking ?

Sure, let's break down why your code isn't working as expected.

What’s Happening with Your Original Code

In your original code, you have this line:

```
<button onClick={handleclick()}>Click me</button>
```

Here’s what’s happening step-by-step:

Khushal Savalakha

1. **Immediate Function Execution**: `handleclick()` is a function call, so it is executed immediately when the component renders, not when the button is clicked. As a result, the `alert('Welcome to LJU')` is shown as soon as the component mounts, not when you press the button.
2. **Return Value of the Function Call**: The `onClick` attribute expects a function reference (i.e., a function that should be called when the click event happens). However, because you are calling the function immediately with `handleclick()`, you are actually passing the return value of that function to `onClick` (which is `undefined` in this case, since `handleclick` doesn't return anything). The result is that `onClick` doesn't have a proper function to execute.

How to Fix It

To ensure that the `handleclick` function is called only when the button is clicked, you need to pass a reference to the function, not the result of calling it. Here's how to do that:

```
<button onClick={handleclick}>Click me</button>
```

In this corrected code:

- **Function Reference**: `handleclick` without parentheses is a reference to the function itself, not the result of calling it.
- **Event Handling**: React will call `handleclick` when the button is clicked, so the `alert` will only be shown at that time.

Summary

In summary, using `handleclick()` executes the function immediately and passes the result (`undefined`) to `onClick`, whereas `handleclick` (without parentheses) simply references the function, allowing React to call it in response to the click event.

Khushal Savalakhia