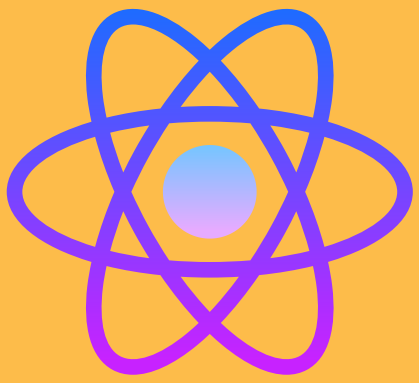


REACT INTERVIEW QUESTIONS & ANSWERS

Important React questions and answers for front-end developer technical interview

DEVELOPER UPDATES

- **What is React?**

  React is a JavaScript library that is primarily used for building user interfaces. It allows developers to create reusable UI components, making it easier to update and maintain the codebase.

  React utilizes a virtual DOM, which is a lightweight in-memory representation of the actual DOM. The virtual DOM improves the performance of the application by only updating the parts of the UI that have changed. This makes React more efficient and faster than other libraries that update the entire DOM on every change.

  One of the key features of React is its ability to manage the state of the components. This means that developers can easily keep track of the data and changes within their UI components, leading to a more efficient and streamlined development process.

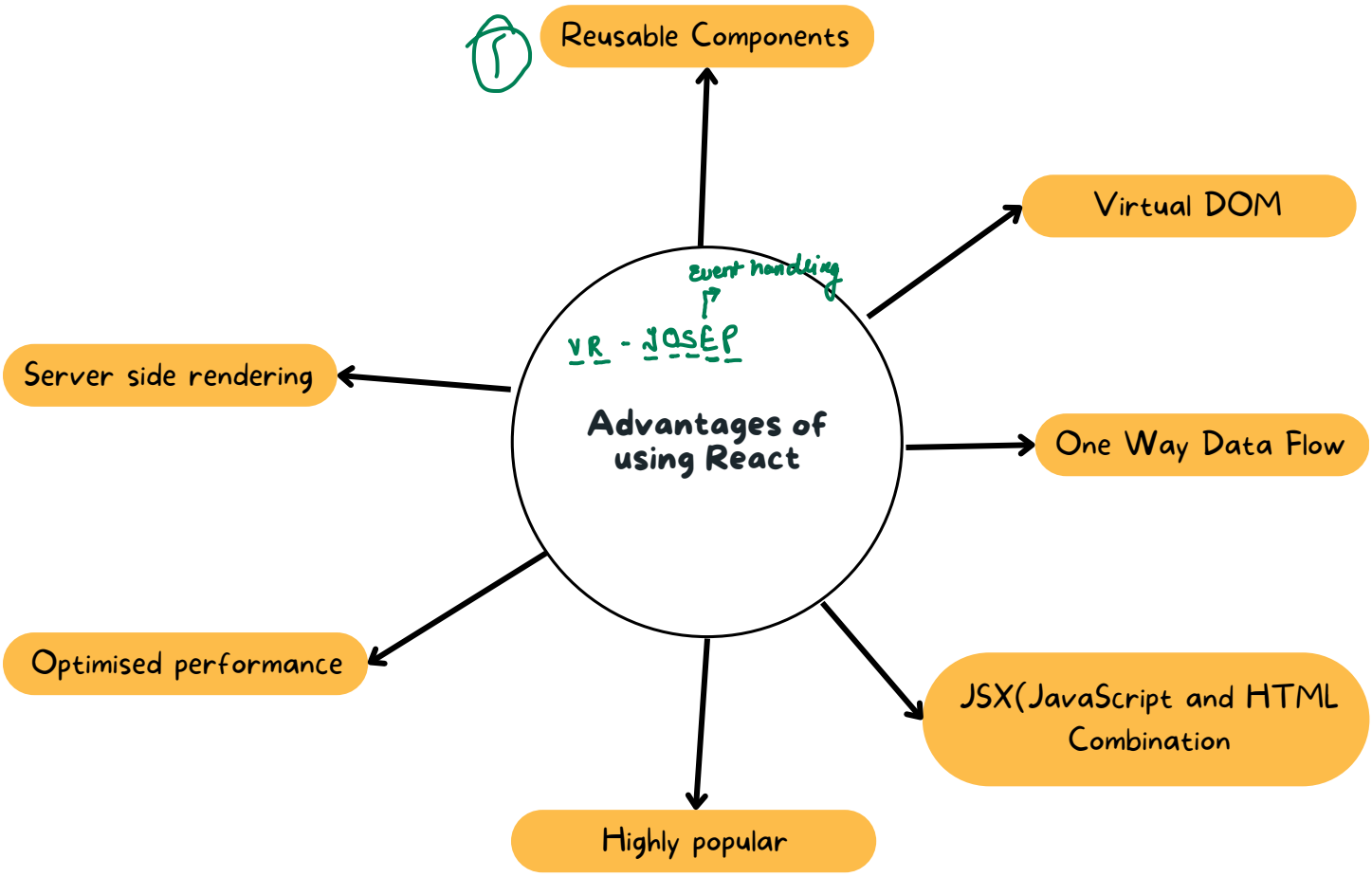
  React is also known for its use of JSX, a syntax extension that allows developers to combine JavaScript and HTML-like elements to create complex user interfaces. This makes it more intuitive for developers who are familiar with HTML and CSS to work with React.

  In addition to being used as a standalone library, React is often paired with other libraries and frameworks such as Redux for state management and React Router for client-side routing. This allows developers to create even more powerful and dynamic user interfaces.

- **What is the latest version of React?**

  React latest version is 18.2 which is release in June 2022

- **What are the advantages of using React?**



- **Components:** Make user interfaces by using small, reusable parts called components. Each component has its own set of information and actions that can be easily managed and changed.

- **JSX:** React uses JSX to make UI using JavaScript and HTML-like elements, making it easy for developers who know HTML and CSS.

- **Reusability:** React components are reusable, making it easy to maintain and scale the codebase as the application grows.

- **Virtual DOM:** React uses a virtual DOM to improve performance by only updating the parts of the UI that have changed.

- **One-way data flow:** React follows a one-way data flow where data is passed from parent component to its children making it easy to manage data in the application.

- **Server-side rendering:** React allows for server-side rendering, which can improve the performance and SEO of the application.

- **Performance Optimization:** React has built-in performance optimization features like shouldComponentUpdate and React.memo to control and avoid unnecessary re-renders.

- **Popularity**: React is popular and has a large community of developers, providing many resources and tutorials to help developers learn and solve issues.

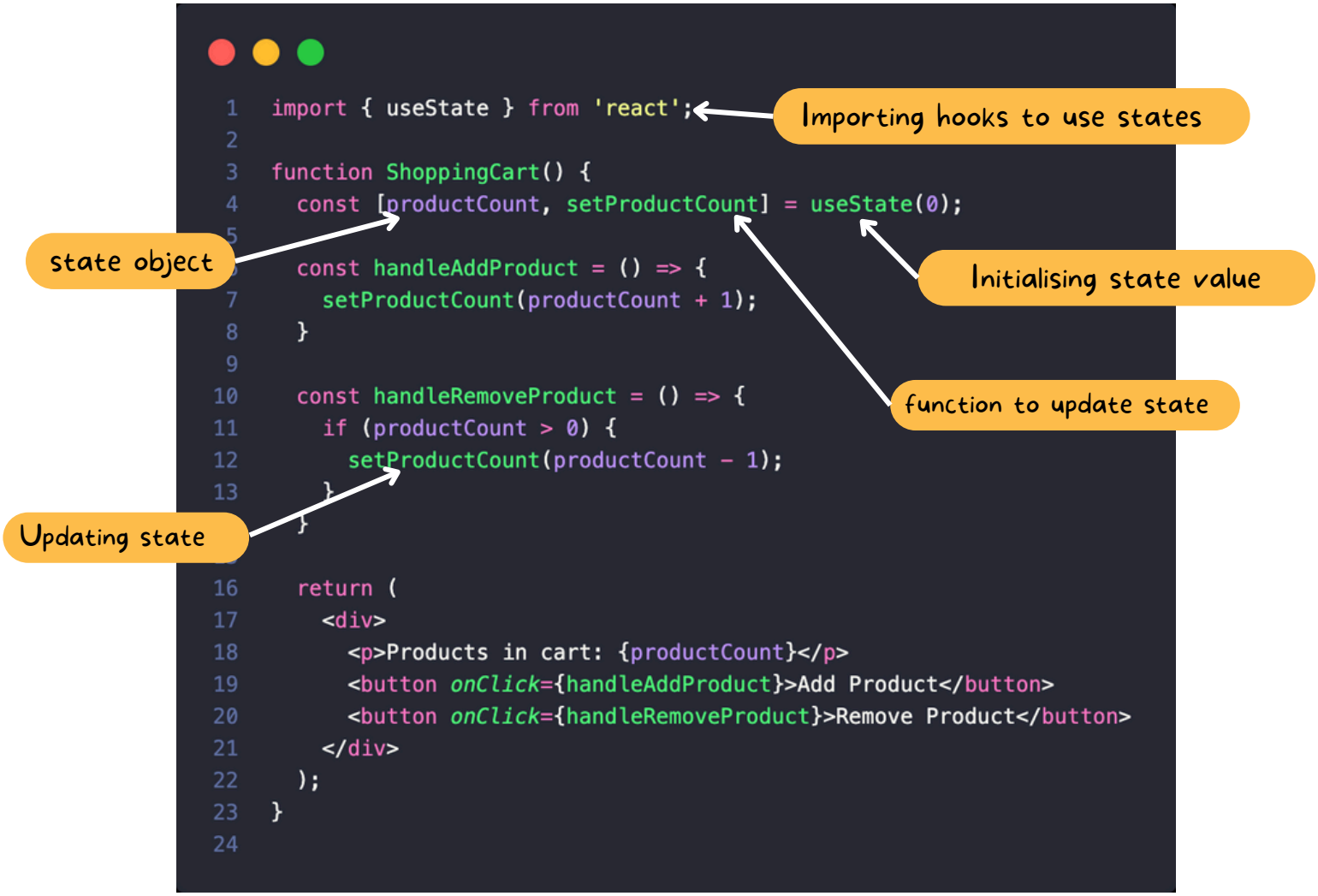
- **Event handling:** React has a consistent and intuitive way of handling events, making it easy to add interactivity to the application.

- **What is State?**

  State is a private object(You can say variable) used to store data that can change within a component, it is used to track the current state(data) of a component and can be updated by the component.

  State can be any type of data, such as a string, number, object, or array. It can also be a complex data structure, like an object containing multiple properties and nested objects.

  Let's take an example of how state can be used in simple shopping cart.



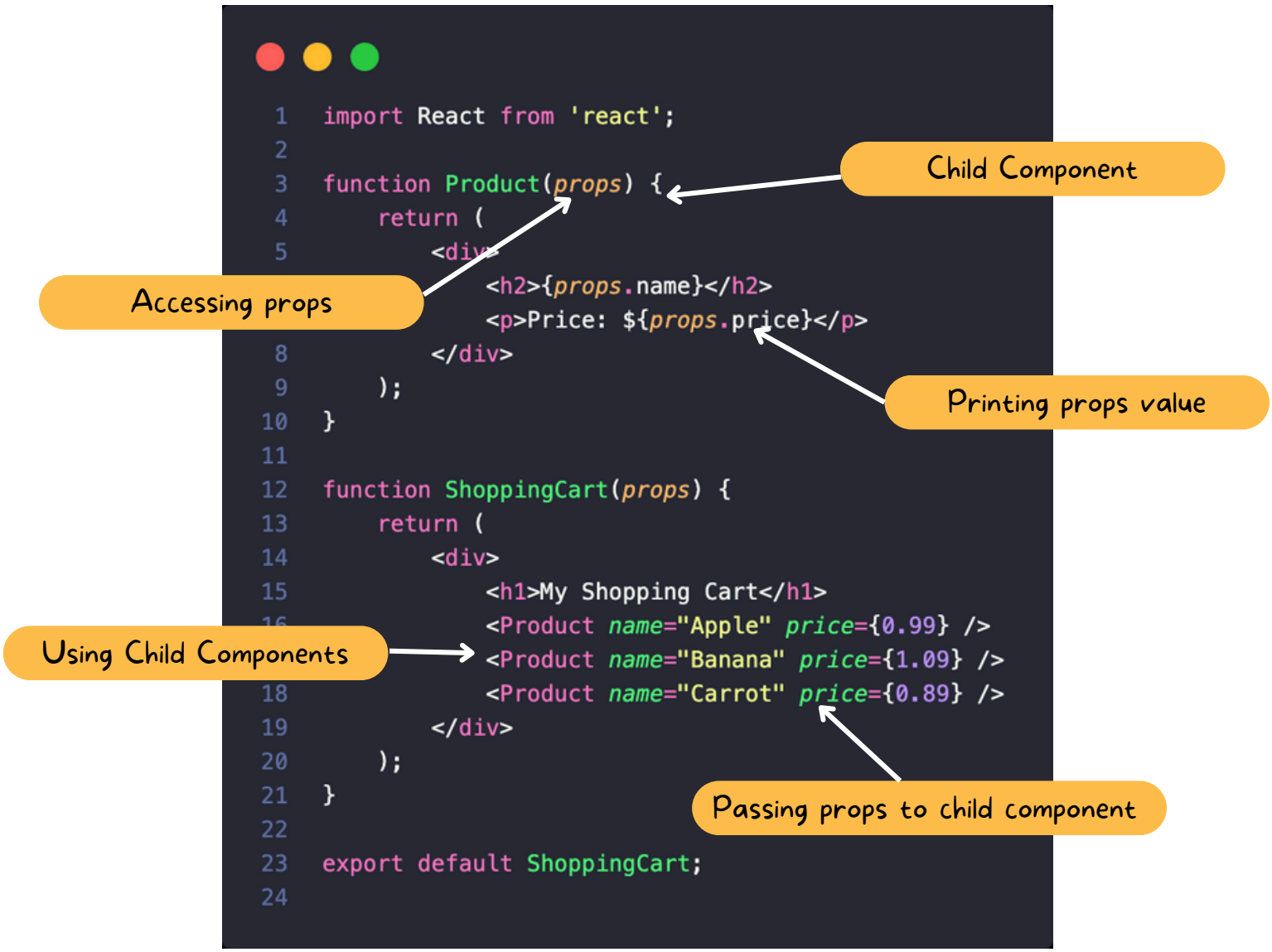
- **What are the props?**

  props stands for "properties" and refers to a way of passing data from a parent component to a child component.

Props are passed to a child component as an object and can be accessed using the function argument.

The child component cannot modify the props, but can use the values to render its own output.

Let's take an example of React Props:




```jsx
import React from 'react';

function Product(props) {
    return (
        <div>
            <h2>{props.name}</h2>
            <p>Price: ${props.price}</p>
        </div>
    );
}

function ShoppingCart(props) {
    return (
        <div>
            <h1>My Shopping Cart</h1>
            <Product name="Apple" price={0.99} />
            <Product name="Banana" price={1.09} />
            <Product name="Carrot" price={0.89} />
        </div>
    );
}

export default ShoppingCart;
```

- **What is the difference between state and props?**
  - Props are passed from a parent component to a child component, while state is managed and controlled within a single component.
  - Props are read-only and cannot be modified by the child component, while state can be updated and modified by the component itself.
  - Props are used to make a component reusable and controlled by the parent component, while state is used to manage the internal data of a component that can change over time.

- **What is virtual DOM in React?**

  The virtual DOM is a technique used in web development to improve the performance of web applications.

  It creates a lightweight copy of the actual DOM, which is the structure of the HTML elements in a web page.

  When the state of the application changes, the virtual DOM updates its copy and then calculates the most efficient way to update the actual DOM.

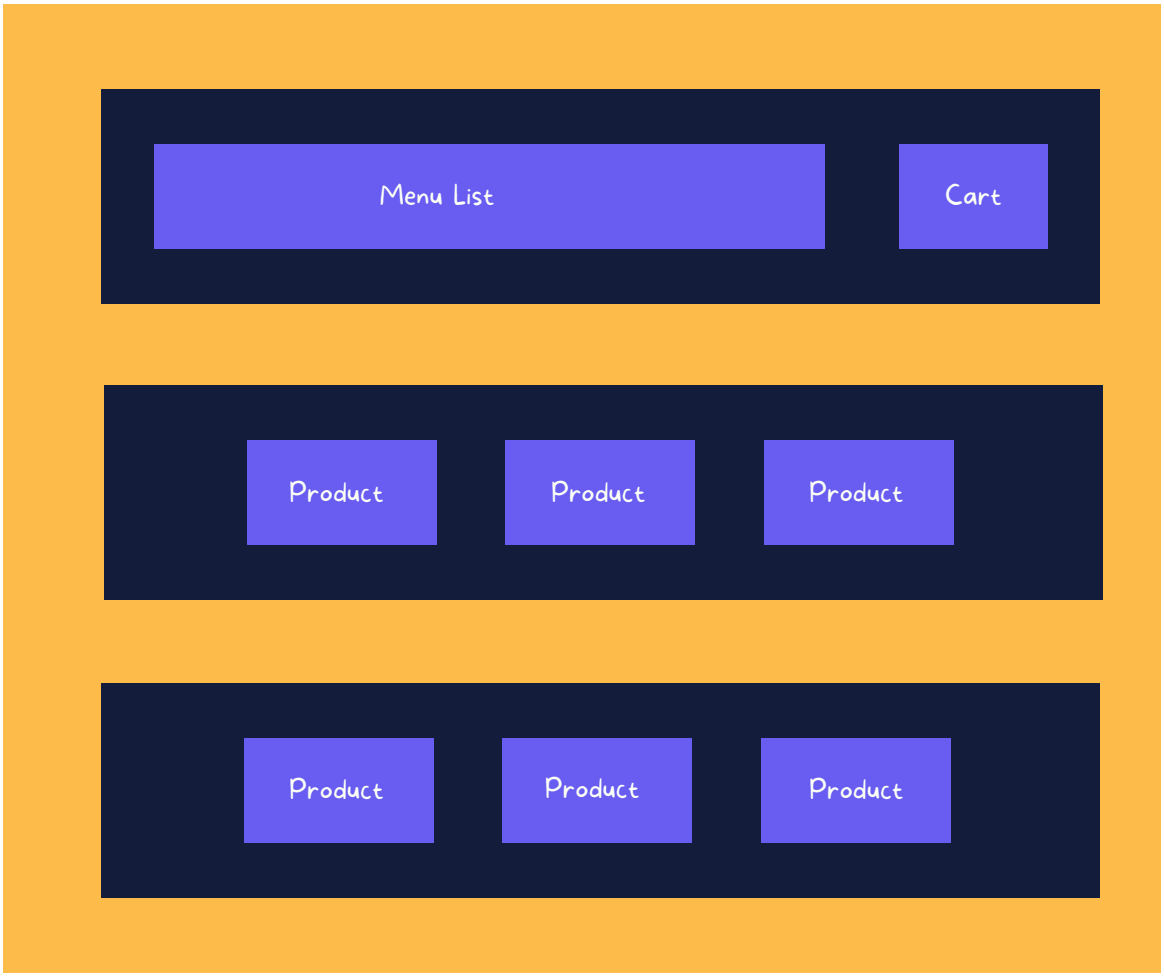
  This minimizes the number of changes that need to be made to the actual DOM.

  It acts as an intermediary between the application state and the actual DOM. When the state of the application changes, the virtual DOM updates its representation, and then compares it to the previous one.

  It then calculates the most efficient way to update the actual DOM, making only the necessary changes to the HTML elements.

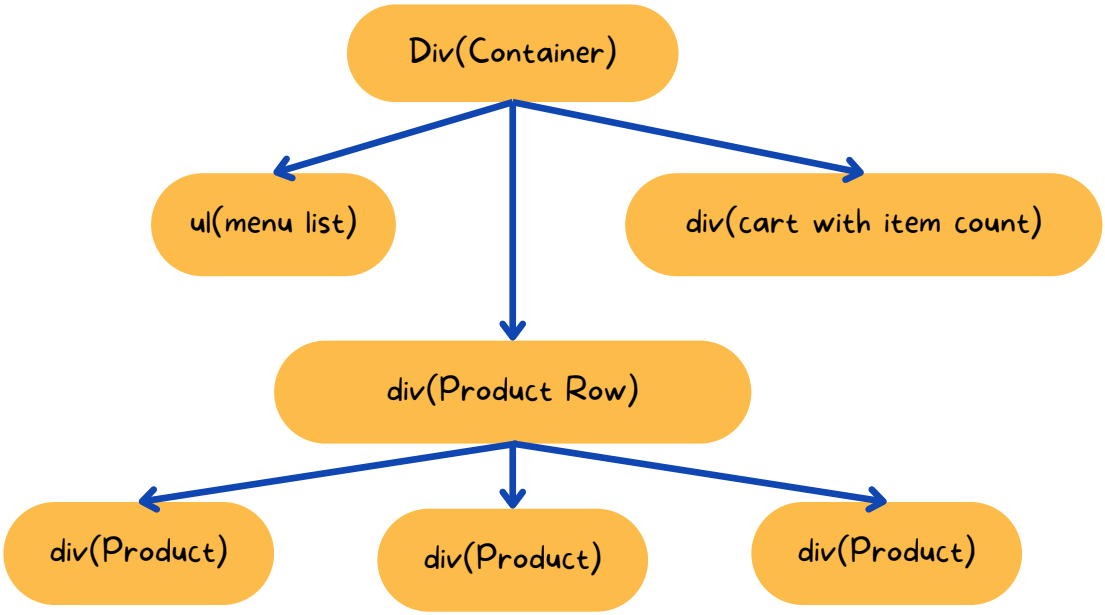
  Let's take an example,

  You are building an eCommerce app and your page design is as follows.

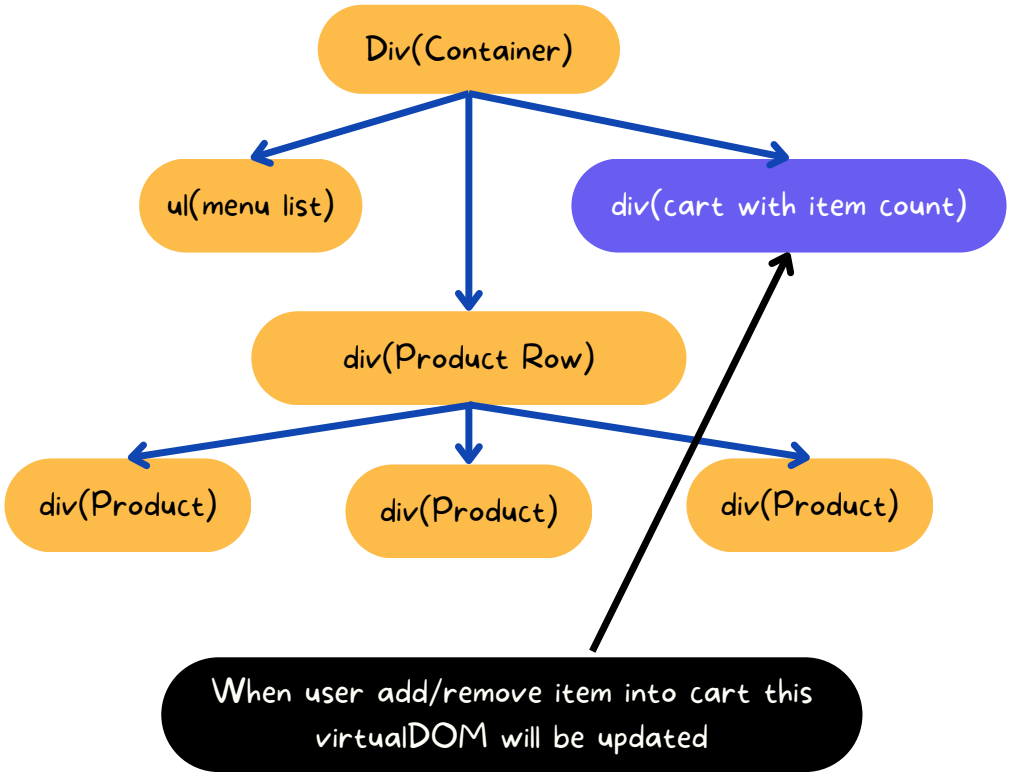


eCommerce HTML Page

Initial virtual DOM Tree will be as follows

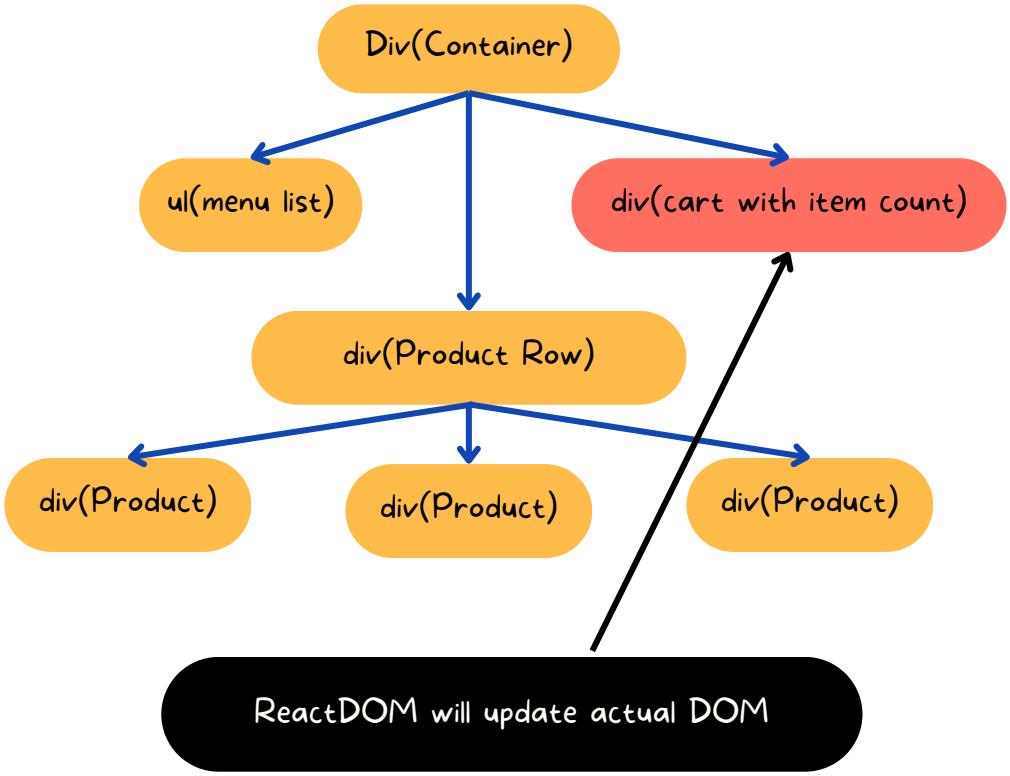


Initial Virtual DOM



Updated Virtual DOM

After comparing previous snapshot of virtual DOM ReactDOM will updated actual DOM as follows

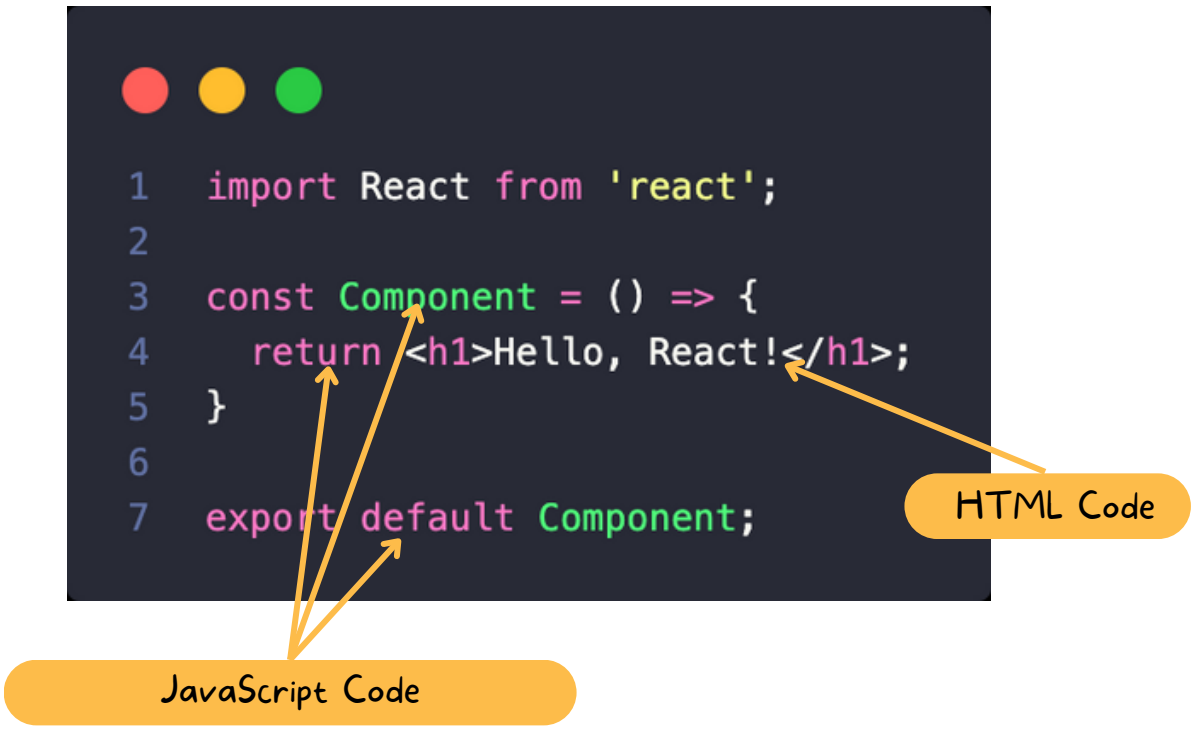


Actual DOM

- **What is JSX?**

  JSX is a way to write HTML-like code in JavaScript, particularly in React.

  It is used to define the structure and content of a component, and when the component is rendered, the JSX is converted into actual HTML elements on the browser.



- **What is the difference between stateful and stateless components in React?**

  A stateful component in React maintains its own internal state,

  while a stateless component does not have its own state and instead relies on data passed to it via props.

  Note:

  - Stateful components are also known as smart or container components
  - Stateless component are also known as dumb or presentational components.

- **Explain the difference between a class and functional components in React?**

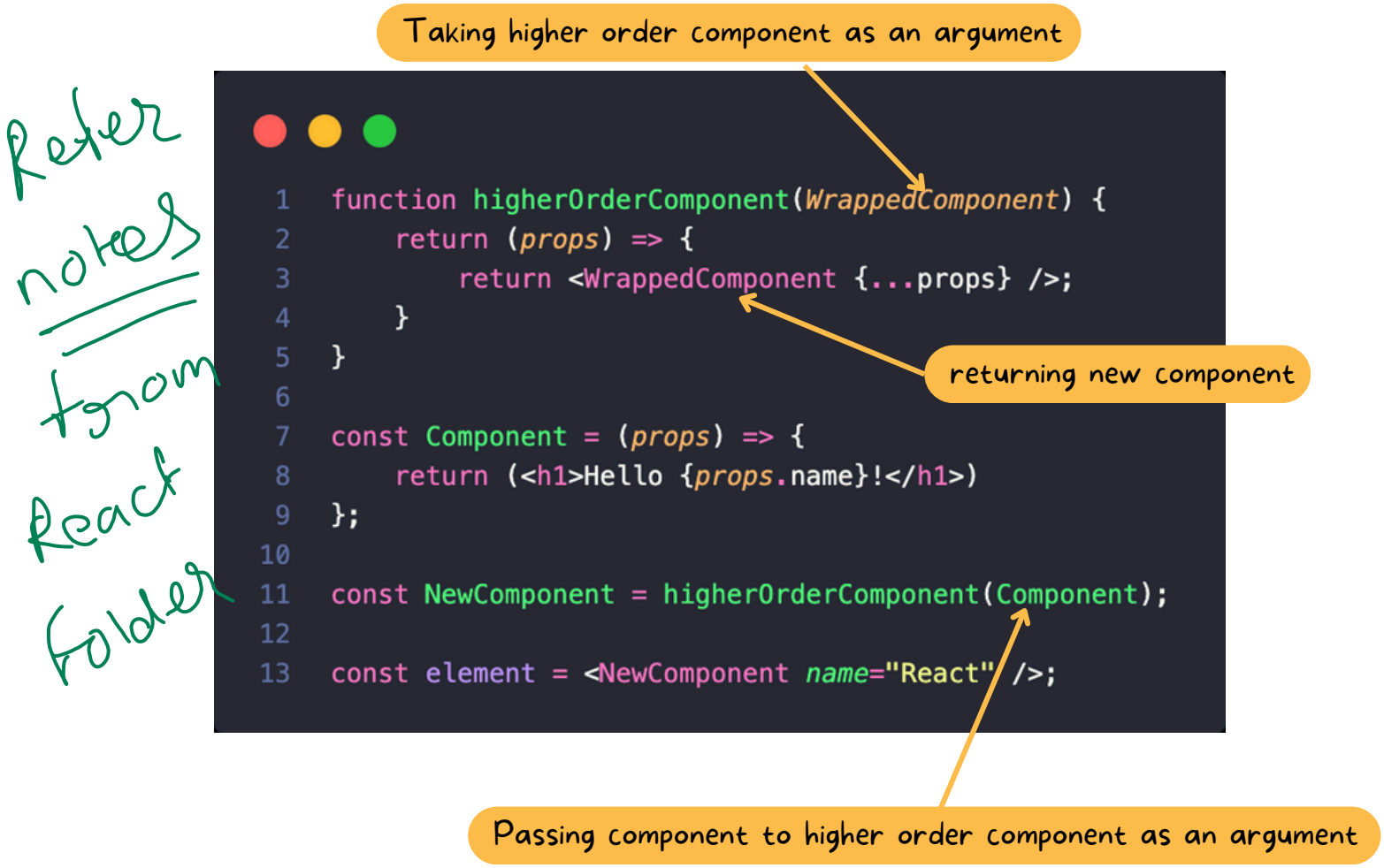
Class Component	Functional Component
Class component defined using the "class" keyword.	Functional components are defined as JavaScript functions.
Class components have a built-in state object.	Functional components do not have built-in state objects, but you can manage the state by using hooks.
Class components have built-in lifecycle methods.	Functional components do not have a built-in lifecycle method, but you can achieve similar functionality using hooks.
The "this" keyword is used to access the component instance in the class component.	Functional components do not have this keyword.
Class component must have render() to return JSX	Functional component does not need a render method, it directly returns JSX.

- **Can you explain the concept of higher-order components in React**

  A Higher Order Component (HOC) is a technique in React that enables code reuse.

  It is a function that takes an existing component as an argument(input) and returns a new component.

This new component "wraps" the original component and can add extra features, such as props or state, to the wrapped component.




```
1  function higherOrderComponent(WrappedComponent) {
2      return (props) => {
3          return <WrappedComponent {...props} />;
4      }
5  }
6
7  const Component = (props) => {
8      return (<h1>Hello {props.name}!</h1>)
9  };
10
11 const NewComponent = higherOrderComponent(Component);
12
13 const element = <NewComponent name="React" />;
```

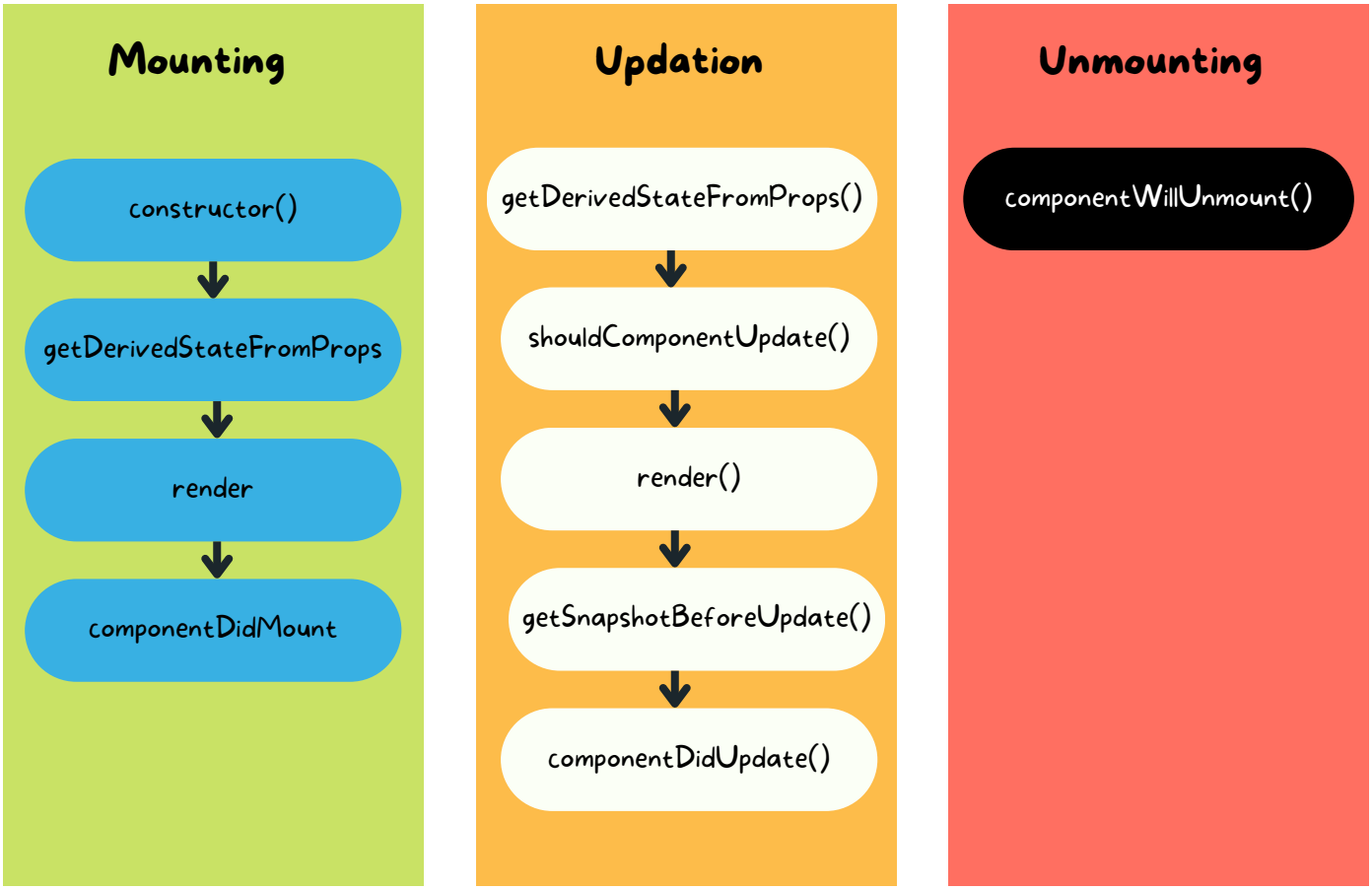
In the above example, higherOrderComponent takes a Component as an argument and returns a new component. Here higherOrderComponent is reusing the Component.

- **Explain the component lifecycle in React.**

  In React component lifecycle has 3 phases:

  1. Mounting
  2. Updating
  3. Unmounting

  Component goes through several stages of its lifecycle, starting from the moment it is created and ending with the moment it is destroyed.



1. Mounting: The Mounting stage is when a component is first added to the DOM. During this stage, the following methods are called in the following order

- **constructor():** This method is called before the component is mounted, it is used for initializing state, binding methods, and other setup tasks
- **getDerivedStateFromProps():** This method is called before the render method, it allows the component to update its internal state in response to changes in props.
- **render()**: This method is responsible for rendering the JSX that represents the component.
- **componentDidMount():** This method is called after the component has been rendered to the DOM. It is used for performing setup that requires the component to be in the DOM, such as fetching data or adding event listeners.

2. Updation: The Updating stage is when a component updates its state or props. During this stage, the following methods are called in the following order:

- **getDerivedStateFromProps():** This method is called before the render method, it allows the component to update its internal state in response to changes in props.
- **shouldComponentUpdate():** This method is called before the render method, it allows the component to decide whether or not to re-render in response to changes in props or state.
- **render()**: This method is responsible for rendering the JSX that represents the component.
- **getSnapshotBeforeUpdate():** This method is called before the component is updated in the DOM. It allows the component to capture some information from the DOM and state values.
- **componentDidUpdate():** This method is called after the component has been updated in the DOM. It is used for performing operations that require the component to be in the DOM, such as fetching data or adding event listeners.

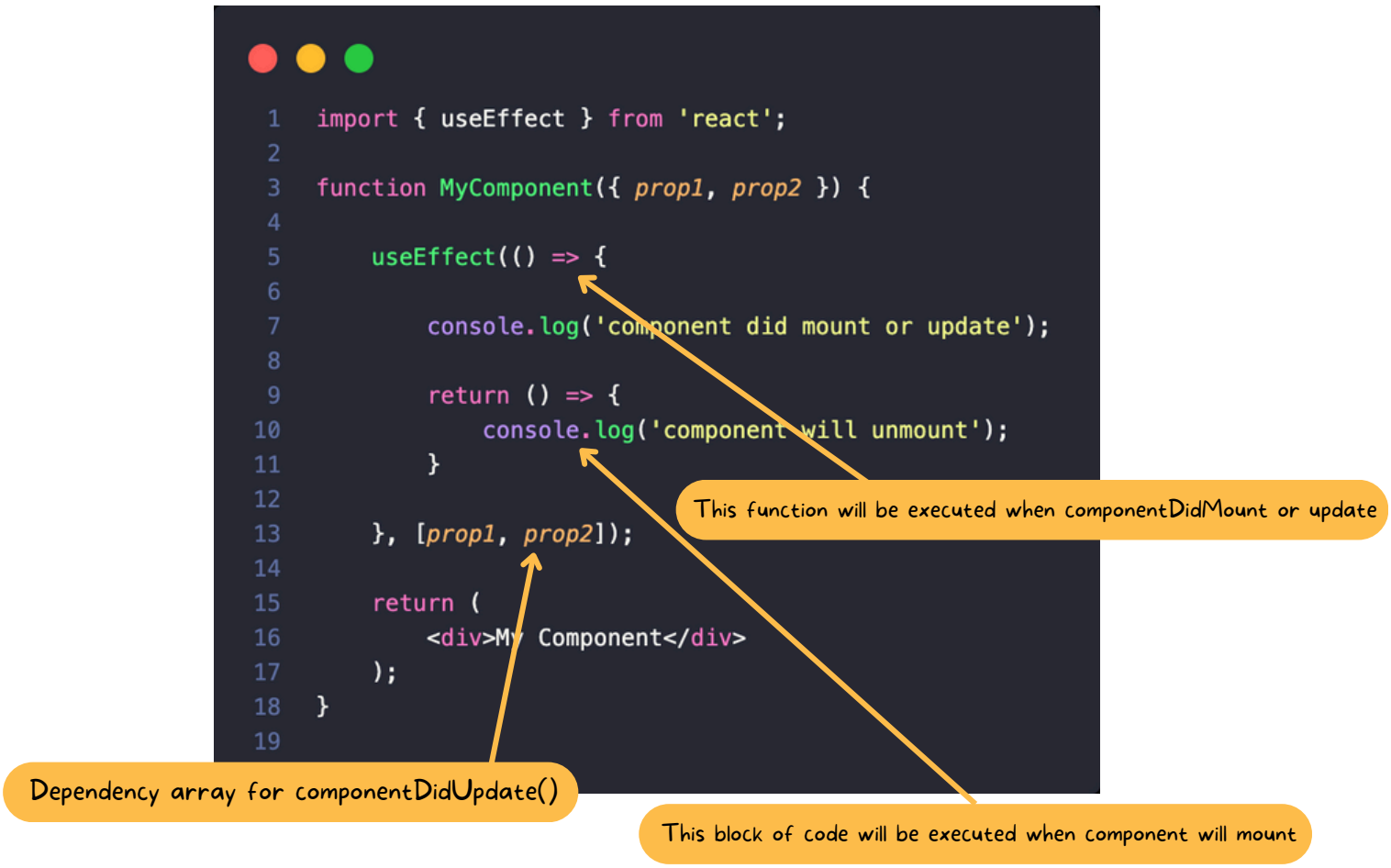
3. Unmounting: The Unmounting stage is when a component is removed from the DOM. During this stage, the following method is called:

- **componentWillUnmount()**: This method is called before the component is removed from the DOM. It is used for performing cleanup tasks, such as removing event listeners or cancelling network requests.

- **How would you implement the functionality of component lifecycle methods in functional components?**

  Here's an example of how you can use the useEffect Hook to implement the componentDidMount(), componentDidUpdate() and componentWillUnmount() lifecycle methods in a functional component.

Refer vs notes



If the dependency array is left empty "[]", the function will only execute once, similar to the behavior of the componentDidMount() method.

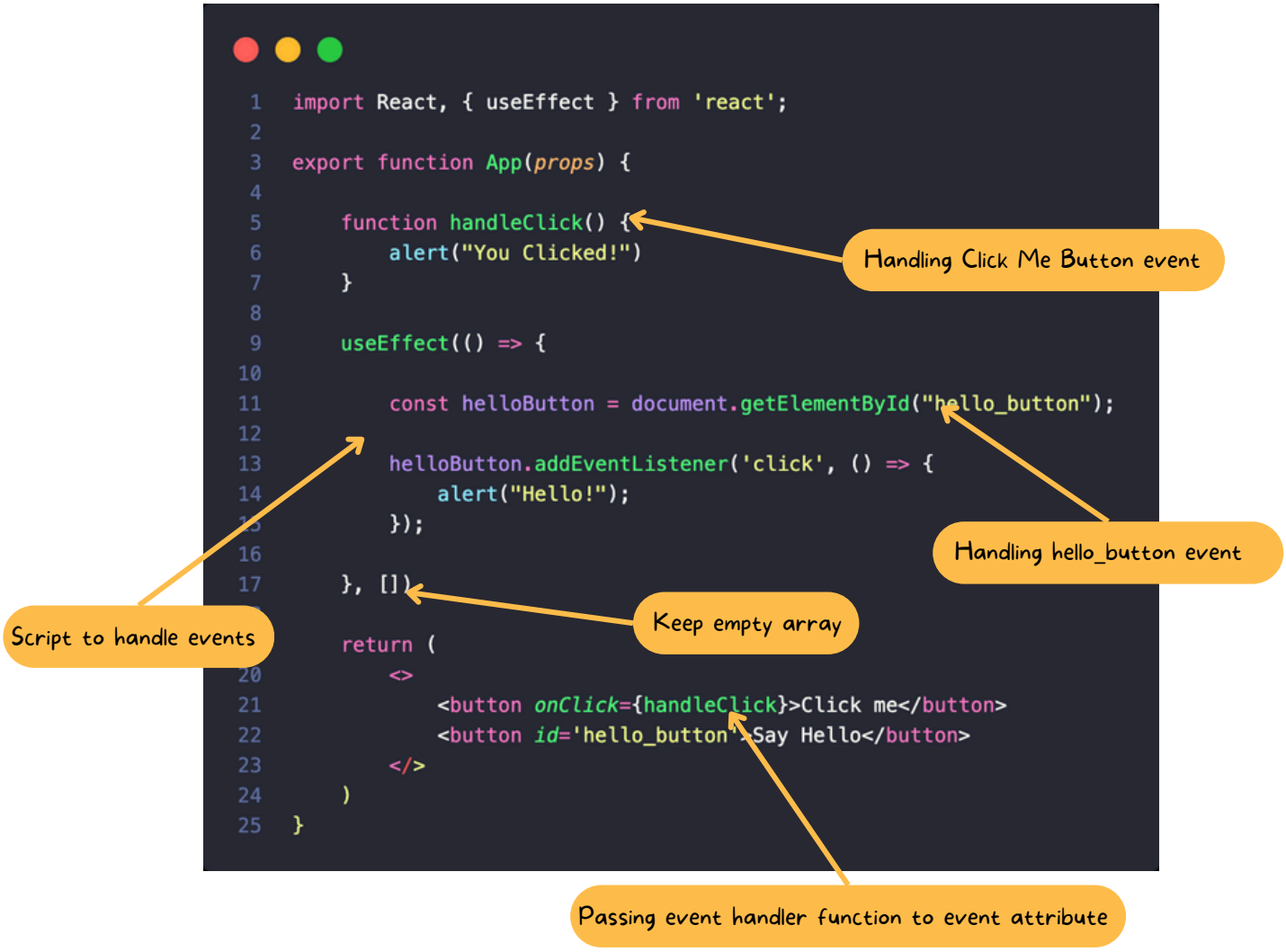
If the dependency array is removed, the function will be executed each time the component is re-rendered.

- **How to handle events in React?**

Later

We can handle events in 2 ways, first one is by passing an event handler function to an element's event attributes and the second one is by adding an event handling script in the **useEffect** hook with an empty dependency array([]).

let's take an example,

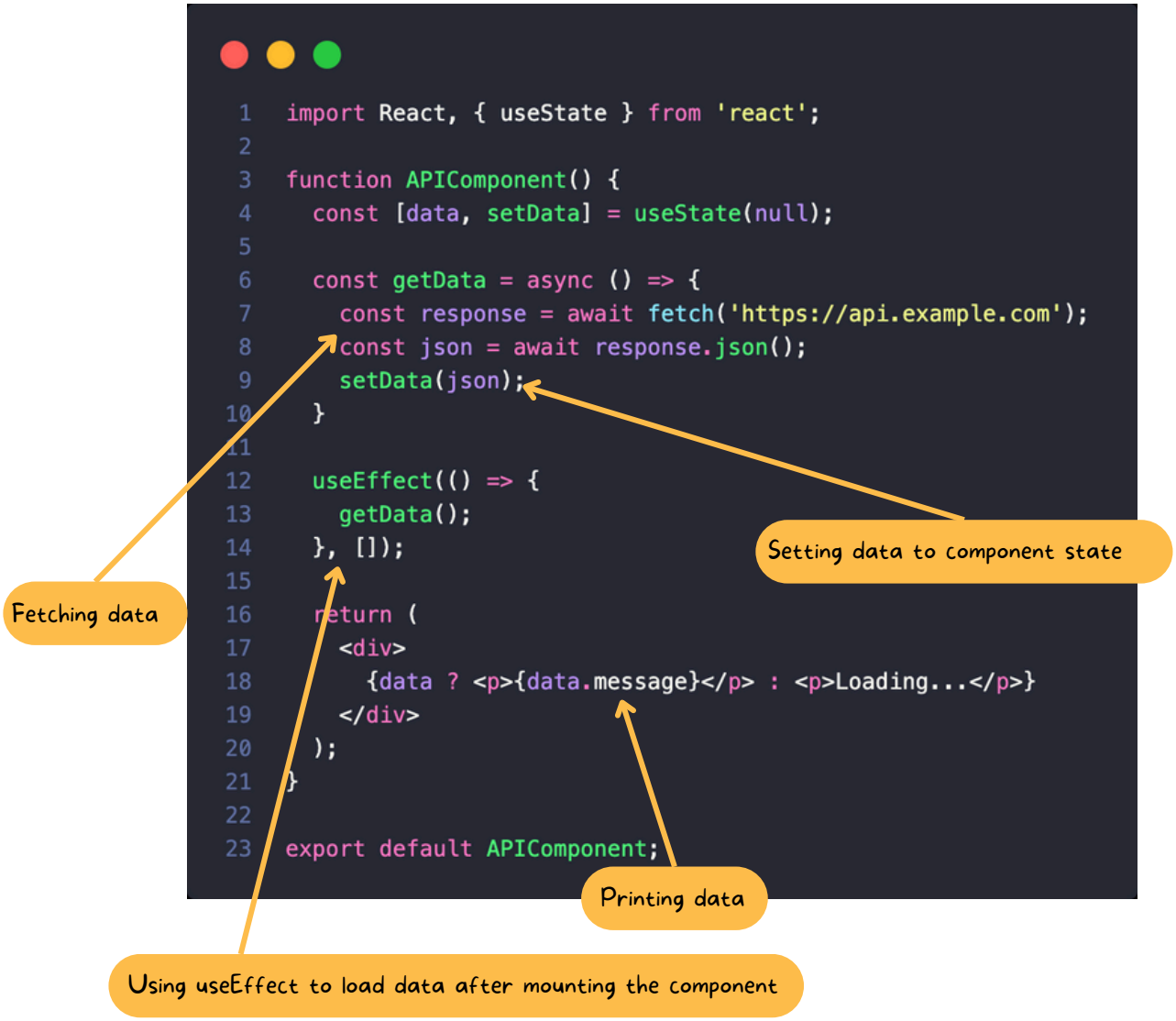


In the above example, For Click Me button we passed event handler function and to Say Hello Button we added script in useEffect hook.

- **How to Integrate REST API in React?**

  Integrating an API into a React application can be done in various ways, a popular method is by utilizing the fetch() function or a library such as axios to perform API calls and subsequently using the returned data to modify the state of the React components.

  Here is the example of fetching data from API using **fetch()**.



In the above example, You can see we have used **useEffect** hook, Because it observes the changes in the component's props or state and triggers the callback function to re-execute upon any change detected.

- **Explain Error Boundaries in React.**

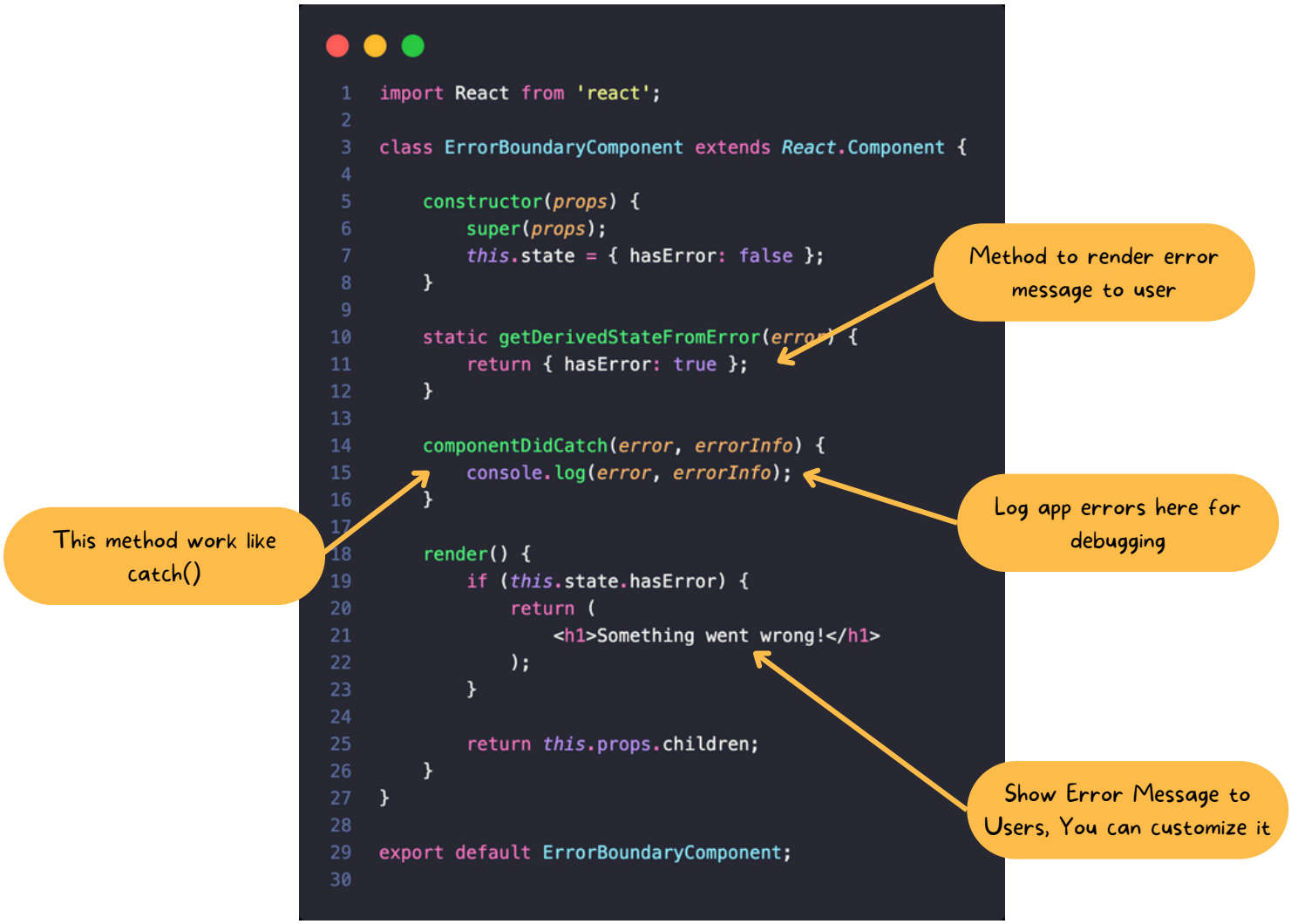
Error boundaries are React components that are used to catch JavaScript errors that occur anywhere in their child component tree.

They are used to handle unexpected errors in your application and display a fallback UI to the user, instead of letting the component tree crash.

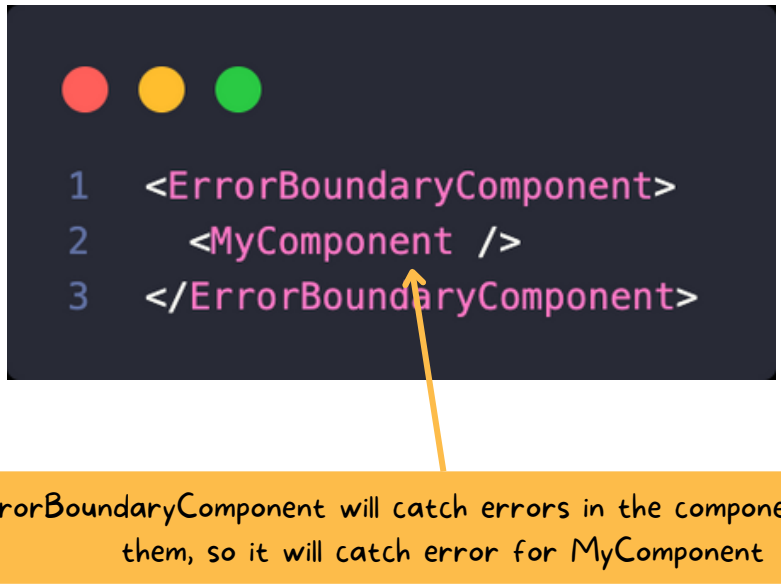
Additionally, error boundaries also log the error that occurred, which can be useful for debugging and troubleshooting.

This allows you to track and identify the cause of the error and take appropriate action to fix it.

Let's take an example of error boundary component:



Integrating Error Boundaries in your app



- **What are the constraints of using error boundaries?**

Error boundaries have some limitations, they don't catch errors that occur in the following cases:

- **Event handlers:** Any errors that happen inside event handlers will not be caught by error boundaries.
- **Asynchronous code:** Errors that occur in asynchronous code such as setTimeout or requestAnimationFrame callbacks will not be caught by error boundaries.
- **Server side rendering:** Error boundaries only work on the client-side, so any errors that happen during server-side rendering will not be caught.
- **Error boundary itself:** If an error boundary component itself throws an error, it will not catch it. The error will be propagated to the closest error boundary above it.

- **What are the different state management libraries available for React?**
  - Redux
  - Context API
  - Recoil
  - MobX
  - Jotai

- **What is redux?**

  Redux is a state management library for JavaScript applications, often used in combination with React, but can also be utilized with other JavaScript libraries and frameworks like Angular and Vue.js.

  Redux enables centralization of the application's state, which is read-only and can only be altered by dispatching actions.

- **Explain the scenario when you use Redux.**

  Let's take an example you are building an e-commerce application, the cart component is a crucial aspect that requires proper management of state.

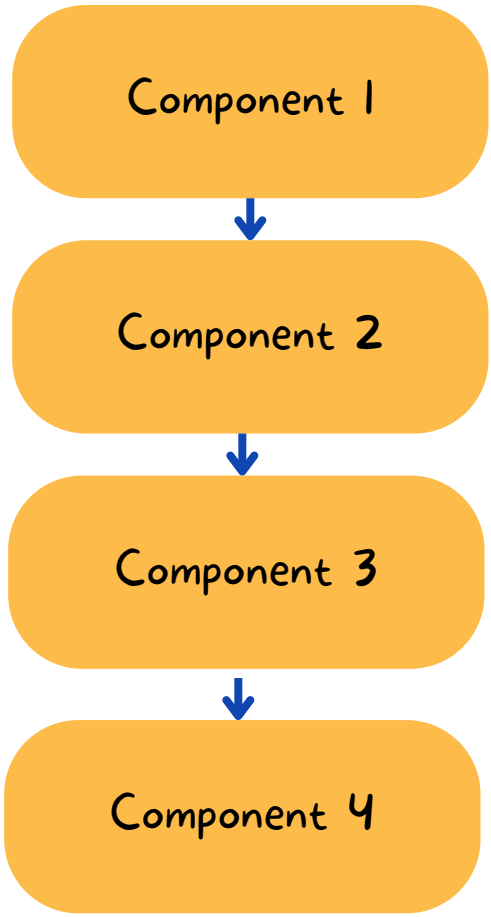
  To store the cart count and data, one approach is to use the local state of the cart component and pass the relevant information to child components via props.

  This allows for a clear and organized flow of data, and makes it easy to track and update the state of the cart throughout the application.

  However, if the need arises to share the cart data with parent components or other components in the application, relying solely on local state and props can become challenging and complex.

  It can be difficult to ensure that all necessary components are properly updated and that the data flows through the application correctly.

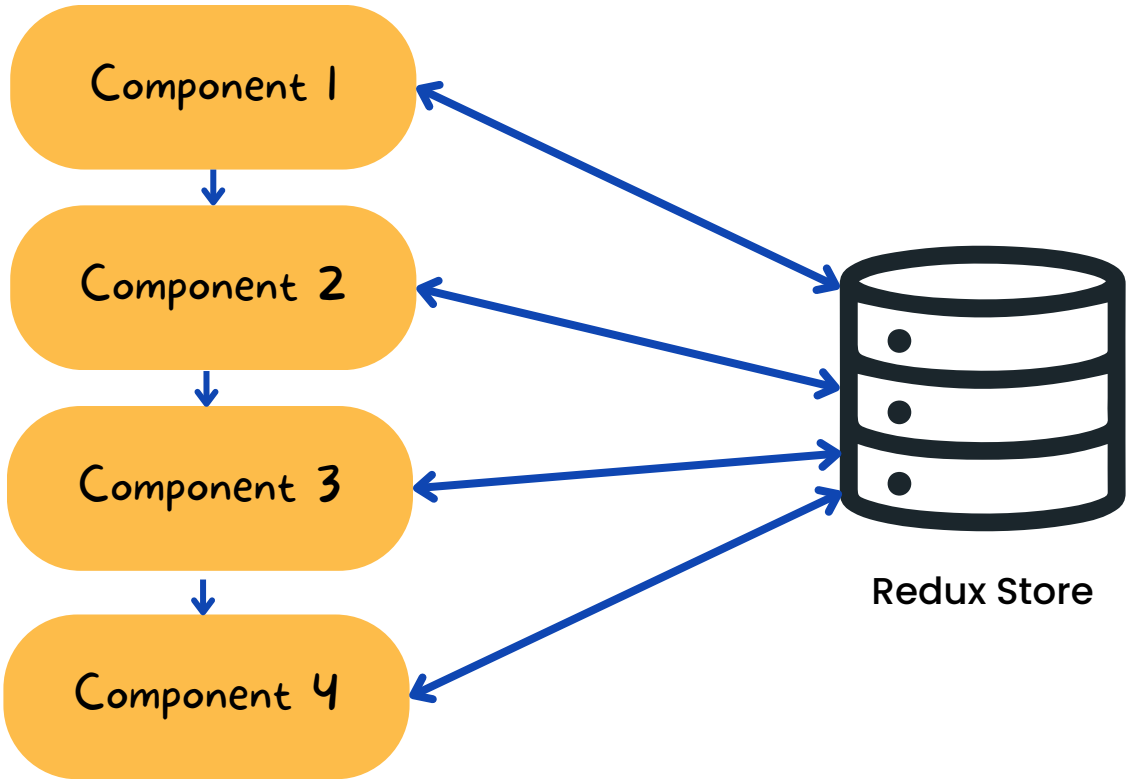
In this scenario, using a state management library like Redux can help simplify the process of sharing and updating the cart data throughout the application.



As you can see in the above diagram, we can pass data to components below it in the tree, such as from Component 1 to Component 2, then to Component 3, and finally to Component 4.

What if you have to pass data from Component 4 to Component 1? It will be very difficult to manage the flow of data.

At that time redux will help you to manage the data easily. Let's understand it through diagram.



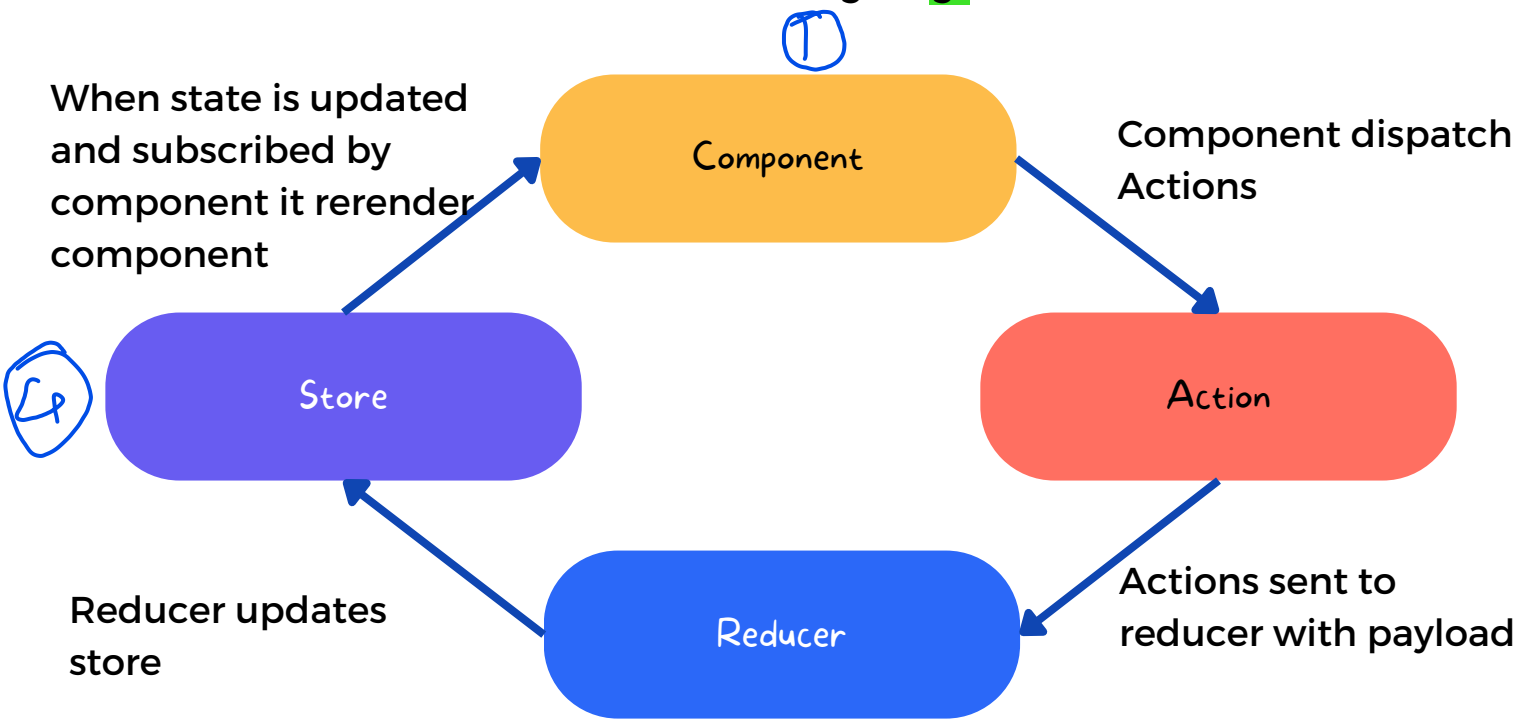
As you can see in the above diagram, Redux allows for easy access and management of the application's state by any component.

Each component can access the state from the centralized store, and update it by dispatching actions.

This approach greatly simplifies the process of managing and updating state across the entire application, as all components have access to redux store.

- **How redux works in React?**

Let's first understand the flow using diagram:



1. **Store**: A central repository holding the application's state.
2. **Actions**: Plain JavaScript objects representing events in the application.
3. **Reducers**: Functions that take current state and an action, then return a new state.
4. **Dispatch**: The method to broadcast actions to the store.

Here is how it works.

1. An event occurs (e.g. user interaction).
2. An action describing the event is dispatched.
3. The dispatched action goes to the reducer.
4. The reducer takes current state and the action to produce a new state.
5. The store updates with the new state.
6. React components connected to the store get the updated state and re-render.

- **Explain about actions in Redux.**

As you can see in the diagram, Components in the application can access the state from the store and update it by dispatching actions.

The state of your application is the collection of all data that represents the current state of the application.

For example, imagine you have a to-do list application. The state of your application would contain a list of to-do items, as well as information about which item is currently selected or being edited.

An "action" in Redux is a plain JavaScript object that represents an intention to change the state of your application. It describes what kind of change you want to make, but doesn't actually make the change itself. For example, consider the following action:

```
{
  type: "ADD_TODO",
  text: "Study Redux"
}
```

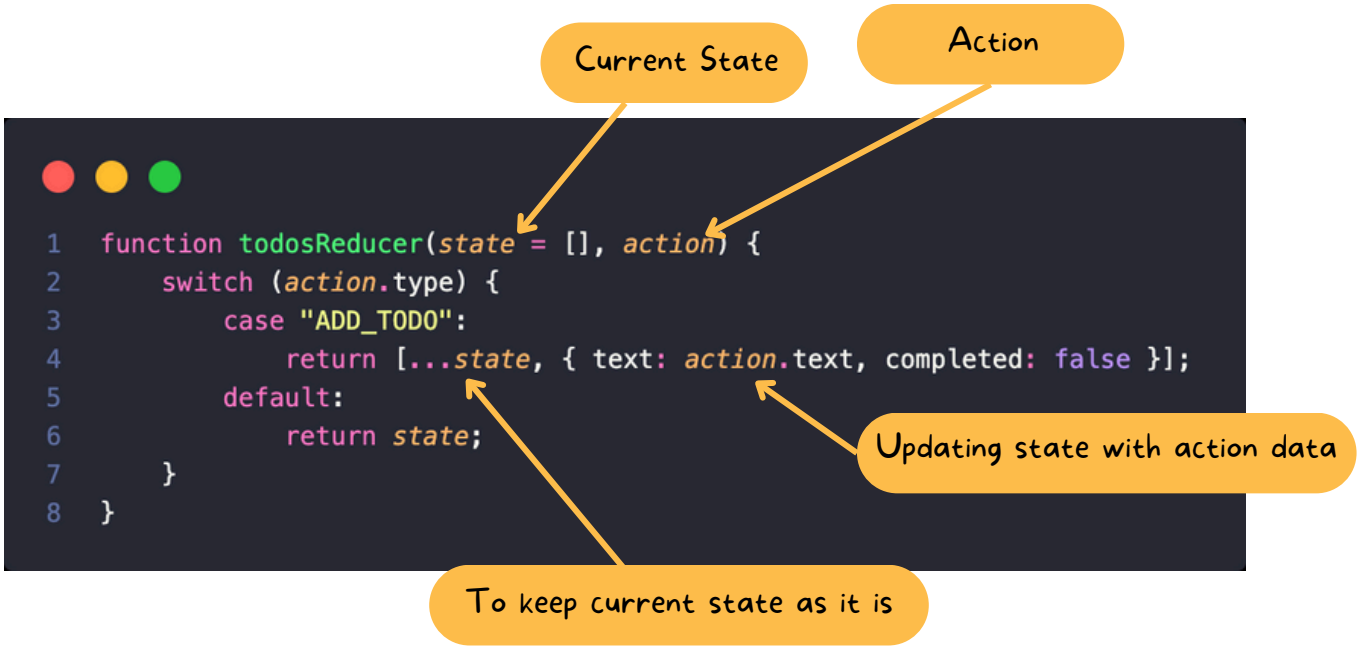
This action represents an intention to add a new to-do item with the text "Study Redux" to the list of to-do items in your application's state.

In Redux, you don't modify the state directly. Instead, you dispatch actions, which are then processed by a reducer.

- **Explain about reducer in Redux.**

A reducer is a pure function that takes the current state of your application and an action, and returns a new state that reflects the change described by the action.

For example, here's a simple reducer that processes the "ADD_TODO" action:



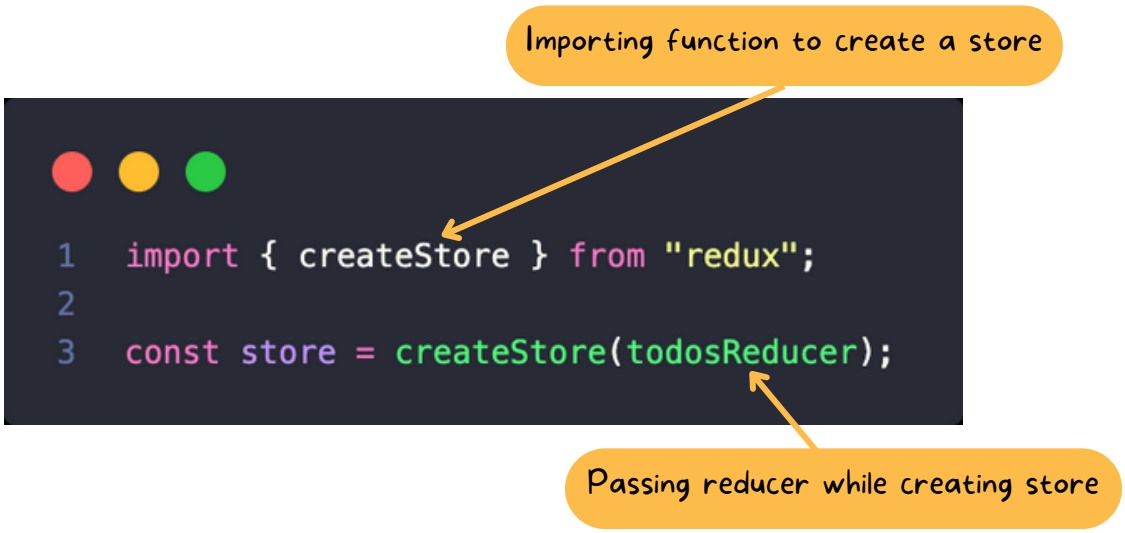
This reducer takes the current state (an array of to-do items), and returns a new state that includes the new to-do item represented by the "ADD_TODO" action.

- **Explain about store in Redux.**

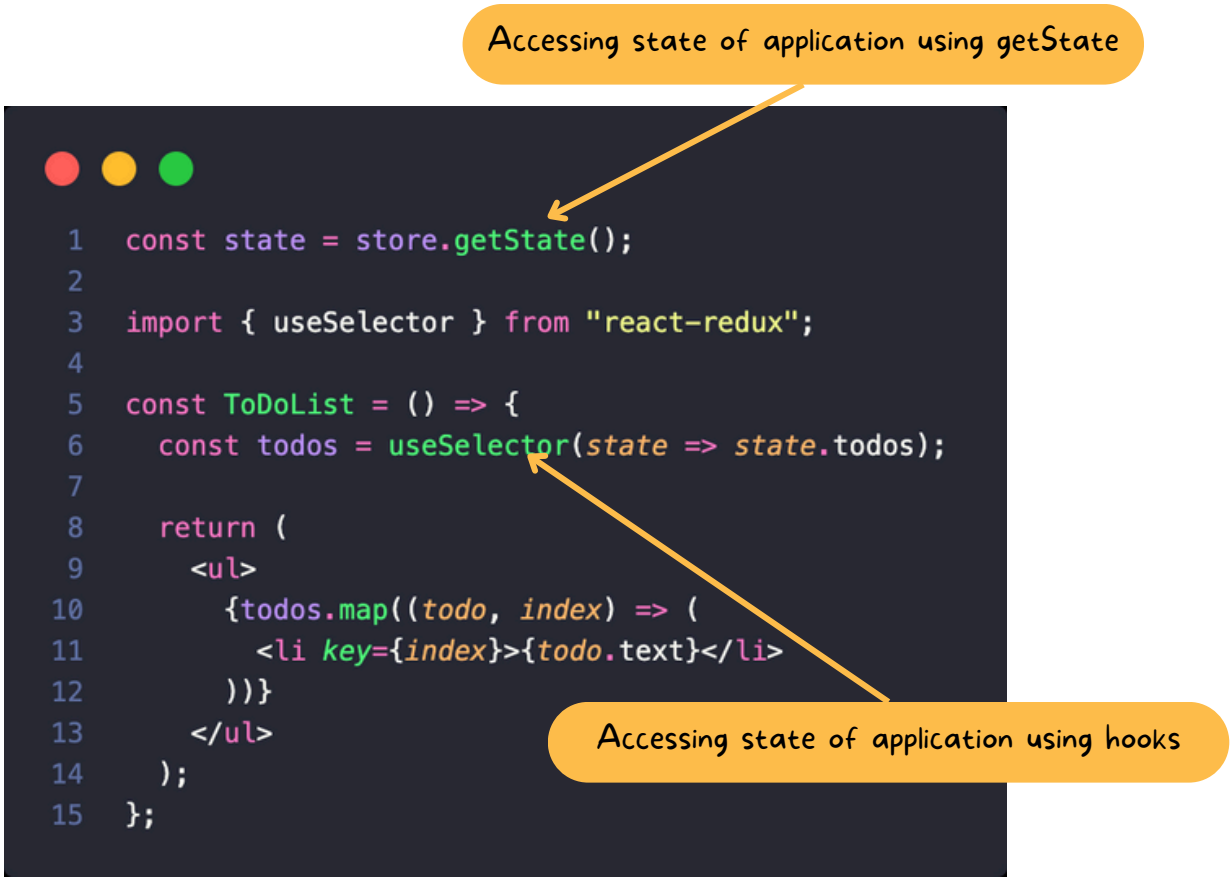
In Redux, the "store" is a central place where the state of your entire application is kept.

It's where you access the state of your application, and it's the only place where the state can be modified by dispatching actions.

The store is created by calling createStore and passing in a "reducer" function, which updates the state based on the actions dispatched to the store.



Once you have created the store, you can access the state of your application by calling the getState method on the store and if you are using a functional component you can use redux hook useSelector to access the state.

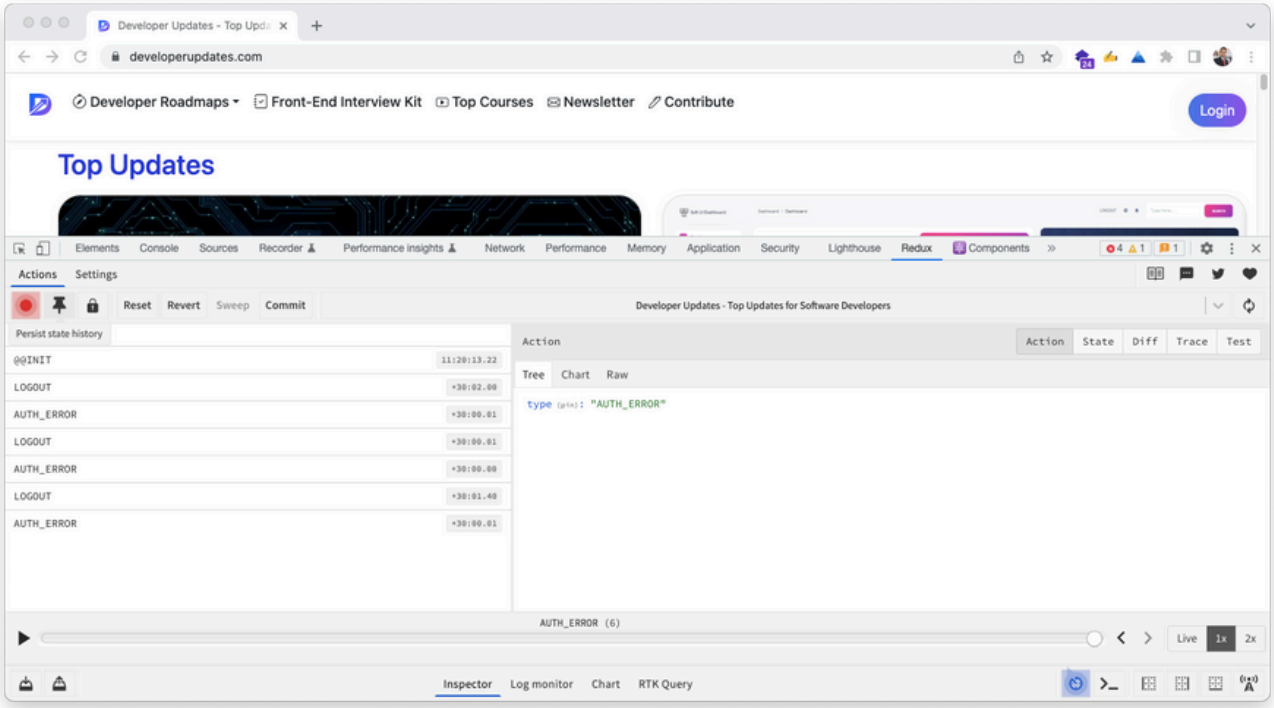


- **How do you debug redux?**

We can debug redux using Google Chrome Extension Redux DevTools.

The Redux DevTools extension provides a visual interface for exploring the state of your Redux store and the actions being dispatched to it.

Here is the screenshot of Redux DevTools



- **What are the hooks?**

  Hooks in React provide a way for functional components to have access to state and lifecycle features, which were previously only available to class components.

  They offer several benefits, including improved code organization, increased code reuse, and better performance through optimized updates and re-renders.

  Hooks allow developers to write more readable and maintainable code by separating state logic from the component logic and making it reusable across multiple components.

- **What are the various types of Hooks available in React?**
  - **Basic Hooks**
    - useState
    - useEffect
    - useContext
    - useSelector
  - **Additional Hooks**
    - useCallback
    - useReducer
    - useMemo
    - useRef
    - useImperativeHandle
    - useLayoutEffect
    - useDebugValue
    - useId

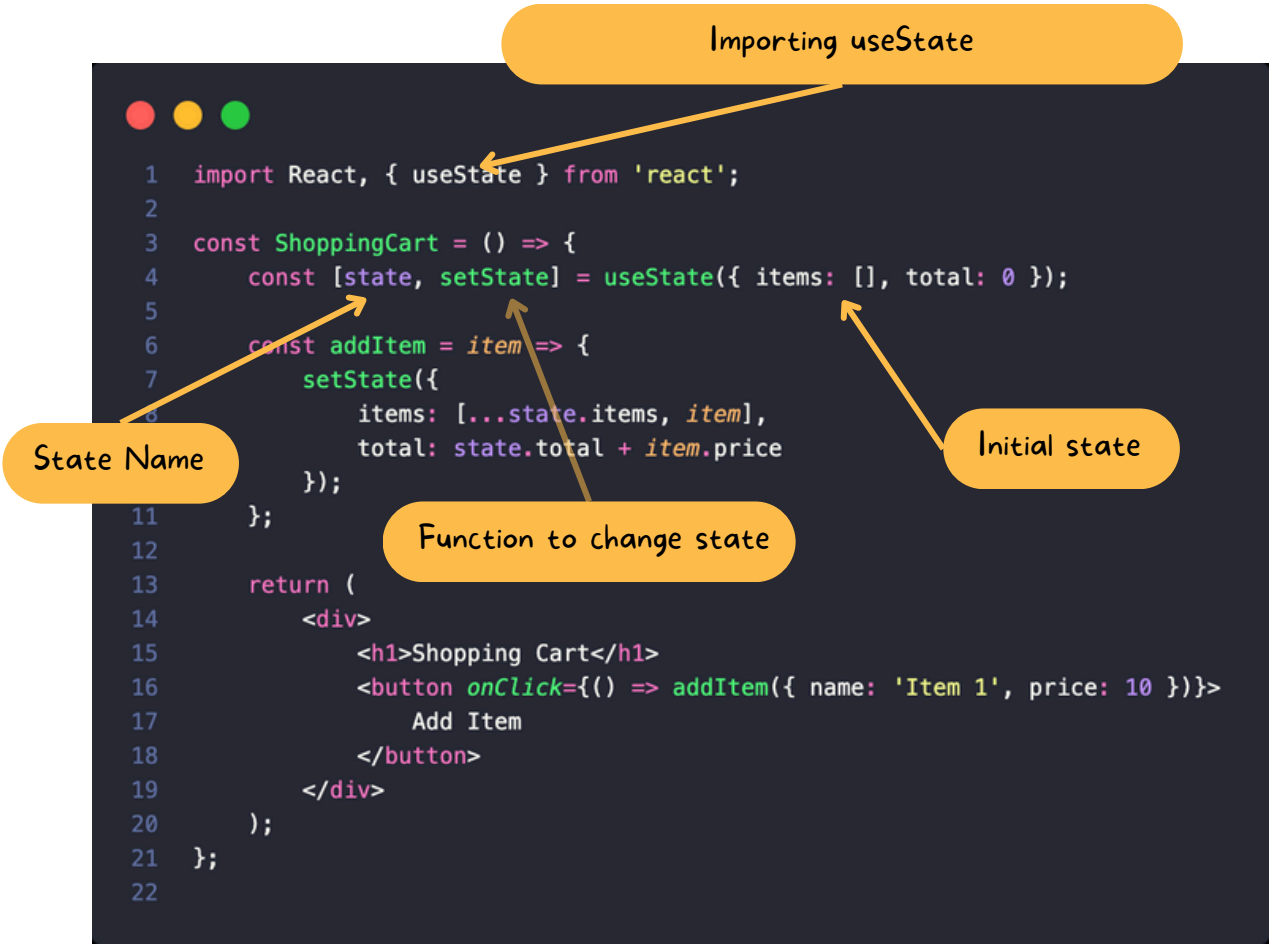
- **Explain about useState Hook.**

useState is a hook in React that allows you to manage the state of your components. State is a way to store and manipulate data in a React component.

For example, consider a shopping cart application. The shopping cart has a list of items and a total price.

The state of the shopping cart component could be represented as an object with two properties: items and total.

Here's an example of how you could use useState to manage the state of a shopping cart component:



- **Explain about useEffect Hook.**

useEffect is a hook in React that allows you to run a side effect, such as updating the state or making an API call, after rendering a component.

For example, consider a shopping cart component that fetches the list of items from a server.

You could use useEffect to run an effect that fetches the items when the component is first rendered(If you want to update after a state change you can pass it to the dependency array).

Here's an example of how you could use useEffect to fetch the items for a shopping cart component:

- **Explain about useRef Hook.**

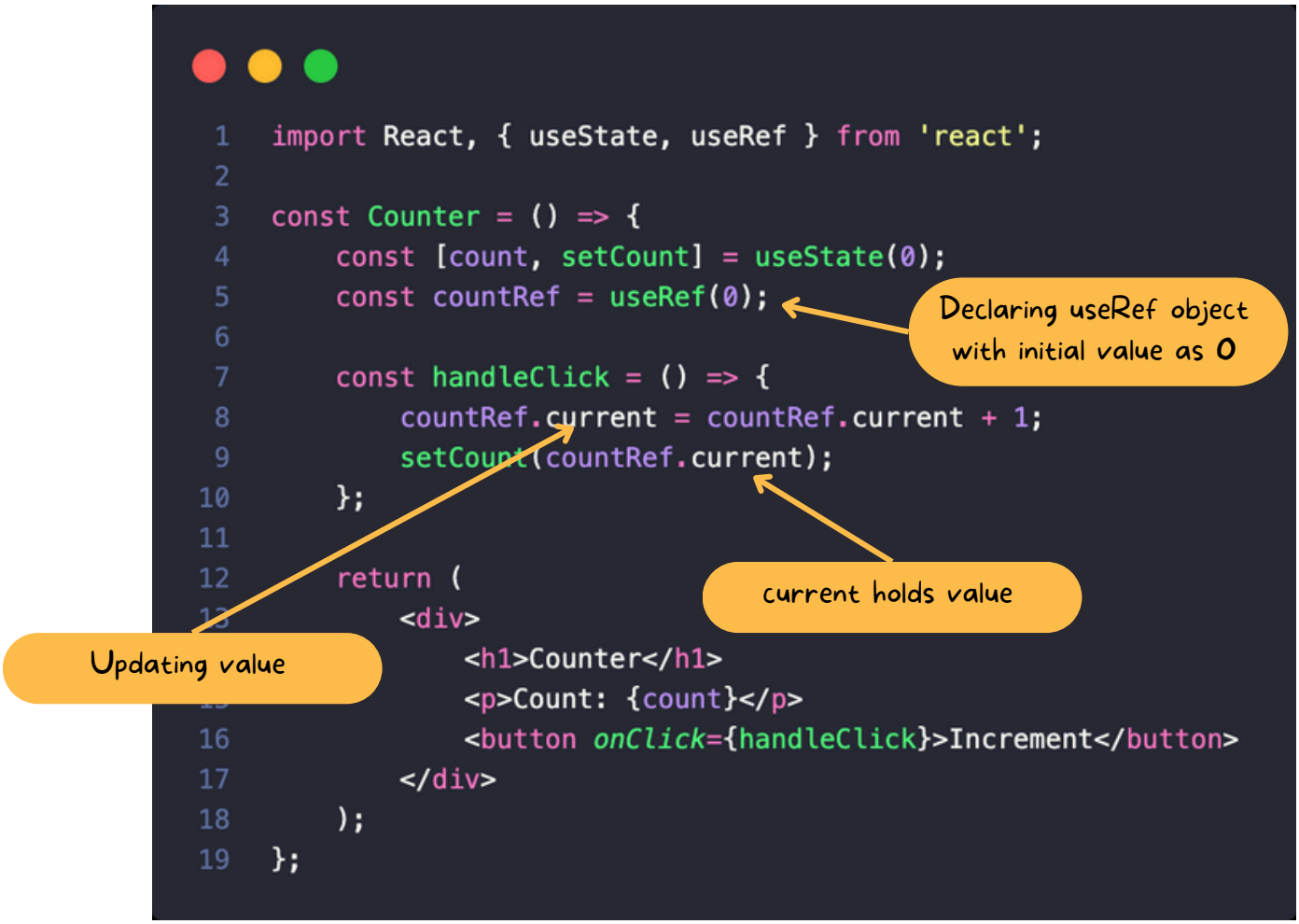
UseRef is like a memory box in React that you can use to store information that you don't want to change, but you still want to use later.

For example, you can store a reference to a DOM element in it so that you can interact with that element in your code.

When you update other parts of your component, the information stored in useRef remains unchanged.

For example, consider a counter component that has a button to increment the counter value. You could use useRef to keep track of the counter value and use it to update the display.

Here's an example of how you could use useRef to keep track of the counter value in a counter component:

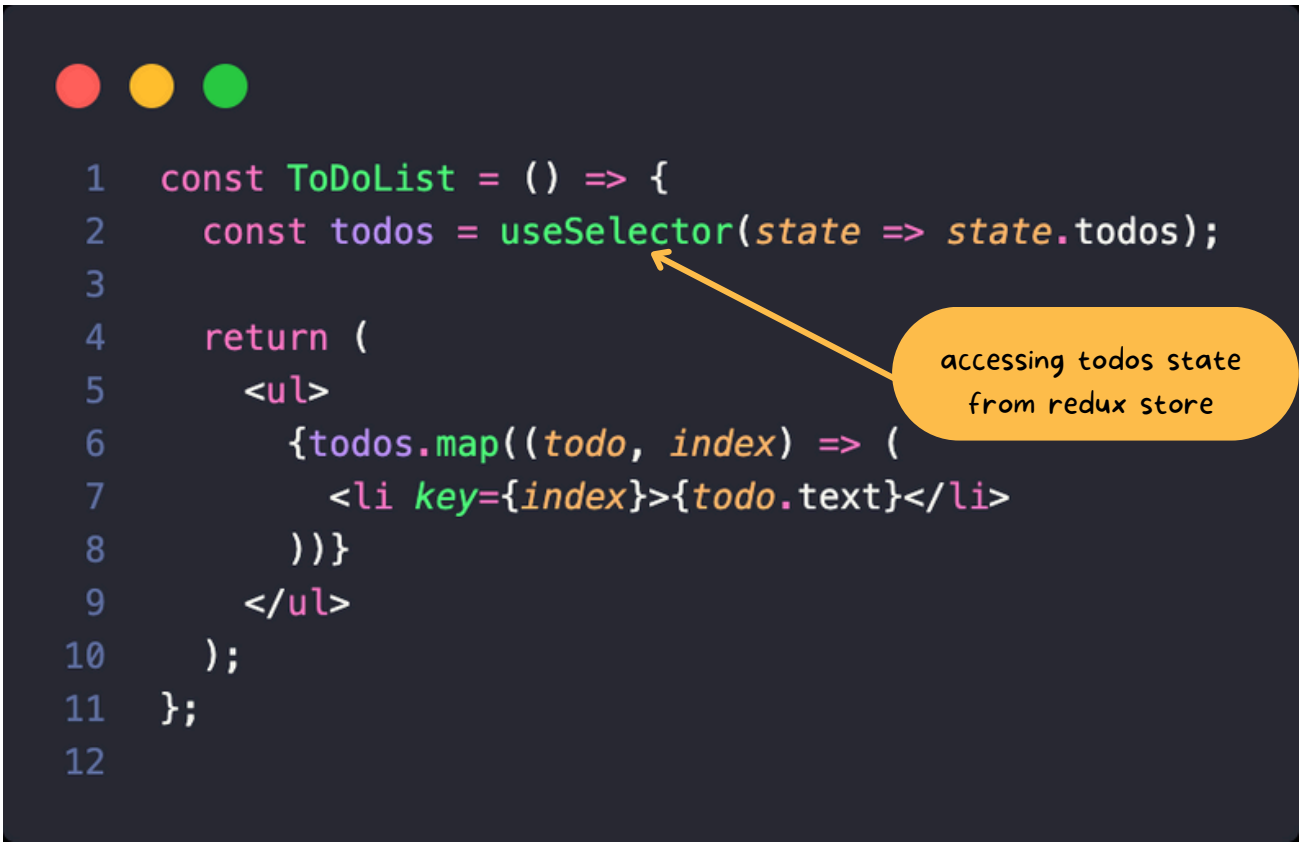


- **Explain about useSelector Hook.**

useSelector is a hook in the React-Redux library that allows you to access the state managed by Redux from your React components.

For example, in an to-do app, you are using Redux to manage the state of the to-do list.

With useSelector, you can access the current state of the to-do list in your React component, so that you can display the all tasks and mange it.



- **Explain about useId Hook.**

useId is a hook in React that generates a unique identifier (id) that can be used for accessibility purposes,

When building accessible UIs, it is important to ensure that all components have a unique and consistent identifier.

The `useId` hook provides a way to generate such identifiers in a way that is consistent across renders and avoids naming collisions.

Let's suppose you are building a simple menu component that allows users to select an option.

You use the `useId` hook to generate a unique identifier for each option, which you then use to associate each option with a label for accessibility purposes. Here is an example:

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

function Menu() {
  const [selectedOption, setSelectedOption] = useState('');
  const options = ['Option 1', 'Option 2', 'Option 3'];

  return (
    <div role="menu">
      {options.map((option, index) => {
        const optionId = useId();

        return (
          <div key={optionId} role="menuitem"
            aria-selected={selectedOption === option}
            onClick={() => setSelectedOption(option)}
          >
            <label htmlFor={optionId}>{option}</label>
          </div>
        );
      })}
    </div>
  );
}
```

Generating unique identifier for each option

- **How to prevent re-renders in React?**

One common technique is to use the shouldComponentUpdate() lifecycle method.

This method allows you to tell React when it should and should not update a component.

The default behavior is for the component to re-render whenever its state or props change, but you can override this behavior with shouldComponentUpdate().

Here's an example of how to use shouldComponentUpdate() to prevent re-renders:

```
class MyComponent extends React.Component {
    shouldComponentUpdate(nextProps, nextState) {
        if (this.props.someProp === nextProps.someProp) {
            return false;
        }
        return true;
    }

    render() {
        return <div>{this.props.someProp}</div>;
    }
}
```

It will re-render only when someProp changes

Another technique for preventing re-renders is to use React.memo(). This is a higher-order component that can "memoize" a component based on its props.

Essentially, it will only re-render the component if its props have changed. Here's an example:

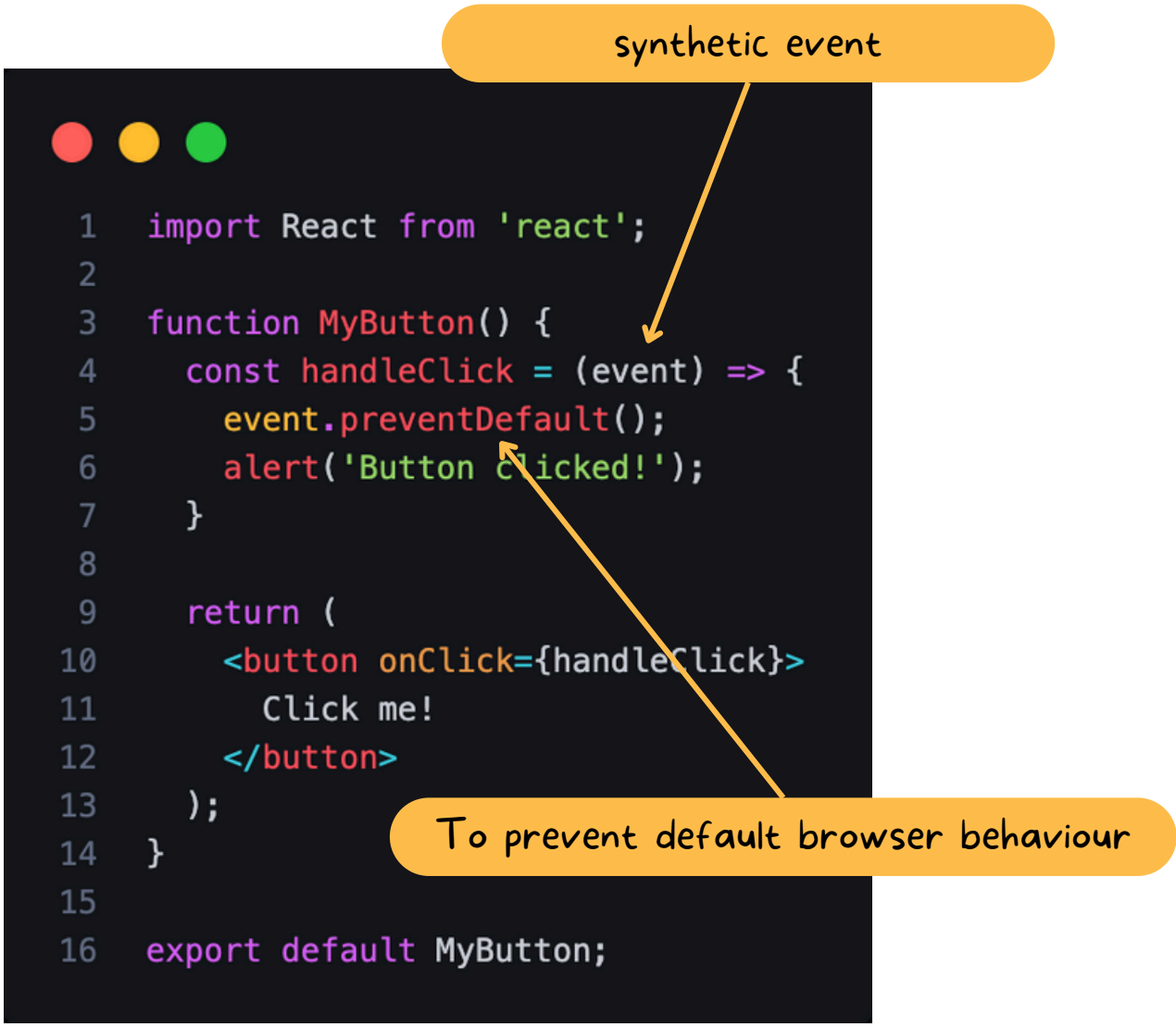
```
const MyComponent = React.memo(props => {
    return <div>{props.someProp}</div>;
});
```

- **What are Synthetic Events?**

A synthetic event is a cross-browser wrapper around the browser's native event. They combine the behavior of different browsers into one API, ensuring that your React apps work consistently across all browsers.

Synthetic events in React serve as a universal interpreter for browser interactions. Synthetic events standardize browser interactions like clicks or keyboard input. This saves you from dealing with varied browser behaviors.

Let's understand it with example:



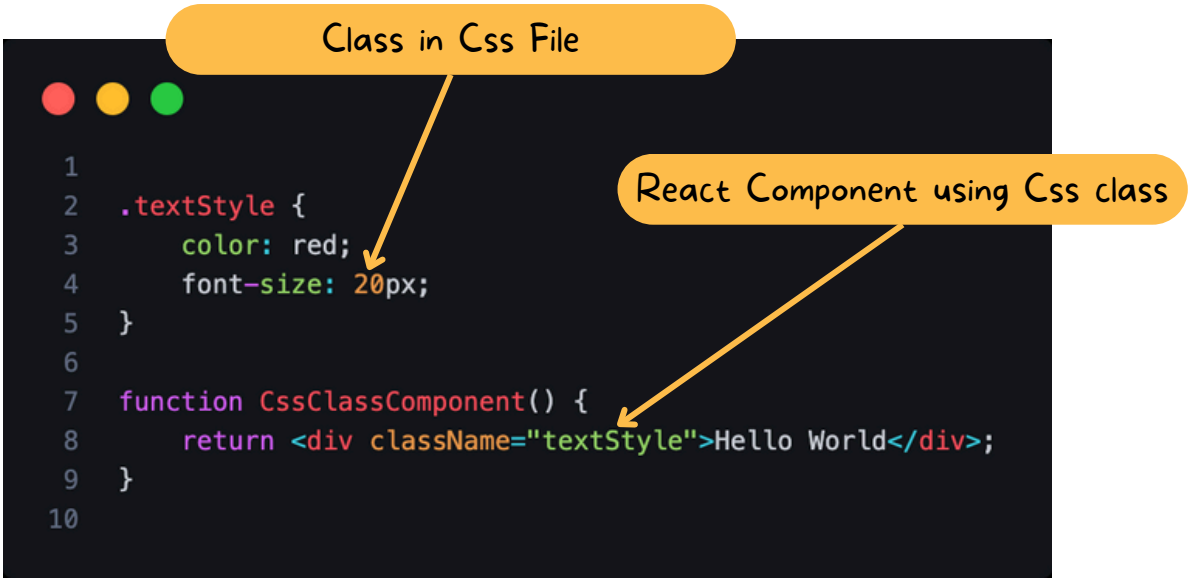
- **What are the different ways to style a React component?**

Here are the four main methods to style React Components:

1. CSS Classes:

This method uses traditional CSS in an **external .css file.**

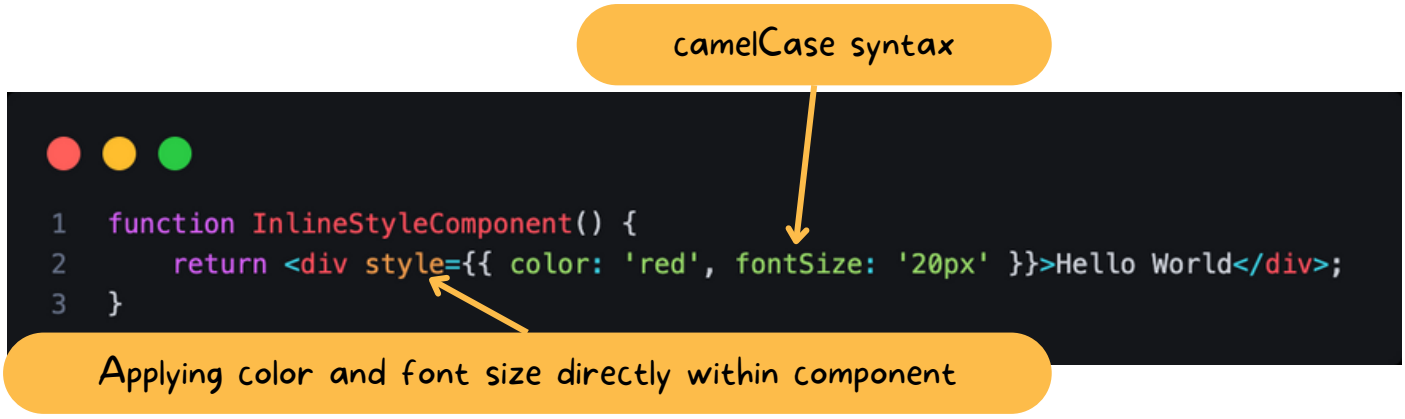
You define classes in your CSS file, and then reference them in your React component with the className attribute.



2. Inline Styles:

Here, we apply styles directly within the React component. React's inline styles don't behave exactly like CSS.

Rather than a CSS property being a string of hyphen-separated words, **React uses camelCase syntax.**

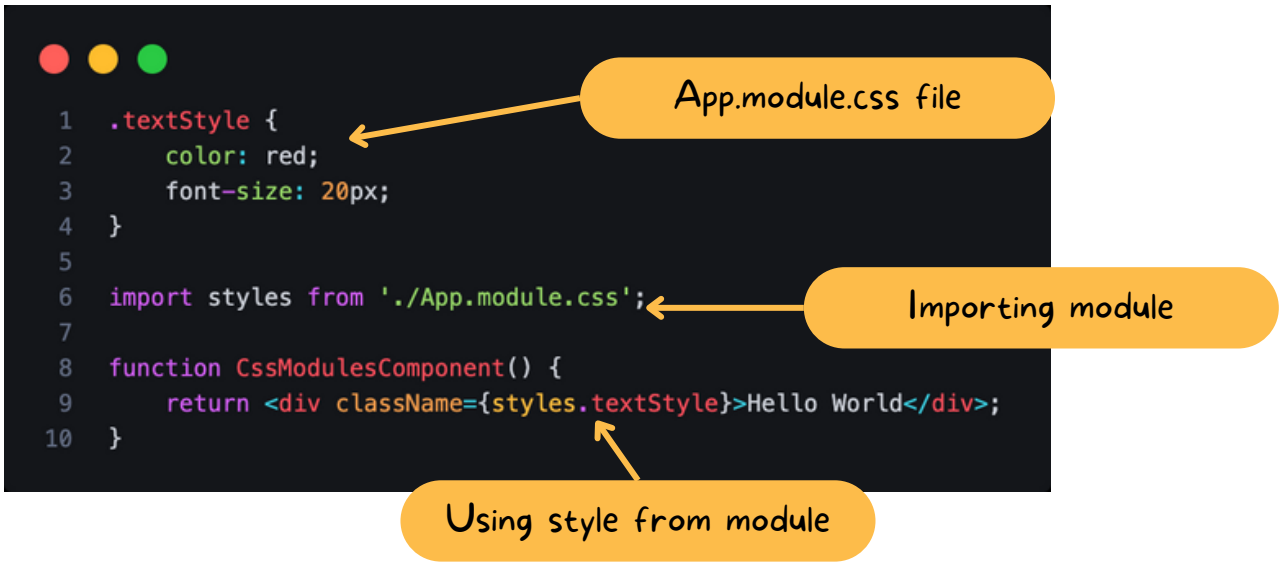


3. Styled-components:

This is a library you can install in your React project. It lets you write actual CSS in your JavaScript files. It keeps the styles scoped to the components, avoiding conflicts.

4. CSS Modules:

With CSS Modules, you can create CSS files that are scoped locally by default. It means you can use the same CSS class in different files without worrying about naming clashes.



- **What is redux thunk?**

Redux Thunk is middleware for Redux that lets you write action creators that return a function instead of an action.

It can be used to delay the dispatch of an action or to dispatch only if certain conditions are met. Essentially, it gives your action creators the power to be asynchronous.

Typically, Redux actions are dispatched immediately. But what if we want to make an API call, wait for a response, and then dispatch an action with the response data?

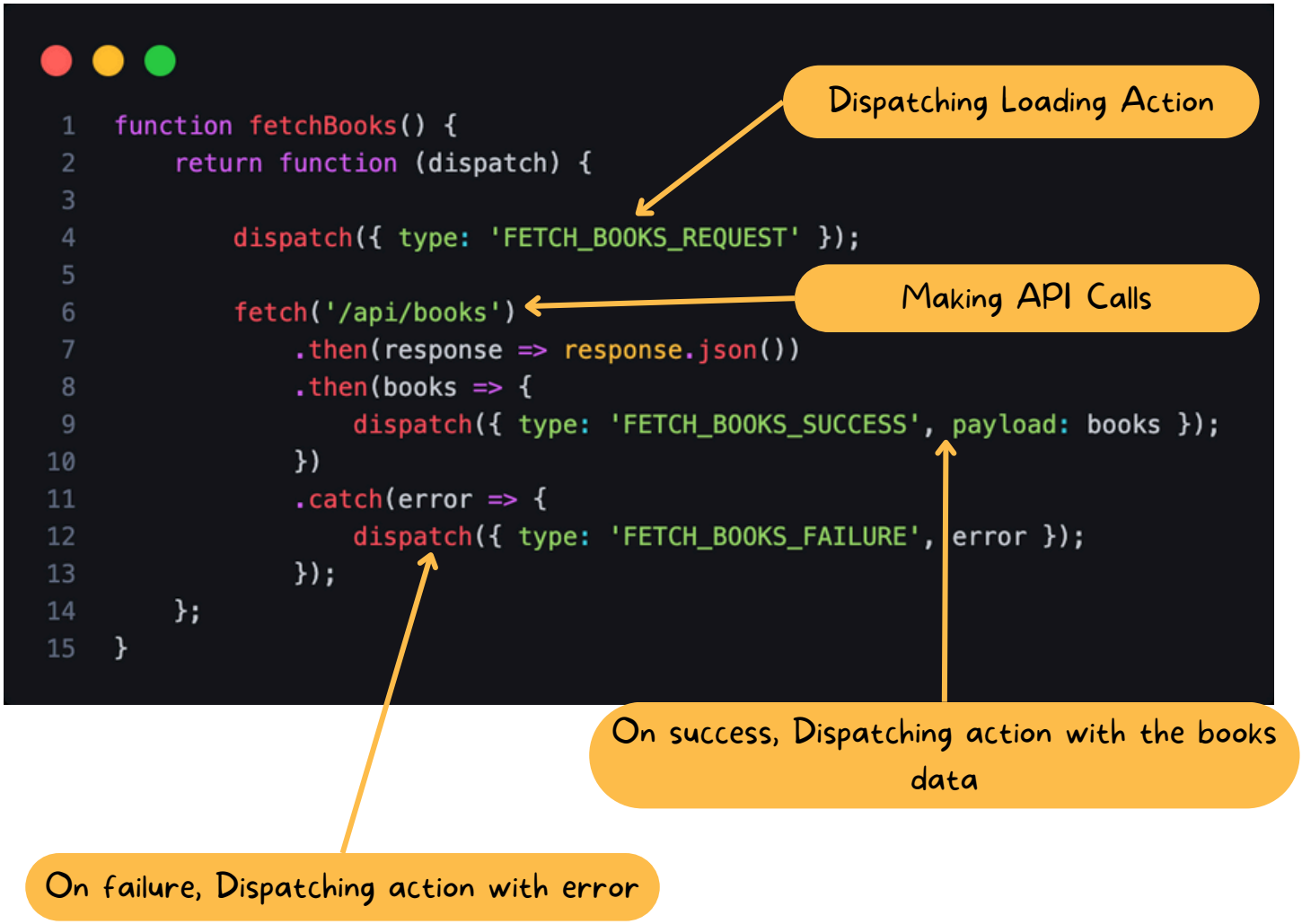
That's where Redux Thunk comes into play.

Let's say we're creating a simple application to fetch and display a list of books from an API.

Without Redux Thunk:

```
function fetchBooks() {
  return {
    type: 'FETCH_BOOKS'
  };
}
```

With Redux Thunk:




```
function fetchBooks() {
    return function (dispatch) {

        dispatch({ type: 'FETCH_BOOKS_REQUEST' });

        fetch('/api/books')
            .then(response => response.json())
            .then(books => {
                dispatch({ type: 'FETCH_BOOKS_SUCCESS', payload: books });
            })
            .catch(error => {
                dispatch({ type: 'FETCH_BOOKS_FAILURE', error });
            });
    };
}
```

- **Differentiate between server-side rendering and client-side rendering in React.**

1. Initial Load

- **Client-side:** The server sends a minimal HTML page with links to JavaScript files.
- **Server-side:** The server sends a fully rendered HTML page to the browser.

2. JavaScript Download

- **Client-side:** The browser downloads the JavaScript files containing React code.
- **Server-side:** The browser **may** download JavaScript files for interactivity.

3. Rendering

- **Client-side:** React creates and manages the DOM in the browser.
- **Server-side:** This HTML is immediately viewable and contains the page's content.

4. Interactivity

- **Client-side:** The web page becomes interactive after all scripts are downloaded and executed.
- **Server-side:** The page becomes fully interactive after the JavaScript is executed.

5. Navigation

- **Client-side:** Navigation between pages happens within the browser without additional requests to the server.
- **Server-side:** Navigation to new pages typically requires a new request to the server.

6. Performance

- **Client-side:** Initial page load might be slower, but subsequent page loads are faster.
- **Server-side:** Faster initial page load, but subsequent page loads might be slower.

7. SEO Impact

- **Client-side:** May not be as SEO-friendly because the content is rendered after the initial page load.
- **Server-side:** Better for SEO as the content is present in the HTML on the initial load.

- **How do you optimize React applications for performance?** optimising MILF of SECR

1. Component Should Update Carefully

Implement **shouldComponentUpdate** or **React.memo** to prevent unnecessary re-renders by comparing props or state.

2. Use Functional Components and Hooks

Functional components with Hooks are generally more performant than class components.

3. Lazy Loading Components
Use React.lazy for dynamic imports of components that aren't immediately needed. This reduces the initial load time.

4. Code Splitting
Split your code into smaller chunks using dynamic **import()** statements or libraries like **Loadable Components**. This ensures users only download what's necessary for the current view.

5. Use Key Prop Appropriately in Lists
Ensure that each list item has a unique and consistent key prop for efficient re-rendering.

6. Throttling and Debouncing Event Handlers
This can optimize events like scrolling, typing, or window resizing that trigger a high number of updates.

7. Optimize Images and Assets
Compress images and use appropriate formats. Consider using techniques like lazy loading for images.

8. Avoiding Memory Leaks
Clean up subscriptions and intervals in your component's **useEffect** cleanup function.

- **Create a Reusable Star Widget**

StarRating.jsx

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

// Star Rating Component
const StarRating = ({ totalStars = 5, value, onChange }) => {
  const [rating, setRating] = useState(value || 0);
  const [hoverRating, setHoverRating] = useState(0);

  const handleMouseEnter = index => {
    setHoverRating(index);
  };

  const handleMouseLeave = () => {
    setHoverRating(0);
  };

  const handleClick = index => {
    setRating(index);
    onChange(index);
  };

  const starStyle = {
    fontSize: '2rem',
    color: '#e4e5e9',
    cursor: 'pointer',
  };

  const filledStarStyle = {
    ...starStyle,
    color: '#ffc107',
  };

  const starRatingContainerStyle = {
    display: 'inline-flex',
    alignItems: 'center',
  };

  return (
    <div style={starRatingContainerStyle}>
      {Array.from({ length: totalStars }, (_, index) => (
        <span
          key={index}
          style={
            hoverRating > 0 ? hoverRating >= index + 1 ? filledStarStyle : starStyle : rating >= index + 1 ? filledStarStyle : starStyle
          }
          onMouseEnter={() => handleMouseEnter(index + 1)}
          onMouseLeave={handleMouseLeave}
          onClick={() => handleClick(index + 1)}
        >
          &#9733;
        </span>
      ))}
    </div>
  );
};

export default StarRating;
```

App.jsx

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

const App = () => {
  const [rating, setRating] = useState(0);

  const handleRatingChange = newRating => {
    setRating(newRating);
    // Do something with the new rating value
  };

  return (
    <div>
      <h2>Star Rating Demo</h2>
      <StarRating value={rating} onChange={handleRatingChange} />
      <p>Selected Rating: {rating}</p>
    </div>
  );
};

export default App;
```

The provided code is a reusable React component called StarRating that renders a star rating widget.

The component accepts three props:

1. **totalStars** (optional, default: 5): The total number of stars to display.
2. **value** (optional, default: 0): The initial rating value.
3. **onChange** (required): A callback function that gets called when the user selects a new rating.

The component uses the useState hook to manage the current rating value and the hover state for highlighting stars on mouseover.

It renders an array of span elements representing the stars, with their appearance (filled or unfilled) determined by the rating and hover state.

The inline styles are defined as JavaScript objects, with separate styles for filled and unfilled stars, as well as the container div.

The styles are applied conditionally based on the rating and hover state using a ternary operator.

The component handles mouse enter, mouse leave, and click events on the star icons, updating the component's state accordingly and calling the onChange callback when the user selects a new rating.

- **How would you handle server-side rendering in a ReactJS application?**

When handling server-side rendering (SSR) in a React.js application, I would follow these steps:

1. Set up the Node.js server:
- Use a Node.js server framework like Express.js
- Configure the server to handle React routes and serve static files

2. Render React components on the server:
- Use a library like React-DOM/server to render React components
- Fetch data from APIs and pass it as props to components

3. Send the rendered HTML to the client:
- Send the initial rendered HTML from the server
- Include a script tag to load the React app bundle

4. Hydrate the React app on the client:
- Use React-DOM.hydrate() to attach event handlers to the markup
- This ensures a seamless transition from server to client rendering

5. Configure code-splitting and routing:

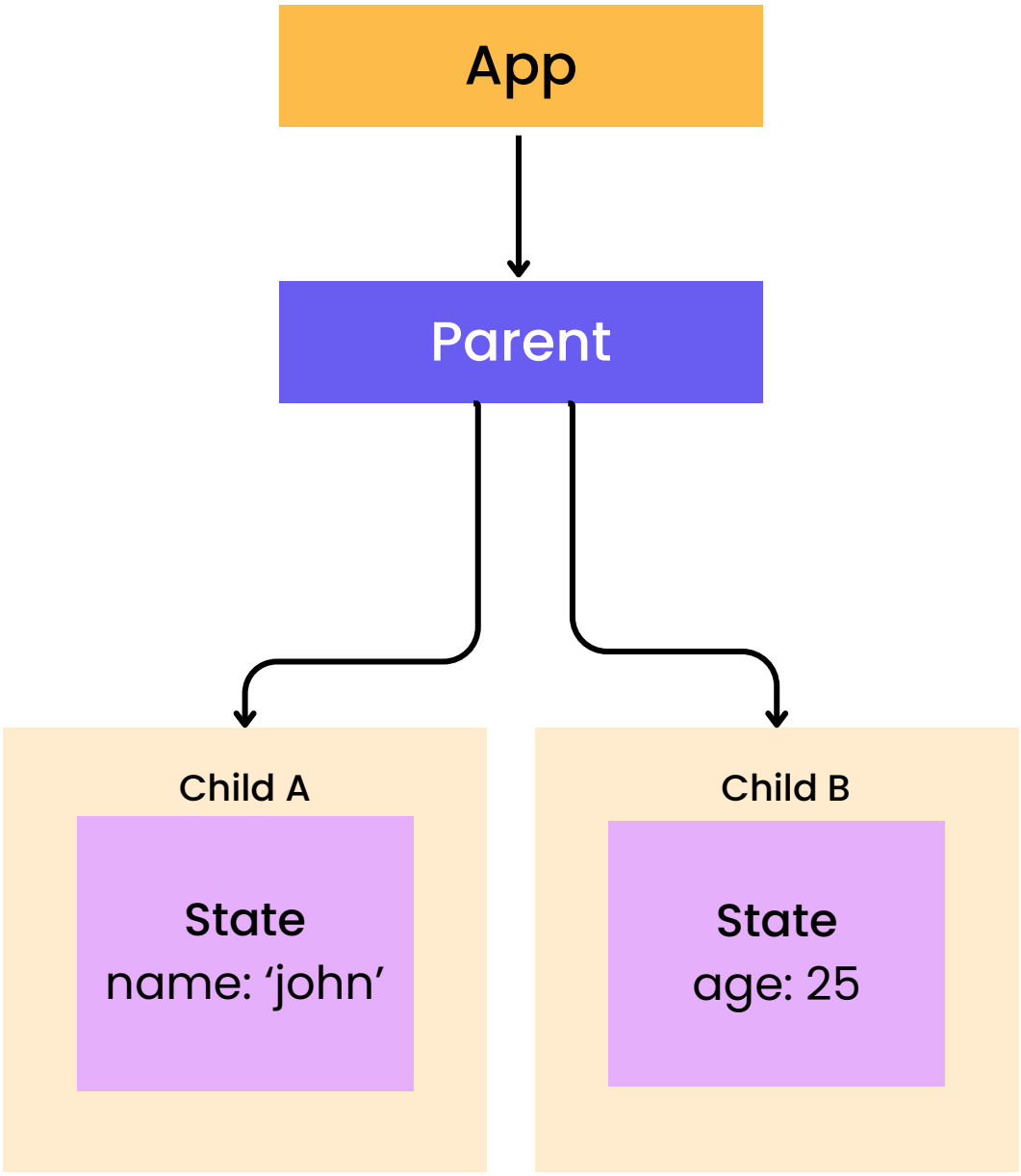
- Use a library like React Router for client-side routing
- Implement code-splitting to load components lazily

6. Deploy the server-rendered React app:

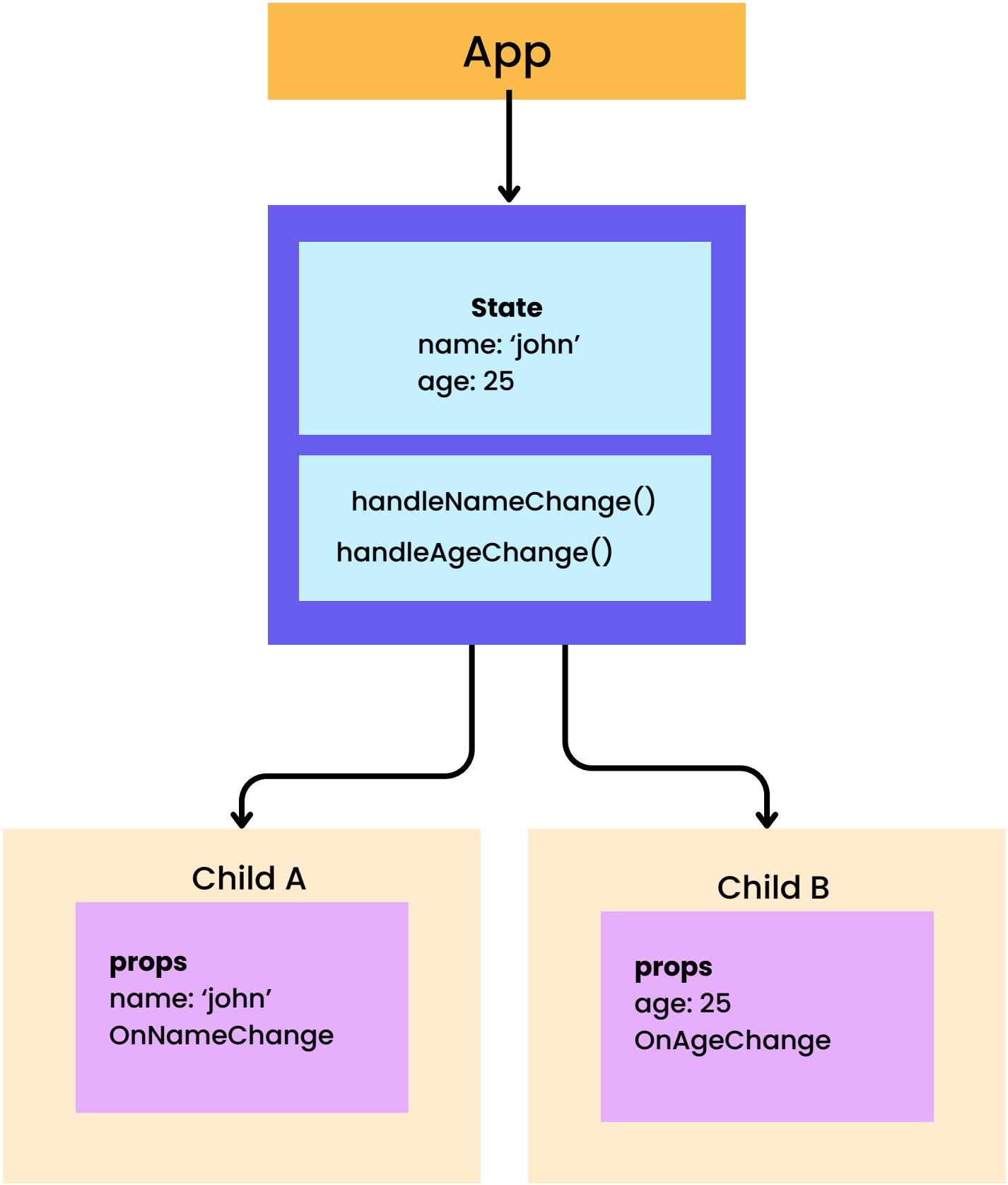
- Bundle the server and client code separately
- Deploy the Node.js server and serve static client files

- **Can you explain the concept of "lifting state up" in ReactJS?**

Without Lifting State Up



With Lifting State Up



"Lifting state up" is a technique used in React to **share state between components**.

It is necessary when multiple components need to access and/or modify the same state.

The process involves moving the **shared state up to the closest common ancestor component**.

This ancestor component then manages and passes down the state to its child components as props.

Child components can receive the shared state as props.

They can also receive functions from the parent to update the shared state.

This way, instead of multiple components managing their own state copies.

The state is managed by a single component and passed down to other components.

Lifting state up promotes a unidirectional data flow.

Data flows down from the parent component to child components through props.

Any state updates are achieved by calling functions passed down from the parent component.

This makes the app more predictable and easier to debug.

Without Lifting State Up

```jsx
// NameInput.js
import React, { useState } from 'react';

const NameInput = () => {
  const [name, setName] = useState('');

  return (
    <div>
      <label>
        Name:
        <input
          type="text"
          value={name}
          onChange={(e) => setName(e.target.value)}
        />
      </label>
    </div>
  );
};

// AgeInput.js
import React, { useState } from 'react';

const AgeInput = () => {
  const [age, setAge] = useState('');

  return (
    <div>
      <label>
        Age:
        <input
          type="number"
          value={age}
          onChange={(e) => setAge(e.target.value)}
        />
      </label>
    </div>
  );
};
```

With Lifting State Up

```jsx
// Parent.js
import React, { useState } from 'react';

const Parent = () => {
  const [name, setName] = useState('');
  const [age, setAge] = useState('');

  const handleNameChange = (e) => {
    setName(e.target.value);
  };

  const handleAgeChange = (e) => {
    setAge(e.target.value);
  };

  return (
    <div>
      <NameInput name={name} onNameChange={handleNameChange} />
      <AgeInput age={age} onAgeChange={handleAgeChange} />
    </div>
  );
};

// NameInput.js
const NameInput = ({ name, onNameChange }) => (
  <div>
    <label>
      Name:
      <input type="text" value={name} onChange={onNameChange} />
    </label>
  </div>
);

// AgeInput.js
const AgeInput = ({ age, onAgeChange }) => (
  <div>
    <label>
      Age:
      <input type="number" value={age} onChange={onAgeChange} />
    </label>
  </div>
);
```

- **How would you handle form validation in a ReactJS application?**

In React, form validation can be done by maintaining form state and performing validation on state changes or form submission.

Let's take a simple example of an email input field:


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

const MyForm = () => {
  const [email, setEmail] = useState('');
  const [emailError, setEmailError] = useState('');

  const handleEmailChange = (e) => {
    const value = e.target.value;
    setEmail(value);

    const emailRegex = /\S+@\S+\.\S+/;
    if (!emailRegex.test(value)) {
      setEmailError('Please enter a valid email address');
    } else {
      setEmailError('');
    }
  };

  const handleSubmit = (e) => {
    e.preventDefault();
    if (emailError) {
      console.log('Form submission failed due to validation errors');
    } else {
      console.log('Form submitted successfully');
    }
  };

  return (
    <form onSubmit={handleSubmit}>
      <input
        type="email"
        value={email}
        onChange={handleEmailChange}
        placeholder="Enter your email"
      />
      {emailError && <div style={{ color: 'red' }}>{emailError}</div>}
      <button type="submit">Submit</button>
    </form>
  );
}

export default MyForm;
```

In this example:

1. We use the **useState hook** to store the email value and error message.
2. On **email input change**, we validate the email format using a regular expression.
3. If the email is invalid, we set the **emailError state** with an appropriate error message.
4. **On form submission**, we check if there are any validation errors.
5. If errors exist, **we prevent form submission** and log an error message.
6. If no errors, we log a success message (and can perform further actions like submitting the form data to a server).
7. The error message is conditionally rendered **below the input field.**

This is a basic example, but in real-world applications, you may have multiple form fields and more complex validation rules.

Additionally, you can use third-party libraries like react-hook-form or formik for more advanced form handling and validation features.

- **How do you handle side effects in React?**

A side effect is an operation that affects something outside the scope of the component, such as making an API call, setting a timer, or updating the DOM directly.

Handling side effects is crucial in React because they can cause unintended consequences, like infinite loops or memory leaks.

In this explanation, we'll dive into how to handle side effects in React.

Key Components to Handle Side Effects in React:

- **Side effects**: Operations that affect something outside the component's scope.
- **UseEffect Hook:** A built-in React hook that allows you to run side effects in functional components.
- **Cleanup functions**: Functions that clean up after side effects to prevent memory leaks.

Step-by-Step Explanation

1. **Identify the side effect:** Determine what operation is causing the side effect, such as making an API call or setting a timer.
2. **Use the useEffect hook:** Wrap the side effect in a useEffect hook to ensure it runs only when necessary.
3. **Specify dependencies:** Provide an array of dependencies that trigger the side effect. If the dependencies change, the side effect will re-run.
4. **Return a cleanup function:** Return a function that cleans up after the side effect to prevent memory leaks.

Example

Suppose we want to fetch data from an API when a component mounts.

We can use the useEffect hook to handle this side effect:

In this example, the useEffect hook fetches data from the API when the component mounts and sets the data state.

The cleanup function logs a message to the console when the component is unmounted.

- **What is the purpose of the key prop in React?**

when rendering a list of items, it's essential to provide a unique identifier for each item. This is where the key prop comes into play.

The key prop is a crucial concept in React that helps the library keep track of the items in a list and optimize the rendering process.

When you render a list of items in React, you typically use a map function to iterate over the array of items. For example:

```
const items = ['apple', 'banana', 'orange'];

return (
  <ul>
    {items.map((item) => (
      <li>{item}</li>
    ))}
  </ul>
);
```

In this example, React will render a list of three items.

However, when the items array changes, React needs to determine which items have changed, added, or removed.

This is where the key prop comes in.

By providing a unique key prop for each item, you help React identify each item in the list.

For example:

```
const items = ['apple', 'banana', 'orange'];

return (
  <ul>
    {items.map((item, index) => (
      <li key={index}>{item}</li>
    ))}
  </ul>
);
```

In this example, we're using the index of each item as the key prop.

This tells React that each item has a unique identifier, which is its index in the array

- **How do you implement caching in a React application?**

To implement caching in a React application, follow these steps:

1. **Choose a caching library:** Select a caching library that suits your application's needs, such as react-query or redux-cache.
2. **Define the cache storage**: Set up the cache storage area, which can be a simple object or a more complex storage solution like Redis.
3. **Create a cache key**: Generate a unique cache key for each piece of data to be cached.
4. **Fetch data and cache it:** When fetching data from an API or database, store the data in the cache using the cache key.
5. **Retrieve data from cache:** When the application needs the data, check the cache first and retrieve the data if it exists.
6. **Invalidate cache:** Implement a mechanism to update or remove cached data when it becomes outdated.

Example: Caching API Data with React Query

Suppose we have a React application that fetches a list of users from an API.

We can use react-query to cache the API data.

Here's an example:

```
import { useQuery, useQueryClient } from 'react-query';

function fetchUsers() {
  return fetch('https://api.example.com/users').then(response =>
response.json());
}

function Users() {
  const { data, error, isLoading } = useQuery(
    'users', // cache key
    fetchUsers
  );

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

  return (
    <ul>
      {data.map(user => (
        <li key={user.id}>{user.name}</li>
      ))}
    </ul>
  );
}
```

In this example, react-query caches the API data using the users cache key.

When the component is rendered, it checks the cache first and retrieves the data if it exists.

If not, it fetches the data from the API and caches it.

- **How do you implement code splitting in a React application?**

To implement code splitting in a React application, I follow these steps:

1. Identify Components to Split: Determine which components or features can be loaded lazily, such as a login page or a dashboard.

2. Create a Lazy Component: Wrap the component to be loaded lazily with the React.lazy function, which returns a promise that resolves to the component.

3. Use Dynamic Imports: Use dynamic imports to load the lazy component only when it is needed. This can be achieved using the import() function.

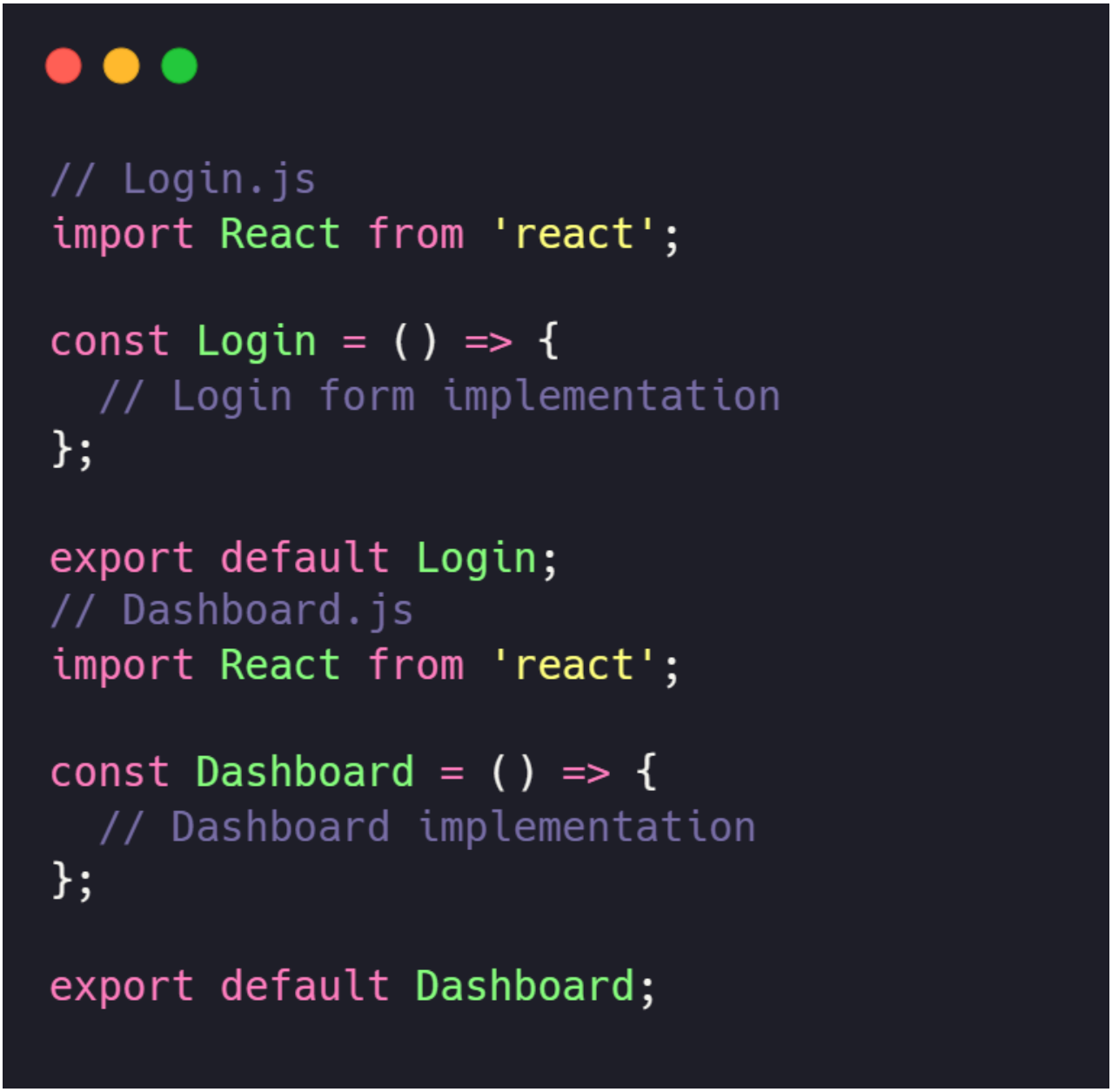
4. Handle Loading States: Use a loading indicator or a fallback component to handle the loading state while the lazy component is being loaded.

5. Optimize Server-Side Rendering: Ensure that server-side rendering is optimized to handle code splitting, by using techniques such as server-side rendering with lazy loading.

Example

Suppose we have a React application with a login page and a dashboard.

We can implement code splitting by lazy loading the dashboard component only when the user logs in successfully.

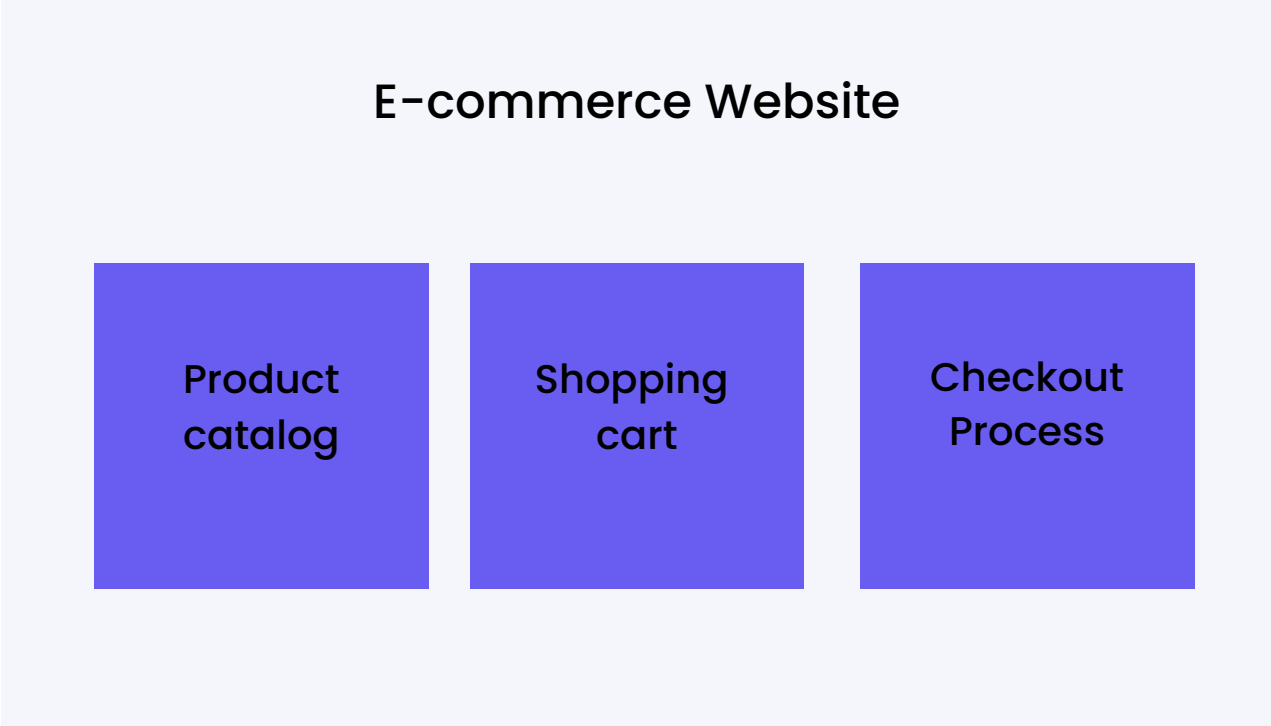


```
// Login.js
import React from 'react';

const Login = () => {
  // Login form implementation
};

export default Login;
// Dashboard.js
import React from 'react';

const Dashboard = () => {
  // Dashboard implementation
};

export default Dashboard;
```

```jsx
// App.js
import React, { Suspense, lazy } from 'react';
import Login from './Login';

const Dashboard = lazy(() => import('./Dashboard'));

const App = () => {
  return (
    <div>
      <Login />
      <Suspense fallback={<div>Loading...</div>}>
        <Dashboard />
      </Suspense>
    </div>
  );
};

export default App;
```

- **What is the difference between monolithic and micro-frontends architecture?**

Monolithic Architecture



Micro-frontends Architecture

A monolithic architecture is a traditional approach where the entire application is built as a single, inseparable unit.

It's like building a house with all the rooms connected together, making it difficult to modify or update individual components without affecting the entire structure.

On the other hand, a micro-frontends architecture is like building a house with separate, self-contained modules or rooms that can be developed, deployed, and maintained independently.

Each micro-frontend is responsible for a specific feature or functionality of the application, allowing teams to work in parallel and scale more efficiently.

Let me give you a small example.

Imagine you're building an e-commerce website with a monolithic architecture.

Every component, such as the product catalog, shopping cart, and checkout process, would be tightly coupled and bundled together.

If you wanted to update the shopping cart feature, you'd have to redeploy the entire application, risking potential bugs or downtime.

With a micro-frontends approach, the e-commerce website would be broken down into separate, independently deployable micro-frontends.

The product catalog could be one micro-frontend, the shopping cart another, and the checkout process a third.

Each team could work on their respective micro-frontend without affecting the others, enabling faster development cycles and easier maintenance

- **How do you implement Authentication and Authorization in a React application?**

Authentication in React involves verifying the user's identity, usually by checking their credentials (username/email and password) against a server or third-party authentication service.

For example, you can use Firebase Authentication or Auth0 to handle user sign-in and sign-up.

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

export const AuthContext = createContext();

export const AuthProvider = ({ children }) => {
  const [user, setUser] = useState(null);

  const login = (userData) => {
    // Authenticate the user with a server or third-party
service
    setUser(userData);
  };

  const logout = () => {
    setUser(null);
  };

  return (
    <AuthContext.Provider value={{ user, login, logout }}>
      {children}
    </AuthContext.Provider>
  );
}
```

Authorization determines what resources or features a user can access based on their authenticated role or permissions.

For instance, an admin user might have access to certain routes or components that a regular user cannot access.

```
import React, { useContext } from 'react';
import { AuthContext } from './AuthContext';
import { Route, Redirect } from 'react-router-dom';

const ProtectedRoute = ({ component: Component, ...rest }) => {
  const { user } = useContext(AuthContext);

  return (
    <Route
      {...rest}
      render={(props) =>
        user ? (
          <Component {...props} />
        ) : (
          <Redirect to="/login" />
        )
      }
    />
  );
};

export default ProtectedRoute;
```

To implement authentication and authorization in React, you typically use a state management library like React Context API or Redux to manage the user's authentication state and permissions.

You can create a higher-order component or a route component that checks the user's authentication status and permissions before rendering the protected content.

- **What are the Different ways of defining a Components?**

1. Functional Components

A functional component is a pure function that takes in props and returns JSX elements.

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

2. Class Components

A class component is a class that extends the React. Component class and has a render method.

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

3. Arrow Function Components

An arrow function component is a concise way to define a functional component using an arrow function.

```
const HelloComponent = (props) => <h1>Hello,
{props.name}!</h1>;
```

4. Class Components with Constructor

A class component with a constructor is similar to a class component, but it also has a constructor method.

```
class HelloComponent extends React.Component {
  constructor(props) {
    super(props);
    this.state = { name: 'John' };
  }

  render() {
    return <h1>Hello, {this.state.name}!</h1>;
  }
}
```

5. Higher-Order Components (HOCs)

A higher-order component is a function that takes in a component as an argument and returns a new component with additional props or behavior.

```
function withLoadingIndicator(WrappedComponent) {
  return function EnhancedComponent({ props }) {
    if (props.isLoading) {
      return <div>Loading...</div>;
    }
    return <WrappedComponent {...props} />;
  };
}
```

- **How to manage events in React?**

In React, event management is crucial to handle user interactions with your application.

To manage events in React, you can follow these steps:

Step 1: Define the Event Handler

Create a function that will handle the event. This function is usually defined as a method in your component's class.

Step 2: Attach the Event Handler

Attach the event handler to the element that will trigger the event. You can do this by adding an onClick, onChange, or other event attribute to the element.

Step 3: Pass the Event Object

Pass the event object as an argument to the event handler function.

This allows you to access the event details, such as the target element or the event type.

Example:

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

class Button extends Component {
  handleClick = (event) => {
    console.log('Button clicked!');
    console.log(event.target); // The button
element
  }

  render() {
    return (
      <button onClick={this.handleClick}>Click me!
</button>
    );
  }
}
```

The handleClick function is defined as an event handler.

It is attached to the button element using the onClick attribute.

When the button is clicked, the handleClick function is called with the event object as an argument.

- **Describe how your ideal folder structure.**

- **What is the purpose of React Query?**

  React Query is a powerful **data-fetching library** for React applications.

  Its primary purpose is to simplify the management of asynchronous **data fetching, caching, and synchronization** in React components.

  React Query helps in:

  1. Declarative Data Fetching
  - It provides hooks like useQuery to fetch data declaratively
  - No need to manage state and lifecycle methods manually

  2. Caching
  - Automatically caches data for future use
  - Prevents unnecessary network requests
  - Ensures consistent data across components

  4. Stale Data Management
  - Tracks when data becomes stale (outdated)
  - Allows re-fetching or background updates
  - Ensures fresh data is displayed

  5. Optimistic Updates
  - Allows updating the UI optimistically
  - Before the server response arrives
  - Improves perceived performance

  6. Error Handling
  - Provides a consistent way to handle errors
  - Retry on error or stale data
  - Display error states in the UI

Here's an example using useQuery hook:

```
import { useQuery } from 'react-query';

function UserDetails({ userId }) {
  const { data, isLoading, error } = useQuery(['user', userId], fetchUser);

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

  return (
    <div>
      <h2>{data.name}</h2>
      <p>Email: {data.email}</p>
    </div>
  );
}

// Function to fetch user data from an API
async function fetchUser(key, userId) {
  const response = await fetch(`/api/users/${userId}`);
  return response.json()
}
```

- **What is the purpose of the React.StrictMode component? How can it help identify potential issues in an application?**

1. Highlighting Potential Problems

StrictMode helps detect issues such as deprecated APIs, side effects within lifecycle methods, and unsafe lifecycles.

It ensures that components are following best practices and will remain compatible with future React updates.

2. Identifying Side Effects

It intentionally invokes certain lifecycle methods (like componentDidMount and componentDidUpdate) twice in development mode.

This helps developers identify unexpected side effects that might not be apparent during a single execution.

By using React.StrictMode, developers can catch and address potential problems early in the development process, leading to more robust and maintainable applications.

Here's an example of how to wrap parts of your application with React.StrictMode:

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

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

In this example, the entire App component tree will be subject to the additional checks provided by StrictMode, helping to ensure a higher quality of code.

- **What are some common React architecture patterns?**

1. Component-Based Architecture This is the most basic pattern in React. Think of your app like building blocks:

- You make small, reusable pieces (components)
- You put these pieces together to build your app
- Each piece can be used many times

How to do it: Break your app into small parts. Make each part a separate component. Use these components to build your app.

2. Container and Presentational Components This pattern splits components into two types:

- **Containers:** These are the "smart" components. They handle data and logic. ↳ class comp
- **Presentational:** These are the "dumb" components. They just show things and look pretty. ↳ fn comp

How to do it: Make container components that fetch and manage data. Then make presentational components that just receive props and display them.

3. Higher-Order Components (HOCs) This is like a special wrapper for your components:

- It's a function that takes a component and returns a new one
- It adds extra features to your component

How to do it: Write a function that takes a component as input. Add some extra stuff to it, then return the enhanced component.

4. Render Props This pattern lets you share code between components using a prop whose value is a function:

- One component passes a function to another
- The receiving component uses this function to render something

How to do it: Create a component that takes a function as a prop. Use this function in the render method to decide what to show.

5. Hooks Hooks are a newer way to use state and other React features without writing a class:

- They let you use state and lifecycle features in function components
- There are built-in hooks like useState and useEffect, and you can make your own

How to do it: Create a component that takes a function as a prop. Use this function in the render method to decide what to show.

- **Explain about useCallback() Hook.**

Defn of memoise

useCallback() allows you to **memoize a function.** Memoization means that the **function is stored and only re-created when its dependencies change**, rather than on every render.

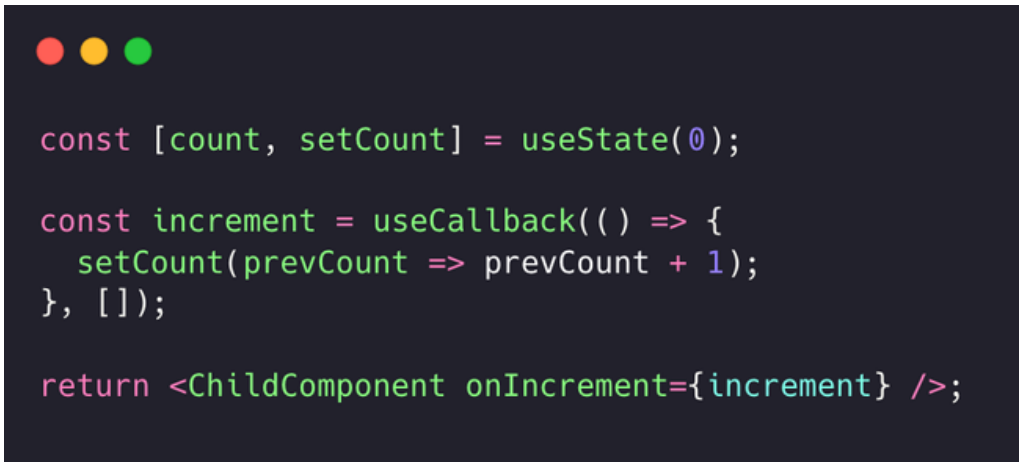
This can improve **performance** by preventing unnecessary re-creations of functions, particularly when passing them as props to child components that rely on reference equality to avoid re-rendering. Here's how useCallback works:

Dependencies Array: The second argument to useCallback is an array of dependencies. The function is only re-created if one of these dependencies changes.

When to Use useCallback:

- **Performance Optimization:** It's useful when you pass callbacks to optimized child components that rely on React.memo or shouldComponentUpdate. It ensures that the callback doesn't change unless necessary.
- **Avoiding Unnecessary Re-renders**: If a child component re-renders based on prop changes and you pass a new function on every render, useCallback can help prevent that.

Example:



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

const increment = useCallback(() => {
  setCount(prevCount => prevCount + 1);
}, []);

return <ChildComponent onIncrement={increment} />;
```

Here, **increment** will only be re-created if its dependencies (none in this case) change, avoiding unnecessary re-renders of **ChildComponent** if it uses **React.memo**.

- **Explain about useMemo() Hook.**

useMemo() allows you to memoize the result of a computation. This means that useMemo will **only re-calculate the value when its dependencies change,** rather than on every render.

This is useful for optimizing performance, especially for expensive calculations that don't need to be re-executed unless certain values change.

How useMemo Works

Here's the basic syntax:

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

- **Callback Function:** The first argument is a function that returns the value you want to memoize. This function only runs when one of the dependencies changes.
- **Dependencies Array:** The second argument is an array of dependencies. The memoized value is only recalculated if one of these dependencies changes.

When to Use useMemo

- **Performance Optimization**: Use useMemo when you have a computationally expensive operation that doesn't need to run on every render. By memoizing the result, you can prevent unnecessary recalculations.
- **Avoiding Unnecessary Calculations**: When certain values or derived data are expensive to compute and don't change often, useMemo ensures that the computation happens only when necessary.

Example:

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

const expensiveCalculation = useMemo(() => {
  console.log("Calculating...");
  return count * 2;
}, [count]);

return (
  <div>
    <p>Expensive Calculation Result: {expensiveCalculation}</p>
    <button onClick={() => setCount(count + 1)}>Increment Count</button>
    <button onClick={() => setOtherCount(otherCount + 1)}>Increment Other Count</button>
  </div>
);
```

The expensiveCalculation is memoized, so it only recalculates when count changes. If otherCount changes and the component re-renders, expensiveCalculation won't recompute unless count also changes, improving performance.

- **Explain about useContext() Hook.**

The **useContext** hook is used to access the value of a context directly in a functional component. Instead of passing data through multiple levels of components, **useContext** allows you to get the data from a context without props drilling.

How It Works:

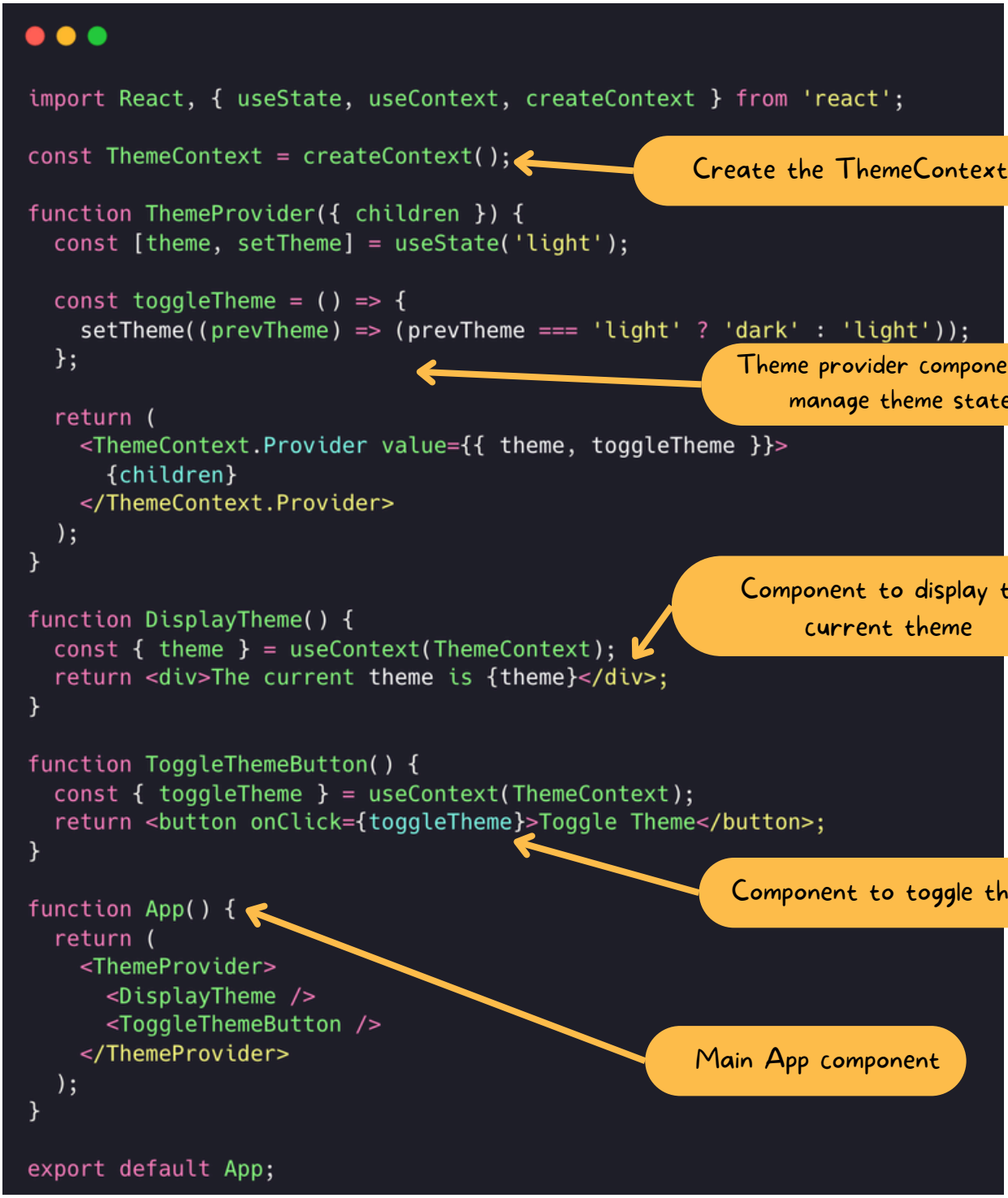
- **Context Creation:** First, you create a context using React.createContext().
- **Provider:** Then, wrap the part of your app that needs the context with a Context.Provider. This allows you to supply the value that will be available to any component within that tree.
- **useContext Hook:** Inside a component, you use useContext to access the value from the nearest context provider.

Use Cases:

1. **Theming**: Share a theme (e.g., dark or light mode) across the entire application.
2. **Authentication**: Manage user authentication state globally, so every component knows if a user is logged in.
3. **Localization**: Share language preferences across the app to support multiple languages.
4. **Global State Management:** Manage state that needs to be accessed by multiple components without passing props.

Example:
In below example, **ThemeContext** is a centralized object that holds the current theme data (like light or dark mode).

The **ThemeProvider** component manages this theme state and provides it to other components in the app. **DisplayTheme** uses the **useContext** hook to access and display the current theme, while **ToggleThemeButton** also uses **useContext** to access a function that toggles the theme when clicked, allowing the app to switch between light and dark modes.


import React, { useState, useContext, createContext } from 'react';
const ThemeContext = createContext();
function ThemeProvider({ children }) {
const [theme, setTheme] = useState('light');
const toggleTheme = () => {
setTheme((prevTheme) => (prevTheme === 'light' ? 'dark' : 'light'));
};
return (
<ThemeContext.Provider value={{ theme, toggleTheme }}>
{children}
</ThemeContext.Provider>
);
}
function DisplayTheme() {
const { theme } = useContext(ThemeContext);
return The current theme is {theme} ;
}
function ToggleThemeButton() {
const { toggleTheme } = useContext(ThemeContext);
return <button onClick={toggleTheme}>Toggle Theme</button>;
}
function App() {
return (
<ThemeProvider>
<DisplayTheme />
<ToggleThemeButton />
</ThemeProvider>
);
}
export default App;
Create the ThemeContext
Theme provider component to manage theme state
Component to display the current theme
Component to toggle the theme
Main App component

- **What is the purpose of the React Router's useParams hook?**

  The useParams hook in React Router is used to get data from the URL. It helps us read dynamic parts of the URL so our app can use it.

  For example, if we have a URL like /user/123, we can get '123' using useParams. This is useful when we want to show details of different users.

  Here's a simple example:

```
import { BrowserRouter as Router, Route, Switch } from 'react-router-dom';
import User from './User';

function App() {
  return (
    <Router>
      <Switch>
        <Route path="/user/:id">
          <User />
        </Route>
      </Switch>
    </Router>
  );
}

export default App;

// User.js
import { useParams } from 'react-router-dom';

function User() {
  let { id } = useParams();
  return (
    <div>
      User ID: {id}
    </div>
  );
}

export default User;
```

Use cases:

1. **User Profiles:** Websites like Facebook show each user's profile using their unique ID in the URL.
2. **Product Pages:** E-commerce sites use product IDs in URLs to display details of each product.
3. **Blog Posts:** Blogging platforms use post IDs or slugs to show individual articles.
4. **Order Tracking:** Online stores use order numbers in URLs to let customers track their orders.

These examples show how useParams helps make URLs dynamic and user-specific

- **What is the difference between useLayoutEffect and useEffect?**

In React, useEffect and useLayoutEffect are two hooks used for side effects, but they work slightly differently.

useEffect runs the effect after the component renders and the paint has been done. This means it doesn't block the rendering process, allowing the browser to update the screen first.

It's useful for tasks like fetching data, subscription, or changing the DOM very late.

Here's a simple example of useEffect:

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

function MyComponent() {
    useEffect(() => {
        document.title = 'Hello!';
    });

    return <div>Hello World</div>;
}
```

In this example, useEffect runs after the render, changing the document title to 'Hello!'.
This won't delay the painting of the page.

useLayoutEffect, on the other hand, runs after the DOM update but before the browser paints. It blocks the screen update until it completes.

It can be useful for measuring DOM elements and reading layout before another paint happens.

Here's how you could use useLayoutEffect:

```
import { useLayoutEffect, useRef } from 'react';

function Example() {
    const divRef = useRef(null);

    useLayoutEffect(() => {
        const { height } = divRef.current.getBoundingClientRect();
        console.log('Height:', height);
    }, []);

    return <div ref={divRef}>Hello Layout</div>;
}
```

In this case, useLayoutEffect ensures we read the div's height before painting.

Use cases:

1. **Form Auto Resize:** If you need to measure and adjust form sizes before they appear onscreen to prevent awkward jumps, use useLayoutEffect.
2. **Synchronized Animations:** When working on animations based on user actions that need synchronized reaction, useLayoutEffect helps calculate positions.
3. **Dynamic Styling:** Use it to read dimensions or styles from the DOM to apply new styles correctly before users see the page.
4. **Canvas:** If drawing on a canvas requires item positioning based on other elements, useLayoutEffect ensures element look before the canvas renders

Both hooks are handy depending on timing needs. Using them wisely helps manage component behavior and improve performance.

- **How do you Test Asynchronous code in React components? Can you provide an example?**

When testing asynchronous code in React components, we typically want to ensure that our component behaves correctly when it changes state based on async operations like API calls or timeouts.

The most common way to handle this is by using testing libraries such as React Testing Library or Enzyme.

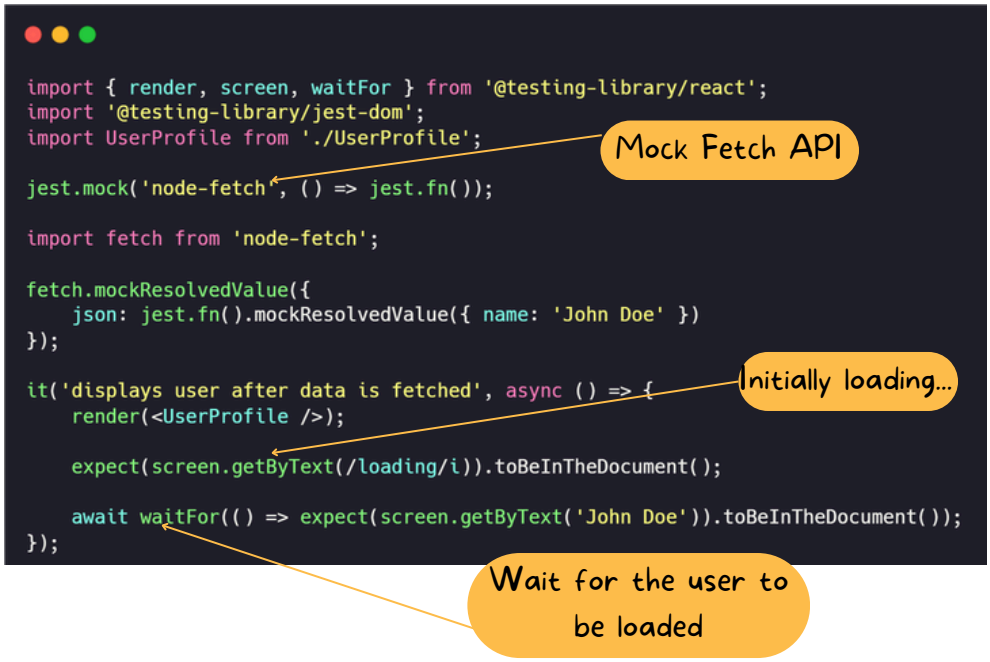
Consider using React Testing Library since it focuses on testing the behavior in a way similar to how a user would interact with the application.

First, we set up our component and the async function.

For example, imagine a component that fetches user data from an API and displays it:

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

    React.useEffect(() => {
        fetch('/api/user')
            .then(response => response.json())
            .then(data => setUser(data));
    }, []);

    return user ? <div>{user.name}</div> : <div>Loading...</div>;
}
```

To test this, we simulate the API call and check that our component displays the data correctly:



Use cases:

1. **Fetching Data:** Many apps need to retrieve data from a server, like user info or product listings.
2. **Form Submissions:** After a user submits a form, the app often sends data to a server and waits for a response.
3. **Loading Resources:** Apps might load scripts or images asynchronously to improve initial load time.
4. **Background Updates:** Apps could fetch updated information without needing a complete page reload, like refreshing a news feed.

- **What are Snapshot Tests, and when should they be used in a React application?**

Snapshot tests in React are a way to test your UI by keeping a copy of what your UI component looks like at a certain point in time.

These tests help you make sure your UI doesn't change unexpectedly.

When you run a snapshot test, it captures the rendered output of a component and saves it as a **snapshot file**.

Later, when you change your code and rerun the tests, it compares the new UI output with the stored snapshot.

Here's a simple example: You have a React component called **MyButton**. It returns a button with the text **Click me!**.

When you first write the snapshot test, a snapshot of this button UI will be saved.

```
import React from 'react';
import renderer from 'react-test-renderer';
import MyButton from './MyButton';

test('MyButton renders correctly', () => {
    const tree = renderer.create(<MyButton />).toJSON();
    expect(tree).toMatchSnapshot();
});
```

Now, if you or anyone else changes the button's text to **Press me!**, the next test run will fail if you do not update the snapshot.

This failure prompts you to review if the UI change was intentional.

Use cases:

1. **UI Libraries:** When working with UI component libraries like Material-UI, snapshot tests ensure components render as expected across updates.
2. **Design Consistency:** Ensures your design stays consistent over time, especially helpful in big teams where many people contribute.

3. Detecting Accidental Changes: If a team's refactoring unintentionally alters the UI, snapshots can catch these changes early.
4. **Legacy Code:** Useful in maintaining projects where understanding the whole codebase is hard. Snapshot tests offer a safety net by letting you know if the UI changes.

- **How can you mock API calls in your tests to ensure they do not rely on external services?**

To test a React app without relying on actual external services, you can mock API calls.

This ensures that your tests run quickly and don't fail due to issues with real APIs.

Mocking allows us to simulate different API responses and states

In React, you can use tools like Jest to mock API calls. Here is a simple way to mock an API call using Jest:

Suppose you have a function to fetch user data

We used **jest.fn()** to create a mock function for **fetch**. It returns a promise that resolves to a fake response when called.

This lets you test the behavior of **fetchUser** without making an actual network request.

Use cases:

- **Speed:** Tests that don't call real APIs run faster, making it easier to run them frequently.
- **Stability:** Mocking prevents issues due to API downtime or rate limiting from breaking your tests.
- **Consistency:** You can ensure the same API responses each time, which is useful for debugging.
- **Security:** Avoid hitting production APIs during tests, which can prevent data corruption.

By mocking API calls, you make your tests more reliable and efficient, ensuring your application behaves as expected without depending on external factors.

- **Describe How you would Test a component that relies on props passed from its parent component?**

When testing a React component that relies on props from its parent component, you want to ensure it behaves correctly based on the props it receives.

There's no need to focus on how the parent component works; just make sure the component under test functions as expected.

A good way to start is by using testing tools like Jest and React Testing Library. They help simulate how the component acts in a real environment.

First, think about the props that component expects.

Let's say we have a component **Greeting** that takes a **name** prop and displays a greeting message.

```
function Greeting({ name }) {
    return <h1>Hello, {name}!</h1>;
}
```

To test it, you'd write a test that renders this component with different name values.

Use React Testing Library to render the component and check if the correct output is displayed.

```
import { render, screen } from '@testing-library/react';
import Greeting from './Greeting';

test('displays the correct greeting message', () => {
    render(<Greeting name="John" />);
    const greetingElement = screen.getByText(/Hello, John!/i);
    expect(greetingElement).toBeInTheDocument();
});

// Test for different name

test('handles a different name correctly', () => {
    render(<Greeting name="Jane" />);
    const greetingElement = screen.getByText(/Hello, Jane!/i);
    expect(greetingElement).toBeInTheDocument();
})
```

This test checks if the **Greeting** component displays the correct message depending on the **name** prop.

Use cases:

- **User Profile Display:** Components in dashboard applications often need to display user details like name and email. Testing ensures the profile displays correctly when provided specific user data.
- **E-commerce Product Page:** Product components rely on product information props like name, price, and description. Testing ensures each product displays accurate details that match what the parent passes.
- **Notification System:** A component that shows notifications might rely on a message prop. Tests check if the notifications appear with the right text.
- **Form Components:** A form input component might take props like label or placeholder text. Testing ensures that these appear correctly and react to user input as expected.

Overall, testing with props focuses on whether the component does the right thing with the data it gets.

- **How do you Handle Testing for user interactions (e.g., clicks, form submissions) in React Components?**

When testing user interactions in React components, like clicks or form submissions, we usually rely on tools like React Testing Library and Jest.

These tools help ensure our user interface works correctly when a user interacts with it.

First, let's understand what we're testing. Imagine we have a simple button in our React component that increments a counter every time it's clicked.

Here's a basic example of a React component:

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

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

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

export default Counter;
```

To test this, we can use React Testing Library with Jest. Our goal is to simulate a click on the button and check if the counter increments.

Here's how we write the test:

```
import { render, screen, fireEvent } from '@testing-library/react';
import Counter from './Counter';

test('increments counter after button click', () => {
    render(<Counter />);

    const button = screen.getByText(/click me/i);
    const counterText = screen.getByText(/you clicked 0 times/i);

    fireEvent.click(button);

    expect(counterText).toHaveTextContent('You clicked 1 time');
});
```

In this test, **render** sets up the component in a virtual DOM. **screen.getByText** fetches elements by their text content.

fireEvent.click simulates the click action, and **expect** checks the result.

It is Used in:

- **E-commerce checkout page:** Testing interactions like adding items to a cart and submitting payment info.
- **Social media post creation:** Ensuring form submissions for creating posts work properly.
- **Online survey:** Testing user answers submission.
- **Interactive dashboards:** Verifying that filters and buttons work when clicked.

Good testing helps catch problems early and gives users a smooth experience.

- **Can you explain the concept of code coverage and how it relates to the quality of your tests in a React application?**

Code coverage is a term used to describe how much of your code is tested by your test suite. It's often shown as a percentage.

For example, if you have 100 lines of code, and your tests cover 90 of them, your code coverage is 90%.

In a React application, you use code coverage to see how many parts of your components and functions are checked by your tests.

It helps you understand which parts of your code may have bugs because they haven't been tested yet.

Having high code coverage usually means you've written enough tests to check most of your code.

But remember, just because you have high code coverage doesn't always mean your tests are good.

You also need to make sure your tests check the right things, like edge cases and different user actions.

Here's a simple React component and a test to show how you might check code coverage:

```
function WelcomeMessage({ name }) {
    return
    Hello, { name }!

    ;
}


import { render } from '@testing-library/react';

test('displays correct welcome message', () => {
    const { getByText } = render();
    expect(getByText("Hello, John!")).toBeInTheDocument();
});
```

In this example, the test checks if the component correctly displays the welcome message for a user named “John.”

Running coverage tools will show if the parts of the WelcomeMessage function have been tested.

Use cases:

- Finding untested parts of your code helps focus testing efforts on critical areas.

- Ensuring new code is tested can prevent future bugs, especially when new features are added.
- Code reviews use coverage reports to see if new changes include adequate tests.
- Teams use coverage goals to maintain overall test quality as the codebase grows.

In short, while code coverage is a useful metric, remember it's not just about the numbers.

You should strive for meaningful tests that genuinely check your application's functionality.

- **What strategies would you use to ensure your tests are maintainable and easy to understand??**

When writing tests in React, it's important to keep them clear and easy to update. Here are a few strategies:

First, follow the **AAA** pattern **(Arrange, Act, Assert)**. This means setting up your data and objects (Arrange), doing the actions you're testing (Act), and checking the results (Assert).

This way, your tests are easy to read and understand.

Second, use descriptive names for your test cases and explain what each test is doing.

Third, make use of testing libraries like React Testing Library. Such libraries focus on testing the user's point-of-view, leading to more meaningful tests.

Here is a simple React test example:

```
import { render, screen, fireEvent } from '@testing-library/react';
import MyButton from './MyButton';

test('renders button correctly', () => {
  render(<MyButton />);
  const buttonElement = screen.getByText(/click me/i);
  expect(buttonElement).toBeInTheDocument();
});

test('button click changes text', () => {
  render(<MyButton />);
  const buttonElement = screen.getByRole('button');
  fireEvent.click(buttonElement);
  const updatedText = screen.getByText(/clicked/i);
  expect(updatedText).toBeInTheDocument()
});
```

Fourth, ensure tests are independent from each other. This means one test should not rely on the result or state of another.

This way, you can run them in any order.

Lastly, keep your test files organized by grouping related tests together. This helps in maintaining them and ensures that all related tests are in one place.

These strategies help when:

- Updating a UI component and ensuring all tests for that component are still valid.
- Collaborating with other developers who need to understand your test logic quickly.
- Refactoring code; with clear tests, you know immediately if changes break anything.
- Adding new features; existing tests ensure new code doesn't mess up old features.

By using these strategies, you ensure that your tests serve as a reliable safety net, allowing for confident updates to your code.

Stay ahead with daily updates!

Follow us on social media for useful web development content.

@richwebdeveloper

@new_javascript

@developerupdates

@developerupdates

@_chetanmahajan

Note: This kit is continuously updated. Please continue to check Gumroad or our website (where you purchased this) for updated questions and answers. You will receive all updates for FREE

Download from our Website

Download on Gumroad

WWW.DEVELOPERUPDATES.COM