1) https://fakestoreapi.com/products/1,https://fakestoreapi.com/carts?limit=5,https://fakestoreapi.com/products

show items in cart and make dropdown to update cart and on each product to view product details

1. How do I fetch products from the Fake Store API in React?

You can use the fetch API or libraries like axios to retrieve data. Here's an example using fetch:

Javascript

Copy code

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

const Products = () => {

const [products, setProducts] = useState([]);

useEffect(() => {

fetch("https://fakestoreapi.com/products")

.then((response) => response.json())

.then((data) => setProducts(data))

.catch((error) => console.error("Error fetching products:", error));

}, []);

Async await

The async and await keywords in JavaScript provide a cleaner and more readable way to handle asynchronous operations, compared to traditional promise chaining. Here's a concise explanation:

1. async Keyword

Declares a function as asynchronous.

An async function always returns a Promise.

If the function explicitly returns a value, it is wrapped in a resolved Promise.

If an error is thrown, it is wrapped in a rejected Promise.

Example:

Javascript

Copy code

async function fetchData() {

return "Data fetched!";

}

fetchData().then((result) => console.log(result)); // Output: Data fetched!

2. await Keyword

Can only be used inside an async function.

Pauses the execution of the function until the Promise is resolved or rejected.

Makes asynchronous code look synchronous, improving readability.

Example:

Javascript

Copy code

async function fetchData() {

const response = await fetch("https://api.example.com/data");

const data = await response.json();

console.log(data);

}

fetchData();

//BASIC QUESTION/////////

1,2,3) https://medium.com/@rishabhraikwar77/all-about-react-hooks-and-their-use-cases-f31f11ad68d4

4,5) https://www.geeksforgeeks.org/reactjs/state-management-in-react-hooks-context-api-and-redux/

https://www.geeksforgeeks.org/node-js/what-is-the-use-of-middleware-redux-thunk/

Redux Flow

1

2

Redux is a state management library for JavaScript applications, commonly used with React. It helps manage the application state in a predictable way, making it easier to debug and maintain. The typical flow of data in a React-Redux application involves several key concepts: store, actions, reducers, and components.

Key Concepts

Redux Store: The store is a single object that holds the entire state of the application. It can be accessed by any component through the connect method.

Action Creators: These are functions that return actions, which are plain objects with a type property and an optional payload. Actions are dispatched when the user interacts with the UI or at certain points in a component's lifecycle.

Reducers: Reducers are pure functions that take the previous state and an action as inputs and return a new state. They determine how the state should change in response to an action.

Data Flow in a React-Redux Application

The flow of data in a React-Redux application is unidirectional and follows these steps:

User Interaction: The user interacts with the UI, triggering an event such as a button click.

Dispatching Actions: The event handler calls an action creator, which returns an action. This action is dispatched to the Redux store

1

.

Reducers: The dispatched action is received by the root reducer, which passes it to all the smaller reducers. Each reducer checks if it needs to update the state based on the action type

1

.

State Update: If a reducer matches the action type, it returns a new state object with the necessary updates. The state is immutable, so a new state object is always created

1

.

Component Re-render: The store informs the connected components about the new state. The components then retrieve the updated state and re-render accordingly

1

.

In this example, the App component is connected to the Redux store. When the button is clicked, the showMessage method dispatches the GetMessage action, which updates the state and re-renders the component with the new message

1

.

Conclusion

Redux simplifies state management in large-scale applications by providing a predictable state container. The unidirectional data flow ensures that the state changes are traceable and easier to debug

1

2

. By understanding the flow of data in a React-Redux application, developers can build more maintainable and scalable applications.

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

6) React component lifecycle

React Component Lifecycle

In React, the component lifecycle refers to the series of events that occur from the creation of a component to its removal from the DOM. It is divided into three main phases:

1. Mounting Phase

This is when a component is created and inserted into the DOM.

Methods:

constructor(): Initializes the component's state and binds methods.

static getDerivedStateFromProps(props, state): Updates the state based on props before rendering.

render(): Returns the JSX to render the component.

componentDidMount(): Executes after the component is mounted. Ideal for API calls or DOM manipulations.

2. Updating Phase

This occurs when a component's state or props change, causing it to re-render.

Methods:

static getDerivedStateFromProps(props, state): Updates state before rendering (also used here).

shouldComponentUpdate(nextProps, nextState): Determines whether the component should re-render (used for performance optimization).

render(): Re-renders the component.

getSnapshotBeforeUpdate(prevProps, prevState): Captures information (e.g., scroll position) before the DOM is updated.

componentDidUpdate(prevProps, prevState, snapshot): Executes after the component updates. Useful for side effects like fetching new data.

3. Unmounting Phase

This happens when a component is removed from the DOM.

Methods:

componentWillUnmount(): Executes cleanup tasks like removing event listeners or canceling API calls.

Special Notes

Error Handling: React provides componentDidCatch(error, info) and static getDerivedStateFromError(error) to handle errors in components.

Functional Components: With React Hooks (e.g., useEffect), lifecycle methods can be mimicked in functional components.

By understanding these phases and methods, you can control a component's behavior at different stages of its lifecycle effectively!

7) how to handle nested fetch

To handle nested fetch calls in JavaScript, you can chain promises to avoid deeply nested structures (commonly referred to as "callback hell"). Here's how you can do it effectively:

Example: Chaining Fetch Calls

Javascript// Example: Fetching user data, then fetching posts by that user

fetch('https://jsonplaceholder.typicode.com/users/1')

.then(response => {

if (!response.ok) {

throw new Error('Failed to fetch user data');

}

return response.json(); // Parse the user data

})

.then(user => {

console.log('User:', user);

// Use the user ID to fetch their posts

return fetch(`https://jsonplaceholder.typicode.com/posts?userId=${user.id}`);

})

.then(response => {

if (!response.ok) {

throw new Error('Failed to fetch posts');

}

return response.json(); // Parse the posts data

})

.then(posts => {

console.log('Posts:', posts); // Handle the posts data

})

.catch(error => {

console.error('Error:', error); // Handle any errors

});

Using Async/Await for Better Readability

Javascriptasync function fetchUserAndPosts() {

try {

const userResponse = await fetch('https://jsonplaceholder.typicode.com/users/1');

if (!userResponse.ok) {

throw new Error('Failed to fetch user data');

}

const user = await userResponse.json();

console.log('User:', user);

const postsResponse = await fetch(`https://jsonplaceholder.typicode.com/posts?userId=${user.id}`);

if (!postsResponse.ok) {

throw new Error('Failed to fetch posts');

}

const posts = await postsResponse.json();

console.log('Posts:', posts);

} catch (error) {

console.error('Error:', error);

}

}

fetchUserAndPosts();

Key Points

Error Handling: Always check response.ok to ensure the request was successful.

Avoid Nesting: Use promise chaining or async/await to keep the code clean and readable.

Scalability: For multiple dependent requests, async/await is generally easier to manage.

Both approaches work well, but async/await is often preferred for its simplicity and readability.

8) class vs functional

Difference Between Class and Functional Components in React

1

2

3

In React, components are the building blocks of the UI. They can be created using class components or functional components, each with distinct characteristics and use cases.

Functional Components

Functional components are JavaScript functions that accept props as input and return JSX. They are simpler and more concise, making them ideal for presentational components.

Example:

import React, { useState } from "react";

const FunctionalComponent = () => {

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

const increase = () => setCount(count + 1);

return (

<div>

<h1>Functional Component</h1>

<h2>Count: {count}</h2>

<button onClick={increase}>Increment</button>

</div>

);

};

export default FunctionalComponent;

Copy

Key Features:

State Management: Uses React Hooks like useState and useEffect for managing state and lifecycle events.

Lifecycle: Lifecycle methods are replaced by hooks (e.g., useEffect for componentDidMount or componentWillUnmount).

Performance: Lightweight and faster due to the absence of class overhead.

Code Simplicity: Requires less boilerplate, making it easier to read and maintain.

Class Components

Class components are ES6 classes that extend React.Component. They are more feature-rich but involve more complexity.

Example:

import React, { Component } from "react";

class ClassComponent extends Component {

constructor() {

super();

this.state = { count: 0 };

}

increase = () => {

this.setState({ count: this.state.count + 1 });

};

render() {

return (

<div>

<h1>Class Component</h1>

<h2>Count: {this.state.count}</h2>

<button onClick={this.increase}>Increment</button>

</div>

);

}

}

export default ClassComponent;

Copy

Key Features:

State Management: Uses this.state and this.setState() for managing state.

Lifecycle: Provides traditional lifecycle methods like componentDidMount, componentDidUpdate, and componentWillUnmount.

Event Handling: Requires explicit binding of methods to this unless using arrow functions.

Code Complexity: Involves more boilerplate, making it less concise compared to functional components.

Comparison

State Management: Functional components use hooks (useState), while class components rely on this.state.

Lifecycle: Functional components use useEffect for lifecycle events, whereas class components use dedicated lifecycle methods.

Performance: Functional components are generally faster due to their simpler structure.

Code Simplicity: Functional components are easier to write and maintain, especially for smaller components.

Error Boundaries: Only class components can be used as error boundaries.

When to Use Each

Functional Components: Preferred for most modern React development due to simplicity, performance, and hooks.

Class Components: Useful in legacy projects or when lifecycle methods like componentDidCatch (for error boundaries) are required.

9)https://abeer.hashnode.dev/lazy-loading-suspense-and-error-boundary-in-react-explained

10) performance react

Code Splitting in React

1

2

3

Code splitting is a technique used to improve the performance of a React application by splitting the code into smaller chunks that can be loaded on demand. This helps in reducing the initial load time of the application by only loading the necessary code for the current view.

Dynamic Imports

One of the most common ways to implement code splitting in React is through the use of dynamic imports. The import() function allows you to dynamically load a module only when it is needed. This function returns a Promise that resolves to the module's namespace object.

Example

// moduleA.js

const moduleA = 'Hello';

export { moduleA };

// App.js

import React, { Component } from 'react';

class App extends Component {

handleClick = () => {

import('./moduleA')

.then(({ moduleA }) => {

// Use moduleA

})

.catch(err => {

// Handle failure

});

};

render() {

return (

<div>

<button onClick={this.handleClick}>Load</button>

</div>

);

}

}

export default App;

Copy

In this example, moduleA.js and its dependencies are loaded only when the user clicks the 'Load' button

1

.

DEBOUNCE THROTTLING

Debouncing and throttling are techniques used to optimize the performance of functions triggered by events that occur frequently, such as scrolling, resizing, or typing. They help in managing how often these functions are executed, which can improve user experience and system performance. Here’s a detailed comparison of both techniques.

Debouncing is a technique where you delay the execution of a function until after a certain amount of time has passed. This is useful if you have a frequently used function—say, a scroll or resize event listener—and don't want to trigger it too frequently because that might slow down the browser.

Throttling is a similar technique to debouncing, but instead of delaying the execution of a function, it limits the rate at which a function. This is useful when a function, such as a mousemove or keydown event listener, may be called repeatedly but need not be run each time.

Throttling is a technique in which, no matter how many times the user fires the event, the attached function will be executed only once in a given time interval. Throttling ensures that the function executes at regular intervals.

https://www.geeksforgeeks.org/javascript/difference-between-debouncing-and-throttling/

11) React Strict Mode is a development tool that helps catch common bugs and side effects by running extra checks that simulate production-like behaviors, such as identifying unsafe lifecycles and preparing your app for upcoming features like concurrent rendering.

https://www.bairesdev.com/blog/react-strict-mode/

12) https://www.geeksforgeeks.org/reactjs/how-to-use-the-usehistory-hook-in-react-router/

https://www.w3schools.com/react/react\_router.asp